O'REILLY®

Третье издание

Head First

Программирование для Android на Kotlin

Дон Гриффитс
Дэвид Гриффитс

ПОДАРОК ДЛЯ МОЗГА

Отзывы о книге

Разработка Android-приложений полностью изменилась за последние годы, из-за чего написать подобную книгу стало в высшей степени непросто. В третьем издании своей классической книги Гриффиты, как обычно, превосходно справились с объяснением того, как осуществляется разработка современных приложений Android, для чего им пришлось почти полностью переписать свою книгу. В результате мы снова получили лучшую из существующих книг в этой области. Если вы хотите освоить правильный подход к построению Android-приложений, купите эту книгу.

— Кен Каусен, президент Kousen IT, Inc.

Создать учебный материал, который был бы одновременно доступным и точным, невероятно сложно, однако у Гриффитов это получилось. Они объясняют многие сложные моменты Android настолько ясно и доступно, что они понятны даже начинающим разработчикам. На меня это произвело глубокое впечатление.

—Г. Блейк Мейк, разработчик Android-приложений и соавтор книги *Programming Android 2e*

Дон и Дэвид обновили свои примеры и перевели их с Java на Kotlin, благодаря чему 3-е издание стало превосходным источником информации для разработчиков, желающих изучить современные средства разработки приложений Android.

—Дункан Макгрегор и Нэт Прайс, авторы книги *Java to Kotlin: A Refactoring Guidebook*

Спасибо вам за книгу. Я только начинаю осваивать разработку Android-приложений, и именно благодаря этой книге для меня все стало предельно понятно.

—Эмбрин Хан, разработчик приложений Android

Разработка Android-приложений сильно изменилась. Очень сильно! Пусть эта книга станет вашим дружелюбным, пунктуальным и умелым наставником. Она научит вас плыть в правильном направлении, прилагая силы именно там, где это необходимо. Держитесь подальше от опасных вод, кишащих пираньями управления зарядом, акулами async и пиявками быстродействия, — и вы получите массу удовольствия!

—Инго Кротцки, начинающий разработчик Android-приложений

Наконец-то! Теперь можно изучать Kotlin, не зная Java. Простая, лаконичная и увлекательная — книга, которую я так долго ждал.

— Доктор Мэтт Уэнем, специалист по анализу и обработке данных, программист Python

Kotlin — модный, подающий большие надежды язык, неуклонно набирающий темп в технологической отрасли. «Head First. Программирование для Android на Kotlin» — увлекательная, доступная книга, которая научит вас программировать на языке Kotlin с нуля. Книга ориентируется на читателей, которые учатся программировать, в ней приводятся снимки экранов среды IntelliJ IDEA, код снабжен примечаниями, а материал содержит множество иллюстраций.

— Аманда Хинчман-Домингес, инженер Android, дипломированный специалист по Groupon и Kotlin, Google

Head First

Android Development

Dawn Griffiths &
David Griffiths

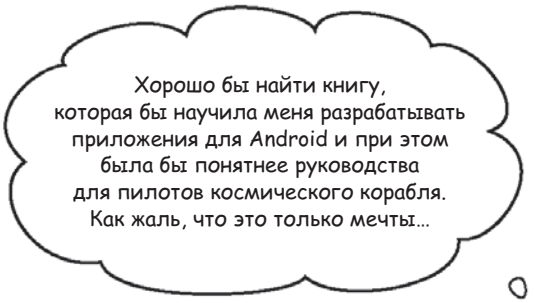
3rd Edition

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First

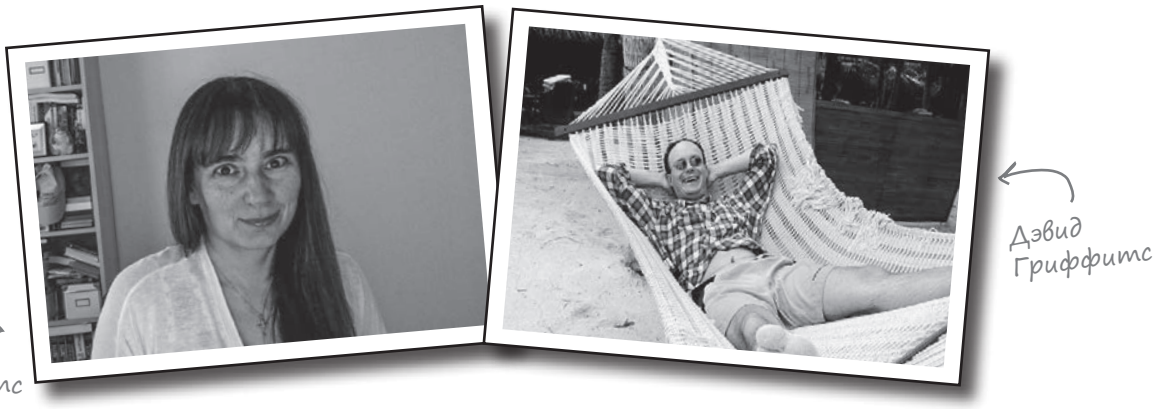
Программирование для Android на Kotlin



Дон Гриффитс
Дэвид Гриффитс

3-е издание

Об авторах



Дон Гриффитс начала с изучения математики в одном из ведущих университетов Великобритании, где получила диплом с отличием. Затем она продолжила карьеру в области разработки программного обеспечения; ее опыт работы в IT-отрасли составляет более 20 лет.

Когда Дон не работает над книгами серии Head First, она обычно совершенствует свое мастерство тайцзи, увлекается чтением, бегом, плетением кружев и кулинарией. Но больше всего ей нравится проводить время со своим замечательным мужем Дэвидом.

Дэвид Гриффитс увлекся программированием в 12 лет после документального фильма о работе Сеймура Пейперта. В 15 лет он написал реализацию языка программирования LOGO, созданного Пейпертом. После этого он работал преподавателем гибких методологий разработки, разработчиком и дежурным по гаражу (хотя и в другом порядке).

Когда он не занят программированием, литературной работой или преподаванием, он проводит свободное время в путешествиях со своей очаровательной женой Дон.

Дон и Дэвид совместно написали немало книг, в числе которых первое и второе издания *Head First. Программирование для Android*, *Head First Kotlin**, *Head First C*, *Head First Rails*, *Head First Statistics* и *React Cookbook*. Также они внесли свой творческий вклад в работу над книгой *97 Things Every Java Programmer Should Know* и разработали видеокурс *The Agile Sketchpad* для представления ключевых концепций и приемов с вовлечением и активным участием студентов. Наконец, они преподают на учебной платформе O'Reilly по адресу *https://www.oreilly.com/live-events*.

Их микроблоги в Twitter доступны по адресу *https://twitter.com/HeadFirstDroid*.

Содержание (сводка)

Содержание (настоящее)

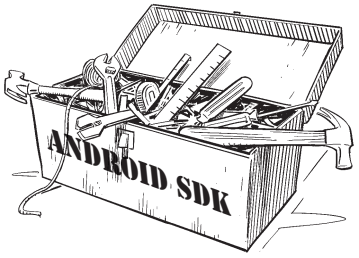
Введение

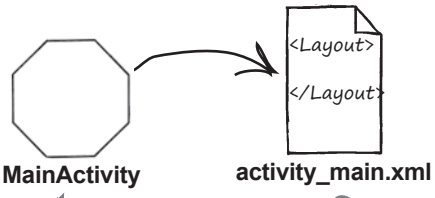
Ваш мозг и Android. Вы пытаетесь изучить что-то новое, а ваш мозг хочет оказать вам услугу и как можно быстрее забыть выученное. Он думает: «Лучше оставить место для чего-то поважнее, например, от каких диких животных стоит держаться подальше или почему на сноуборде не стоит кататься нагишом». Так как же заставить ваш мозг думать, что ваша жизнь зависит от умения разрабатывать приложения для Android?

1 Первые шаги

С головой в пучину

Android — самая популярная мобильная операционная система в мире. А по всему миру живут миллиарды пользователей Android, и все они мечтают загрузить вашу следующую замечательную разработку. В этой главе вы узнаете, как воплотить ваши идеи в жизнь для самой популярной мобильной операционной системы в мире, как **построить базовое приложение Android** и как обновить его. Также вы узнаете, как запустить его на физических и виртуальных устройствах. А попутно будут рассмотрены основные компоненты всех приложений Android: **активности** и **макеты**. Итак, за дело!





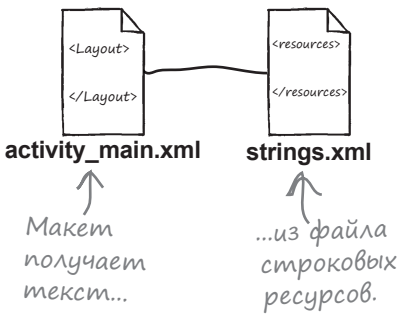
Активность указывает, что делает приложение и как оно должно взаимодействовать с пользователем.

Макет указывает, как выглядит приложение.

2 Построение интерактивных приложений

Приложения, которые что-то делают

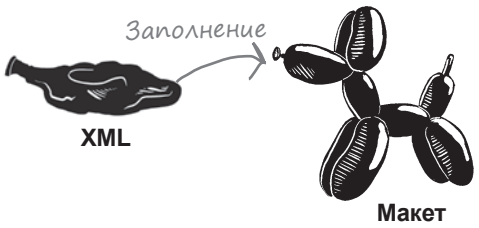
Обычно приложение должно реагировать на действия пользователя. В этой главе вы узнаете, как существенно повысить **интерактивность** ваших приложений. Вы узнаете, как добавить в код активности метод ***OnClickListener***, чтобы приложение могло **прослушивать** действия пользователя и соответствующим образом на них реагировать. Также вы научитесь **конструировать макеты** и поймете, как каждый UI-компонент, добавляемый в макет, происходит от общего **предка View**. Попутно вы узнаете, почему **строковые ресурсы** настолько важны для гибких, хорошо спроектированных приложений.



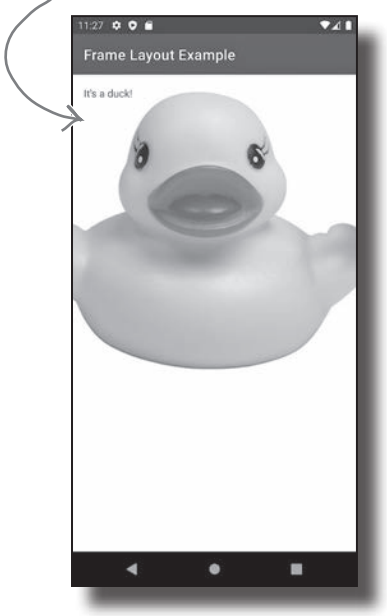
3 Макеты

Как работают макеты

Мы едва затронули тему использования макетов. К настоящему моменту вы видели, как разместить представления в простых линейных макетах, но это лишь малая часть того, на что способны макеты. В этой главе мы **рассмотрим вопрос глубже** и покажем, как на самом деле работают макеты. Вы узнаете, как **настроить линейные макеты**, и научитесь использовать **фреймовые макеты** и **представления с прокруткой**. А к концу главы вы узнаете, что, несмотря на внешние различия, все макеты — и добавленные к ним представления — имеют **больше общего**, чем кажется на первый взгляд.



Мы воспользуемся фреймовым макетом для наложения текстового поля на изображение утки.



4

Макеты с ограничениями

Построение эскиза

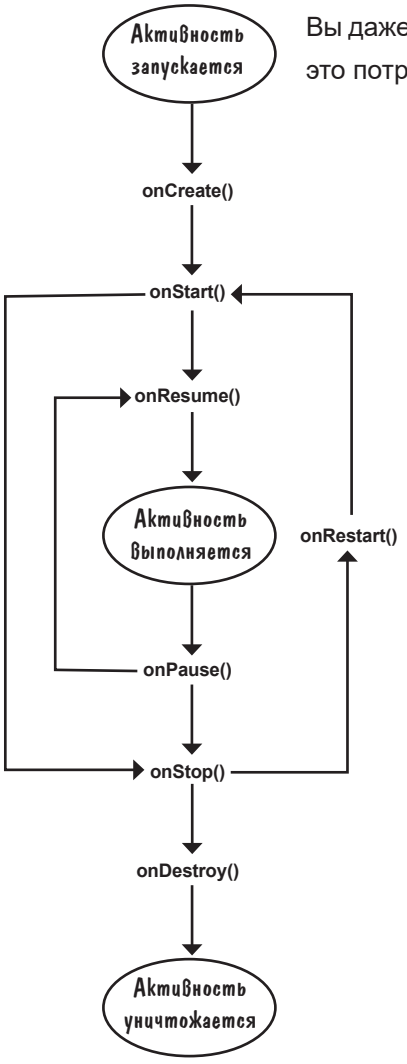
Дом невозможно построить без плана. А некоторые макеты строятся по **эскизам,** чтобы они выглядели **в точности так, как вам нужно**. В этой главе представлены **ограничения в макетах** Android: **гибкий механизм проектирования более сложных графических интерфейсов.** Вы узнаете, как **ограничения** и **смещения** используются для позиционирования и определения размеров ваших представлений **независимо от размера и ориентации экрана**. Вы узнаете, как закрепить представления в положенных местах при помощи **направляющих** и **барьеров**. Наконец, вы научитесь группировать или распределять представления с использованием **цепочек** и **потоков**. За дело!

Эскиз макета с ограничениями. Он содержит всю информацию, необходимую макету с ограничениями для размещения представлений на экране.

5 Жизненный цикл активности

Из жизни активностей

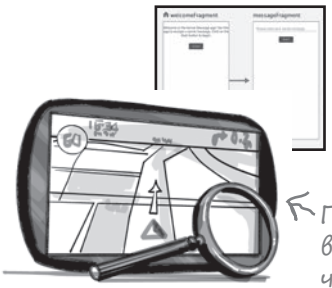
Активности образуют основу любого Android-приложения. Ранее вы видели, как создавать активности и как использовать активности для взаимодействия с пользователем. Но если вы еще не знакомы с **жизненным циклом активностей**, некоторые аспекты их поведения могут вас застать врасплох. В этой главе вы узнаете, что происходит при **создании** или **уничтожении** активностей и к каким **неожиданным последствиям** это может привести. Вы узнаете, как управлять их поведениями, когда они становятся **видимыми** или **скрываются**. Вы даже научитесь **сохранять** и **восстанавливать состояние активности**, когда это потребуется. Просто продолжайте читать дальше…



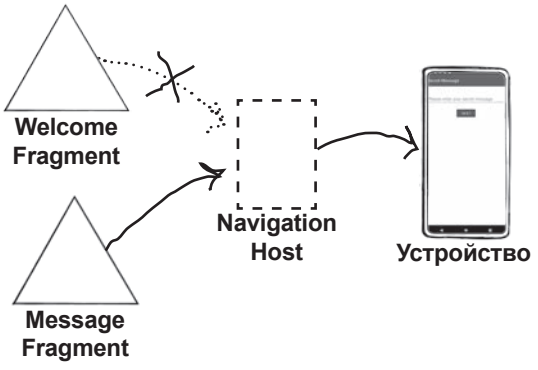
6 Фрагменты и навигация

Найди свой путь

Для большинства приложений одного экрана недостаточно. До настоящего момента мы рассматривали приложения с одним экраном; для простых приложений этого хватает. А если в вашем приложении **более сложные требования**? В этой главе вы научитесь использовать **фрагменты** и компонент **Navigation** для построения **приложений с несколькими экранами.** Вы узнаете, что **фрагменты похожи на субактивности**, имеющие собственные методы. Вы научитесь **проектировать эффективные графы навигации**. В последней части главы представлен **навигационный хост** и **навигационный контроллер**; вы узнаете, как они используются для перемещения в приложениях.



Граф навигации — что-то вроде GPS-навигатора, который сообщает вам, как добраться до того или иного места.

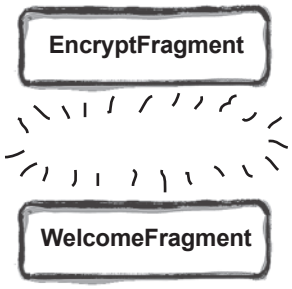


7

Плагин safe args

Передача информации

Иногда фрагментам для нормальной работы требуется дополнительная информация. Например, если во фрагменте выводятся контактные данные, этот фрагмент должен знать, какой контакт он должен выбрать. Но что, если эта информация **должна поступать из другого фрагмента**? В этой главе мы **воспользуемся вашими знаниями о навигации** и применим их **для передачи данных между фрагментами**. Вы узнаете, **как добавить аргументы** к целям навигации, чтобы они могли получать необходимую информацию. Вы познакомитесь с плагином **Safe Args** и научитесь использовать его **для написания кода, безопасного по отношению к типам**. Наконец, вы научитесь **работать со стеком возврата** и управлять поведением кнопки Back. Читайте дальше — отступать уже поздно.



Удаление MessageFragment из стека возврата означает, что при щелчке на кнопке Back из EncryptFragment вместо MessageFragment отображается WelcomeFragment.

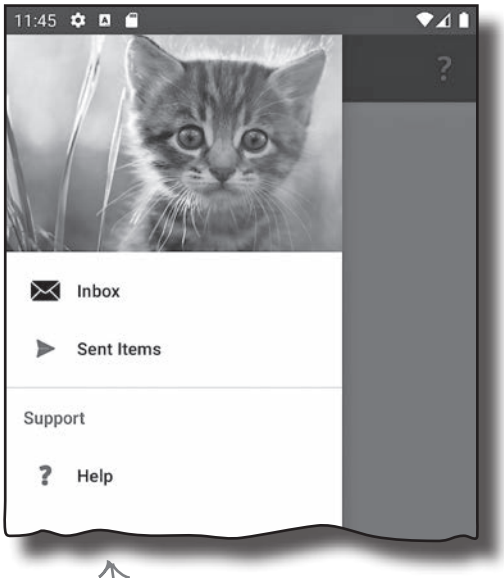
8

Интерфейс навигации

Туда и обратно

Во многих приложениях возникает необходимость в переходе между разными целями. Компонент Navigation сильно упрощает построение таких пользовательских интерфейсов. В этой главе вы научитесь пользоваться некоторыми навигационными UI-компонентами для Android, **упрощающими перемещение пользователя по приложению**. Вы узнаете, как пользоваться **темами** и как заменить панель приложения **панелью инструментов**. Вы научитесь добавлять **элементы меню**, которые могут использоваться для **навигации**, и освоите **реализацию навигации с нижней панели**. Наконец, мы создадим эффектную выдвижную панель, которая появляется на экране сбоку от вашей активности.

Контроллер навигации

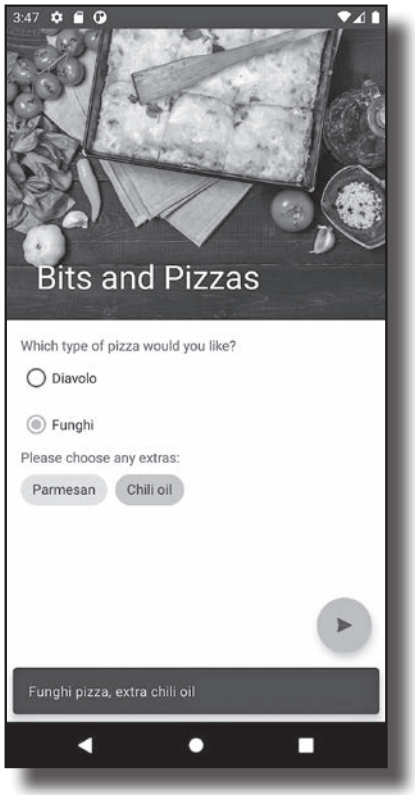


Если не связать выдвижную панель с контроллером навигации, элементы будут отображаться на выдвижной панели, но по щелчку на них ничего не произойдет.

9 Представления material

Материальный мир

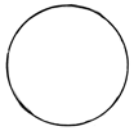
Многим приложениям нужен эффектный пользовательский интерфейс, быстро реагирующий на действия пользователя. В предыдущих главах вы научились пользоваться разными представлениями: **текстовыми представлениями, кнопками и полями выбора**, а также применять **темы Material** для внесения масштабных изменений в оформление и поведение приложения. Впрочем, это далеко не все. В этой главе вы узнаете, как улучшить скорость реакции вашего приложения применением **координирующего макета**. Вы создадите **панели инструментов** с возможностью сворачивания или прокрутки содержимого по вашему усмотрению. Вы познакомитесь с **интересными новыми представлениями: флажками**, **переключателями**, **плашками** и **плавающими кнопками действий**. Наконец, вы узнаете, как отображать удобные всплывающие сообщения с использованием панелей **Toast** и **Snackbar**. Переверните страницу, чтобы узнать больше.

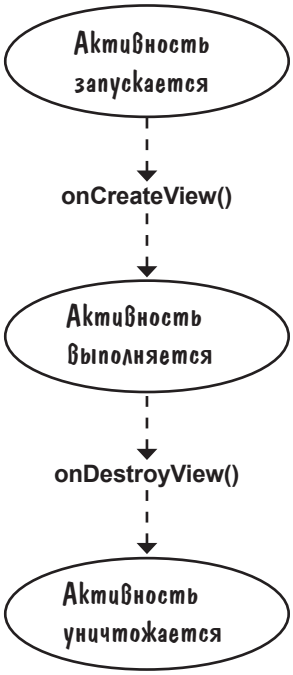


10 Связывание представлений

Связанные одной целью

Пришло время попрощаться с findViewById(). Как вы, вероятно, уже заметили, чем больше у вас представлений и чем более интерактивным становится ваше приложение, тем больше вызовов ***findViewById()*** приходится использовать. И если вам надоело вызывать этот метод каждый раз, когда вам потребуется поработать с представлением, знайте: *вы не одиноки*. В этой главе вы узнаете, **как связывание представлений делает *findViewById()* пережитком прошлого**. Вы научитесь применять этот механизм в коде **активностей и фрагментов** и узнаете, почему этот подход предоставляет более безопасный и эффективный способ обращения к представлениям вашего макета. Итак, за дело!

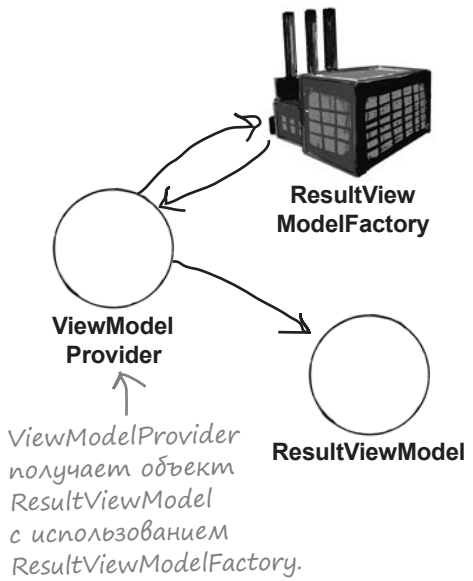




11 Модели представлений

Поведение модели

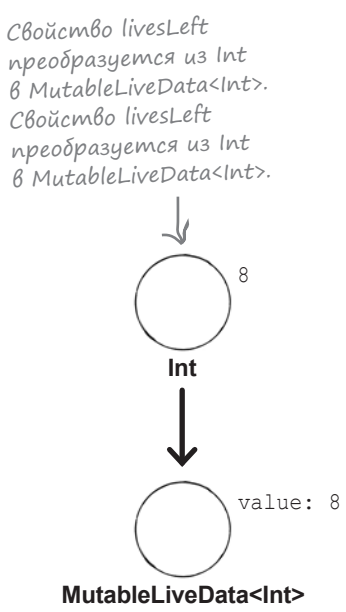
С ростом сложности приложений фрагментам приходится жонглировать все большим количеством объектов. И если не проявить осторожность, это может привести к **разбуханию** кода, который пытается делать все сразу. Бизнес-логика, навигация, управление пользовательским интерфейсом, обработка изменений в конфигурации... что ни возьми, оно здесь. В этой главе вы узнаете, как действовать в подобных ситуациях с использованием **моделей представлений.** Вы узнаете, **как они упрощают код активности и фрагментов** и как они **справляются с изменениями конфигурации**, поддерживая целостное состояние приложения. Наконец, мы покажем, как построить **фабрику моделей представлений** и когда может возникнуть такая необходимость.



12 Живые данные

В самой гуще событий

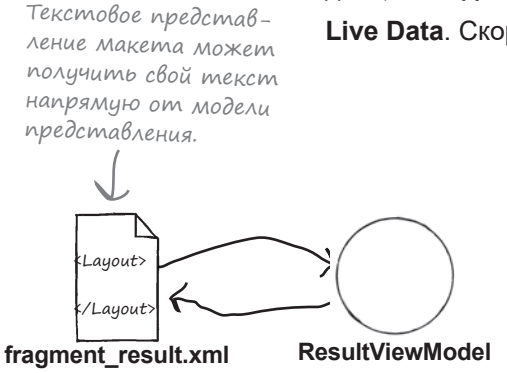
Вашему коду часто приходится реагировать на изменения значений свойств. Например, если свойство модели представления меняет значение, **фрагмент может отреагировать** обновлением своих представлений или переходом. Но **как фрагмент узнает о том, что свойство было обновлено**? В этой главе вы узнаете о **Live Data**: механизме **оповещения заинтересованных сторон о каких-либо изменениях**. Вы узнаете о ***MutableLiveData*** и о том, как **заставить ваш фрагмент наблюдать за изменениями свойств этого типа. Мы покажем, как тип *LiveData*** помогает обеспечить целостность данных вашего приложения. Вскоре вы будете писать приложения, которые реагируют на происходящее быстрее, чем когда-либо.



13 Связывание данных

Построение умных макетов

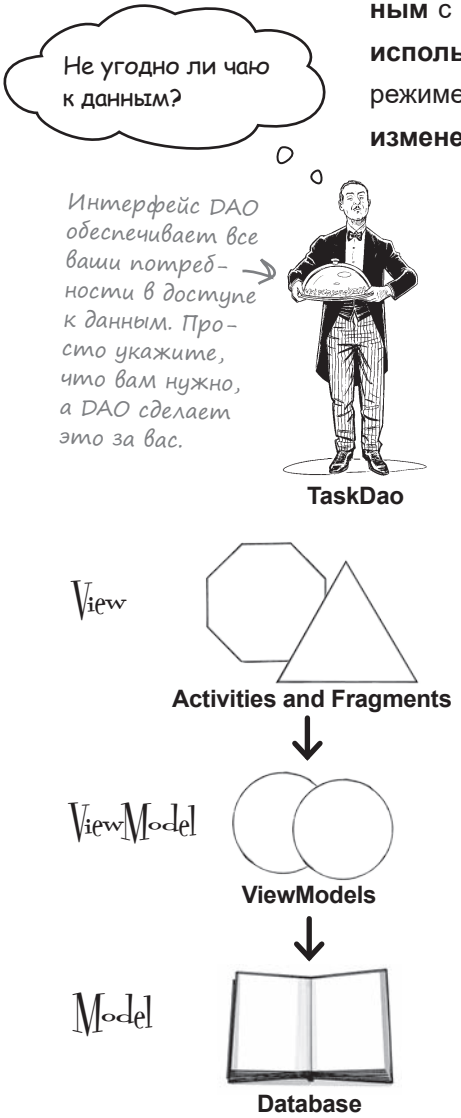
Возможности макетов не ограничиваются управлением внешним видом вашего приложения. У всех макетов, написанных нами ранее, поведение должно было определяться кодом активности или фрагмента. Но только представьте, что макеты могли бы думать самостоятельно и принимать собственные решения. В этой главе представлен механизм **связывания данных**: способ наращивания интеллекта ваших макетов. Вы узнаете, **как заставить представления получать данные** непосредственно от модели представления. Мы воспользуемся **связыванием слушателей**, чтобы кнопки вызывали свои методы. Вы даже увидите, как **одна простая строка кода позволяет реагировать на обновления Live Data**. Скоро ваши макеты станут более мощными, чем когда-либо.



14 Базы данных Room

Номер с видом

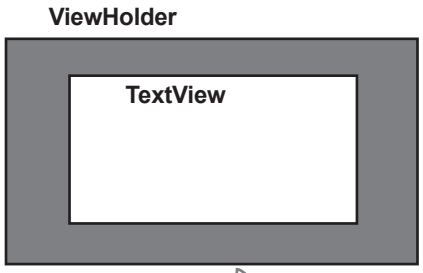
В большинстве приложений используются данные. Но если вы не позаботитесь о том, чтобы эти данные были где-то сохранены, **данные будут навсегда потеряны** при закрытии приложения. В мире приложений Android информация обычно ***хранится в базах данных***, поэтому в этой главе мы представим **библиотеку хранения данных Room**. Вы научитесь **строить базы данных**, **создавать таблицы** и **определять методы доступа к данным** с использованием аннотаций классов и интерфейсов. Вы узнаете, как **использовать сопрограммы** для выполнения кода баз данных в фоновом режиме. Заодно вы научитесь **преобразовывать данные Live Data при их изменении с помощью *Transformations.map()***.



15 Представления с переработкой

Экономия и переработка

Списки данных являются важнейшей частью многих приложений. В этой главе мы покажем, как создать такой список с использованием **представлений с переработкой**: **невероятно гибкого** способа построения **прокручиваемых списков.** Вы научитесь создавать **гибкие макеты** для ваших списков, включая текстовые представления, флажки и многое другое. Вы узнаете, **как создавать адаптеры**, которые **отображают ваши данные** в представлениях с переработкой так, как вам нужно. Вы научитесь использовать **карточные представления** для оформления данных **с имитацией объема**. Наконец, мы покажем, как **менеджеры макетов могут полностью изменить внешний вид вашего списка всего одной или двумя строками кода**. Давайте займемся переработкой.



Каждый держатель представления содержит корневое представление макета для каждого элемента. В данном случае корневым является текстовое представление.

16 DiffUtil и связывание данных

На полных оборотах

Приложение должно быть настолько плавным и быстрым, насколько это возможно. Но если действовать неосторожно, большие и сложные наборы данных могут привести к сбоям в вашем представлении с переработкой. В этой главе мы представим ***DiffUtil***: вспомогательный класс, который **расширяет возможности представлений с переработкой**. Вы научитесь использовать его для **эффективного обновления** представлений с переработкой. Вы узнаете, как объекты ***ListAdapter*** упрощают работу с ***DiffUtil***. А заодно будет показано, **как полностью избавиться от *findViewById()*** реализацией **связывания данных в коде представления с переработкой.**

17 Навигация в представлениях с переработкой

Карточные фокусы

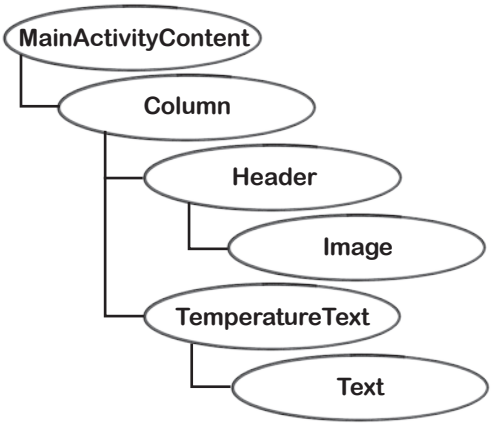
Некоторые приложения требуют, чтобы пользователь выбрал элемент из списка. В этой главе вы узнаете, **как сделать представления с переработкой центральной частью структуры вашего приложения**, для чего следует организовать **обработку щелчков на их элементах**. Вы увидите, как **реализовать навигацию в представлениях с переработкой**, чтобы приложение переходило к новому экрану каждый раз, когда пользователь щелкает на записи. Мы покажем, **как вывести дополнительную информацию** о выбранной записи и обновить ее в базе данных. К концу этой главы у вас будут все средства, необходимые для того, чтобы **преобразовать ваши великолепные замыслы в приложение вашей мечты**.

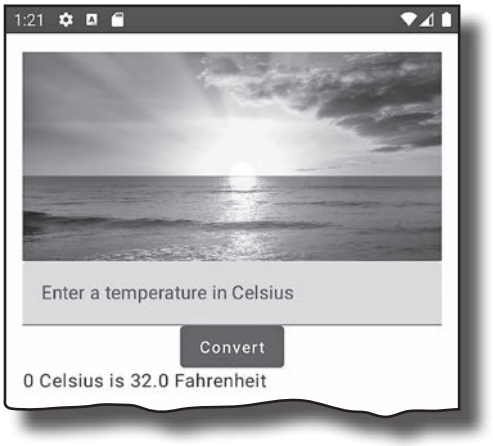
18 Jetpack compose

В мире Compose

Во всех пользовательских интерфейсах, которые мы строили ранее, использовались представления и файлы макетов. Но благодаря инструментарию **Jetpack Compose это не единственный вариант**. В этой главе перед вами развернется **мир Compose**. Вы научитесь строить пользовательские интерфейсы из компонентов Compose вместо представлений. Вы узнаете, как пользоваться такими компонентами Compose, как **Text**, **Image**, **TextField** и **Button**, размещать их по строкам и столбцам и применять стилевое оформление при помощи **тем**. Вы напишете собственные **компонентные функции** и даже научитесь управлять **состоянием** компонентов Compose с использованием объектов ***MutableState***. Переверните страницу.

Compose строит дерево компонентов, которое выглядит так:

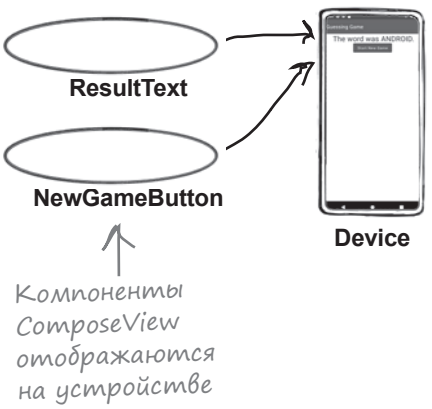




19 Интеграция с представлениями

Полная гармония

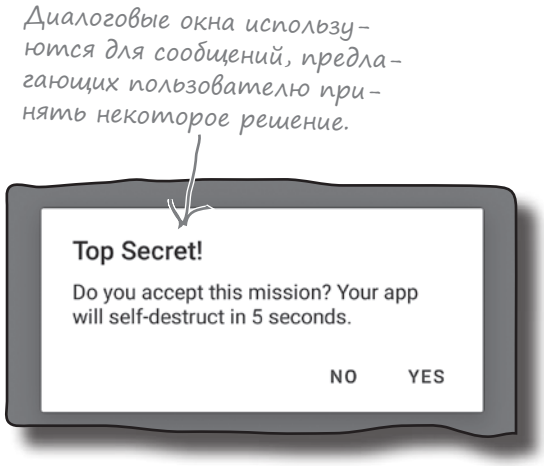
Лучшие результаты достигаются при совместной работе. К настоящему моменту вы научились строить пользовательские интерфейсы из представлений и компонентов Compose. А если вы хотите использовать их **одновременно**? В этой главе вы узнаете, как пользоваться преимуществами обеих технологий, добавляя компоненты Compose в пользовательские интерфейсы на базе представлений. Вы освоите средства, позволяющие компонентам **работать с моделями представлений**. Вы даже узнаете, как заставить их **реагировать на *обновления Live Data***. К концу главы у вас появится все необходимое для **использования компонентов Compose с представлениями** и даже для **перехода на пользовательские интерфейсы**, построенные **исключительно на базе Compose.**

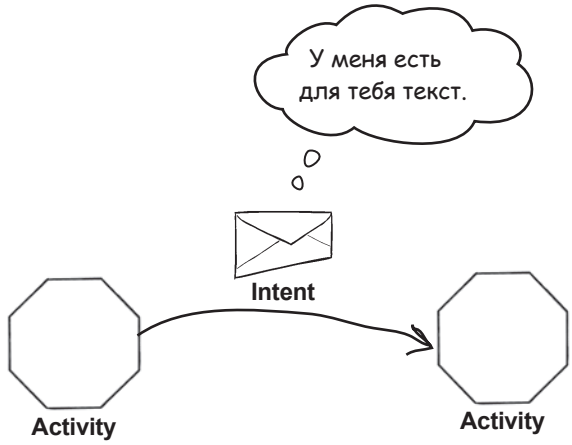


Приложение. Остатки

Десять важных тем, которые мы не рассмотрели

Даже после всего сказанного еще кое-что осталось. Нам хотелось бы рассмотреть еще ряд вопросов. Было бы неправильно делать вид, что их не существует; с другой стороны, нам не хотелось, чтобы нашу книгу можно было поднять только после интенсивной тренировки в спортзале. Прежде чем ставить книгу на полку, **ознакомьтесь с приложением**.





Как работать с этой книгой

Введение

В этом разделе мы ответим на важный вопрос: «Почему они включили ТАКОЕ в книгу по программированию для Android?»

Для кого написана эта книга?

Если вы ответите «да» на все следующие вопросы:

1. Вы уже умеете программировать на Kotlin, Java или другом объектно-ориентированном языке?
2. Вы хотите достичь мастерства в области разработки приложения для Android, создать следующий бестселлер в области программных продуктов, заработать целое состояние и купить собственный остров?
3. Вы предпочитаете заниматься практической работой и применять полученные знания, вместо того чтобы выслушивать нудные многочасовые лекции?

Ладно, здесь мы слегка хватили через край. Но ведь нужно с чего-то начинать, верно?

тогда эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» на один из следующих вопросов:

1. Вам нужен краткий вводный курс или справочник по разработке приложений для Android?
2. Вы скорее пойдете к зубному врачу, чем опробуете что-нибудь новое? Вы считаете, что в книге по Android должно быть рассказано абсолютно обо *всем* (особенно о том, что вы никогда не будете использовать), а если читатель будет помирать со скуки — еще лучше?

...эта книга *не* для вас.

[Замечание от отдела продаж: вообще-то эта книга для любого, у кого есть деньги... и если что — система быстрых платежей тоже подойдет.]

Мы знаем, о чем вы думаете

«Разве серьезные книги по программированию для Android *такие?*»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь *научиться?*»

И мы знаем, о чем думает ваш *мозг*

Мозг жаждет новых впечатлений. Он постоянно ищет, анализирует, *ожидает* чего-то необычного. Он так устроен, и это помогает нам выжить.

Как наш мозг поступает со всеми обычными, повседневными вещами? Он всеми силами пытается оградиться от них, чтобы они не мешали его *настоящей* работе — сохранению того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит фильтр, отсекающий «очевидно несущественное».

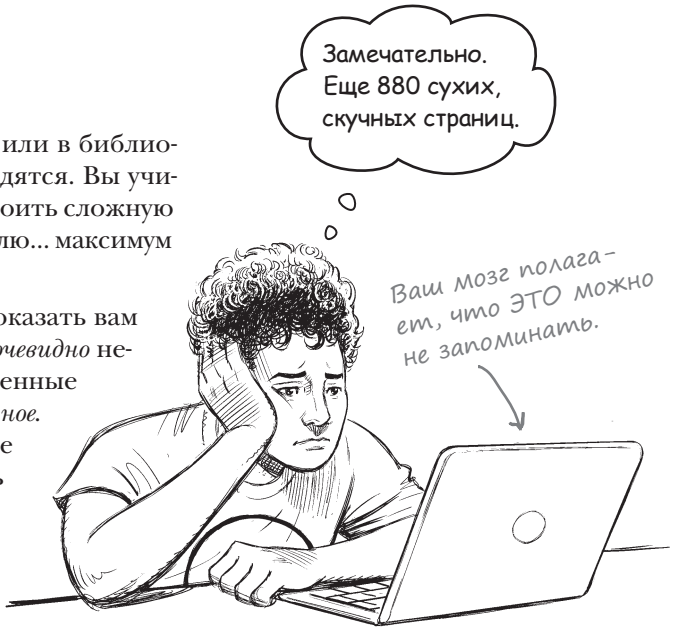
Но как же мозг *узнает*, что важно? Представьте, что вы выехали на прогулку и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и в теле?

Активизируются нейроны. Вспыхивают эмоции. *Происходят химические реакции.*

И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библиотеке — в теплом, уютном месте, где тигры не водятся. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему, на которую вам выделили неделю... максимум десять дней.



И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь *важное*. На тигров, например. Или на то, что к огню лучше не прикасаться. Или что не стоит выкладывать фотки с вечеринок в интернете. Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Эта книга для тех, кто хочет учиться.

Как мы что-то *узнаем*? Сначала нужно это «что-то» понять, а потом *не забыть*. Затолкать в голову побольше фактов недостаточно. Согласно новейшим исследованиям в области когнитивистики, нейробиологии и психологии обучения, для *усвоения* материала требуется что-то большее, чем простой текст на странице. Мы знаем, как заставить ваш мозг работать.

Основные принципы серии «Head First»:

Наглядность. Графика запоминается лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89 %, по данным исследований). Кроме того, материал становится более понятным. Текст размещается на рисунках, к которым он относится, а не под ними или на соседней странице — и вероятность успешного решения задач, относящихся к материалу, повышается *вдвое*.

Разговорный стиль изложения. Недавние исследования показали, что при разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании достигает 40 %. Рассказывайте историю, вместо того чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что привлечет ваше внимание: занимательная беседа за столом или лекция?

Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых задействованы оба полушария мозга и разные чувства.

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой странице». Мозг обращает внимание на интересное, странное, притягательное, неожиданное. Изучение сложной технической темы не обязано быть скучным. Интересное узнается намного быстрее.

Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. Мы запоминаем, когда что-то *чувствуем*. Нет, сантименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес и чувство «Да я крут!» при решении задачи, которую окружающие считают сложной, — или когда вы понимаете, что разбираетесь в теме *лучше*, чем всезнайка Боб из технического отдела.

Метапознание: наука о мышлении

Если вы действительно хотите быстрее и глубже усваивать новые знания, задумайтесь над тем, как вы думаете. Учитесь учиться.

Мало кто из нас изучает теорию метапознания во время учебы. Нам *положено* учиться, но нас редко этому *учат*.

Но раз вы читаете эту книгу, то вероятно, хотите освоить программирование для Android, и по возможности быстрее. Вы хотите *запомнить* прочитанное, а для этого абсолютно необходимо сначала *понять* прочитанное.

Чтобы извлечь максимум пользы из учебного процесса, нужно заставить ваш мозг воспринимать новый материал как Нечто Важное. Критичное для вашего существования. Такое же важное, как тигр. Иначе вам предстоит бесконечная борьба с вашим мозгом, который всеми силами уклоняется от запоминания новой информации.

Как же УБЕДИТЬ мозг, что программирование не менее важно, чем голодный тигр?

Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию *можно* запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и то же повторяется *столько* раз... Ладно, уговорил».

Быстрый способ основан на *повышении активности мозга* и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном тексте и т. д.) заставляет мозг анализировать связи между текстом и графикой, а это приводит к активизации большего количества нейронов. Больше нейронов — выше вероятность того, что информация будет *сочтена* важной и достойной запоминания.

Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно *не интересует*, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует то же, что чувствуете вы на скучной лекции в роли пассивного участника. Его клонит в сон.

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ

Мы использовали ***рисунки***, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок сто́ит тысячи слов. А когда текст комбинируется с графикой, мы внедряем текст *прямо в рисунки*, потому что мозг при этом работает эффективнее.

Мы используем ***избыточность***: повторяем одно и то же несколько раз, применяя *разные* средства передачи информации, обращаемся к *разным чувствам* — и все для повышения вероятности того, что материал будет закодирован в нескольких областях вашего мозга.

Мы используем концепции и рисунки несколько ***неожиданным*** образом, потому что мозг лучше воспринимает новую информацию. Кроме того, рисунки и идеи обычно имеют ***эмоциональное*** *содержание*, потому что мозг обращает внимание на биохимию эмоций. То, что заставляет нас *чувствовать*, лучше запоминается, будь то ***шутка***, ***удивление*** или ***интерес***.

Мы используем ***разговорный стиль***, потому что мозг лучше воспринимает информацию, когда вы участвуете в разговоре, а не пассивно слушаете лекцию. Это происходит и при *чтении*.

В книгу включены многочисленные ***упражнения***, потому что мозг лучше запоминает, когда вы что-то *делаете*. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

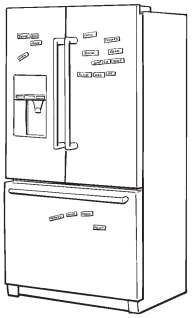
Мы совместили ***несколько стилей обучения***, потому что одни читатели предпочитают пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений полезно видеть несколько вариантов представления одного материала.

Мы постарались задействовать ***оба полушария вашего мозга***; это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

А еще в книгу включены ***истории*** и упражнения, отражающие другие точки зрения. Мозг глубже усваивает информацию, когда ему приходится оценивать и выносить суждения.

В книге часто встречаются ***вопросы***, на которые не всегда можно дать простой ответ, потому что мозг быстрее учится и запоминает, когда ему приходится что-то делать. Невозможно накачать ***мышцы***, наблюдая за тем, как занимаются ***другие***. Однако мы позаботились о том, чтобы усилия читателей были приложены в ***верном*** направлении. Вам не придется ломать голову над невразумительными примерами или разбираться в сложном, перенасыщенном техническим жаргоном или слишком лаконичном тексте.

В историях, примерах, на картинках используются образы ***людей,*** потому что вы тоже ***человек*** и ваш мозг обращает на людей больше внимания, чем на неодушевленные ***предметы***.



Вырежьте и прикрепите на холодильник.

Что можете сделать ВЫ, чтобы заставить ваш мозг повиноваться

Мы свое дело сделали. Остальное за вами. Эти советы станут отправной точкой; прислушайтесь к своему мозгу и определите, что вам подходит, а что не подходит. Пробуйте новое.

1 Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.

Просто читать недостаточно. Когда книга задает вам вопрос, не переходите к ответу. Представьте, что кто-то *действительно* задает вам вопрос. Чем глубже ваш мозг будет мыслить, тем скорее вы поймете и запомните материал.

2 Выполняйте упражнения, делайте заметки.

Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте* упражнения. **Берите карандаш и пишите**. Физические действия *во время* обучения повышают его эффективность.

3 Читайте врезки.

Это значит: читайте все. ***Врезки — часть основного материала***! Не пропускайте их.

4 Не читайте другие книги после этой перед сном.

Часть обучения (особенно перенос информации в долгосрочную память) происходит после того, как вы откладываете книгу. Ваш мозг не сразу усваивает информацию. Если во время обработки поступит новая информация, часть того, что вы узнали ранее, может быть потеряна.

5 Говорите вслух.

Речь активизирует другие участки мозга. Если вы пытаетесь что-то понять или получше запомнить, произнесите вслух. А еще лучше — попробуйте объяснить кому-нибудь другому. Вы будете быстрее усваивать материал и, возможно, откроете для себя что-то новое.

6 Пейте воду. И побольше.

Мозгу нужна влага, так он лучше работает. Дегидратация (которая может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.

7 Прислушивайтесь к своему мозгу.

Следите за тем, когда ваш мозг начинает уставать. Если вы поверхностно воспринимаете материал или забываете только что прочитанное, пора сделать перерыв.

8 Чувствуйте!

Ваш мозг должен знать, что материал книги действительно *важен*. Переживайте за героев наших историй. Придумывайте собственные подписи к фотографиям. Поморщиться над неудачной шуткой все равно лучше, чем не почувствовать ничего.

9 Пишите программы!

Освоить разработку Android-приложений можно только одним способом: **писать побольше кода**. Именно этим мы и будем заниматься в книге. Программирование — искусство, и добиться мастерства в нем можно только практикой. В каждой главе приведены упражнения, в которых вам придется решать задачи. Не пропускайте их — работа над упражнениями является важной частью процесса обучения. К каждому упражнению приводится решение — не бойтесь **заглянуть в него**, если окажетесь в тупике! (Иногда можно застрять на элементарном.) Но все равно пытайтесь решать задачи самостоятельно. Пока ваш код не начнет работать, не стоит переходить к следующим страницам книги.

Примите к сведению

Это учебник, а не справочник. Мы намеренно убрали из книги все, что могло бы помешать изучению материала, над которым вы работаете. И при первом чтении книги начинать следует с самого начала, потому что книга предполагает наличие у читателя определенных знаний и опыта.

Предполагается, что у вас нет опыта программирования для Android, но вы в какой-то степени владеете Kotlin.

Мы будем строить приложения Android с использованием Kotlin и XML. Предполагается, что вы уже знакомы с языком программирования Kotlin или другим объектно-ориентированным языком, таким как Java. Если вы еще *никогда* не писали программы на Kotlin, прочитайте Head First Kotlin, прежде чем браться за эту книгу.

Мы начинаем строить приложения с первой главы.

Хотите верьте, хотите нет, но даже если вы никогда не программировали для Android, вы все равно можете с ходу взяться за создание приложений. Заодно вы познакомитесь с Android Studio, основной интегрированной средой разработки для Android.

Примеры создавались для обучения.

Во время работы над книгой мы построим несколько разных приложений. Некоторые из них очень малы, чтобы вы могли сосредоточиться на конкретных аспектах Android. Другие, более крупные приложения показывают, как разные компоненты работают в сочетании друг с другом. Мы не будем доводить до конца все части всех приложений, но ничто не мешает вам экспериментировать с ними самостоятельно — это часть учебного процесса. Исходный код всех приложений доступен по адресу *https://tinyurl.com/hfad3*.

Упражнения ОБЯЗАТЕЛЬНЫ.

Упражнения являются частью основного материала книги. Одни упражнения способствуют запоминанию материала, другие помогают лучше понять его, третьи ориентированы на его практическое применение.

Не пропускайте упражнения.

Повторение применяется намеренно.

У книг этой серии есть одна принципиальная особенность: мы хотим, чтобы вы *действительно хорошо* усвоили материал. И чтобы вы запомнили все, что узнали. Большинство справочников не ставит своей целью успешное запоминание, но это не справочник, а *учебник*, поэтому некоторые концепции излагаются в книге по нескольку раз.

Упражнения «Мозговой штурм» не имеют ответов.

В некоторых из них правильного ответа вообще нет, в других вы должны сами решить, насколько правильны ваши ответы (это является частью процесса обучения). В некоторых упражнениях «Мозговой штурм» приводятся подсказки, которые помогут вам найти нужное направление.

(наши потрясающие) Научные редакторы

Жаки Коуп

Инго Кротцки

Кен Каусен

Эш Теппин

Научные редакторы:

Жаки Коуп занялась программированием, чтобы избежать школьных тренировок по баскетболу. С тех пор она приобрела серьезный опыт работы со многими финансовыми и образовательными системами, от программирования на COBOL до управления тестированием. За прошедшее время она получила степень магистра в области компьютерной безопасности, а основной сферой ее деятельности стал контроль качества в сфере высшего образования. В свободное время Жаки любит готовить, гулять на природе и смотреть «Доктора Кто».

Инго Кротцки работал на различных должностях в отрасли здравоохранения, прежде всего для контрактных исследовательских организаций, занимающихся клиническими испытаниями, — он был системным архитектором, администратором и программистом базы данных, разработчиком и аналитиком данных. Свободное время он любит проводить среди диких (и не очень диких) животных в сельской местности; занимается парным программированием с белками, сойками и зябликами; изучает новейшие и самые модные мобильные фреймворки.

Кен Каусен — большой поклонник Java, энтузиаст Oracle и Grails. Он написал целый ряд книг, в числе которых *Help Your Boss Help You* (Pragmatic Library), *Kotlin Cookbook*, *Modern Java Recipes* и *Gradle Recipes for Android* (O'Reilly), а также *Making Java Groovy* (Manning). Он записал свыше десятка видеокурсов для учебной платформы O'Reilly по темам, связанным с Android, Spring, Java, Groovy, Grails и Gradle. Также является многократным обладателем премии JavaOne Rockstar. Его образовательный уровень включает степени бакалавра по инженерной механике и математике в MIT, степень магистра и доктора философии Принстонского университета и степень магистра по вычислительным технологиям в RPI. В настоящее время Кен возглавляет компанию Kousen IT, Inc. из Коннектикута.

Эш Теппин — разработчик-полиглот, работающий в основном над веб-приложениями. Он с большим энтузиазмом относится к построению новых приложений, которые упрощают жизнь людей. Время, свободное от программирования, Эш посвящает игре на гитаре, слушает музыку, занимается бегом, туризмом и садоводством.

Благодарности

Редактору

Мы безгранично благодарны нашему замечательному редактору **Вирджинии Уилсон**, взявшей на себя управление третьим изданием книги. Работать с Вирджинией было настоящим удовольствием, она щедро делилась с нами своими бесценными замечаниями и советами. Ее выдающееся мастерство организатора позволило нам не сбиться с пути в своей работе, и она приложила массу усилий, чтобы предоставить нам все необходимое именно тогда, когда это было нужно. Мы искренне благодарны Вирджинии за упорную работу и поддержку.

Вирджиния Уилсон

Зен Маккуэйд

Команде O'Reilly

Огромное спасибо **Зен Маккуэйд**, которая предложила нам написать третье издание книги, за поддержку; **Шире Эванс и Николь Таш,** делившимися своим мнением в ходе редактирования; **группе дизайнеров** за превосходное новое оформление и за помощь с заменой старых элементов; **Кэти Тозер** и **Кристен Браун** за распространение ранних версий книги. Наконец, мы благодарим **производственную группу, которая мастерски руководила процессом печати и приложила массу усилий, незаметных для внешнего наблюдателя.**

Семье, друзьям и коллегам

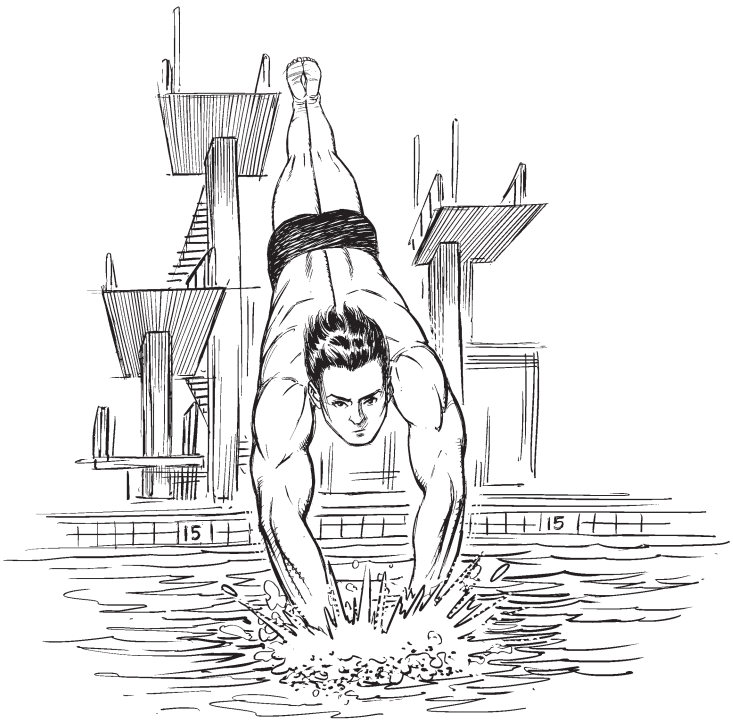
Написание книг из серии Head First никогда не шло гладко, и третье издание не стало исключением. Мы по-настоящему благодарны семье и друзьям за всю доброту и поддержку на этом пути. Хотим отдельно поблагодарить **маму**, **папу**, **Роба**, **Лоррейн**, **Марка**, **Лору**, **Энди**, **Айшу**, **Энди**, **Матти**, **Иэна**, **Ванессу**, **Дон**, **Уильяма** и **Саймона**.

Без кого эта книга никогда бы не появилась

Наша замечательная группа технических редакторов упорно работала над тем, чтобы предоставить обратную связь и следить за точностью приведенной в книге информации. Также мы благодарны всем, кто поделился своим мнением о ранних версиях книги и двух предыдущих изданиях. Как нам кажется, книга от этого стала намного, намного лучше.

И наконец, спасибо **Кэти Сьерра** и **Берту Бэйтсу**, которые создали эту замечательную серию книг, уговорили нас отложить старые учебники и поделились своим ви́дением.

С головой в пучину



Android — самая популярная мобильная операционная система в мире. А по всему миру живут миллиарды пользователей Android, и все они мечтают загрузить вашу следующую замечательную разработку. В этой главе вы узнаете, как воплотить ваши идеи в жизнь для самой популярной мобильной операционной системы в мире, **как построить базовое приложение Android** и как обновить его. Также вы узнаете, как запустить его на физических и виртуальных устройствах. А попутно будут рассмотрены основные компоненты всех приложений Android: **активности** и **макеты**. Итак, за дело!

Добро пожаловать в мир Android

Android — самая популярная мобильная платформа в мире. Согласно последним опросам, в мире свыше *трех миллиардов* активных Android-устройств, и их количество продолжает стремительно расти. Android — полнофункциональная платформа с открытым кодом на базе Linux, разрабатываемая компанией Google. Это мощная платформа разработки, включающая все необходимое для построения современных приложений. Более того, построенные приложения могут устанавливаться на множестве разных устройств — телефонах, планшетах и не только. Что же собой представляет типичное Android-приложение?

Для построения приложений Android мы будем использовать Kotlin и XML. Некоторые вещи будут объясняться по ходу дела, но чтобы извлечь пользу из этой книги, читатель должен понимать Kotlin.

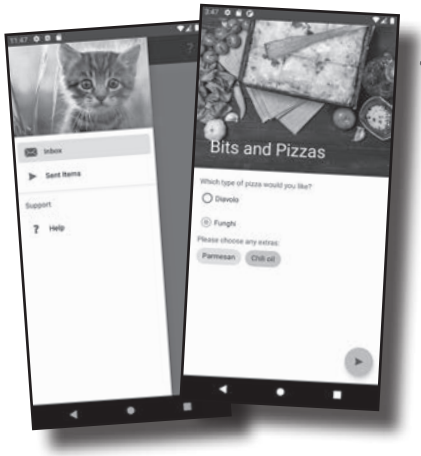
Активности определяют, что приложение делает

Каждое приложение Android включает одну или несколько **активностей**. Активность представляет собой специальный класс, обычно написанный на Kotlin, который управляет поведением приложения и решает, как реагировать на действия пользователя. Например, если в приложении присутствует кнопка, в активность добавляется код, который указывает, что должно происходить при нажатии этой кнопки.

Активности определяют, что приложение должно делать.

Макеты определяют, как будут выглядеть экраны приложения

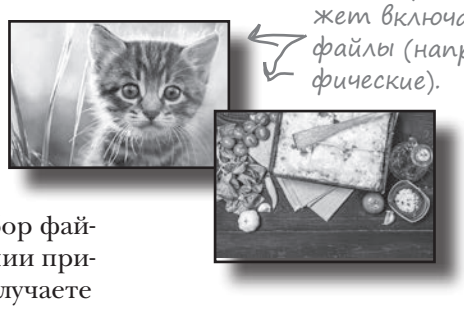
Типичное приложение Android состоит из одного или нескольких экранов. Внешний вид каждого экрана определяется файлом **макета** или дополнительным кодом активности. Макеты обычно определяются на языке XML, и каждый экран может включать такие компоненты, как кнопки, текст и графика.



Макеты сообщают Android, как будут выглядеть экраны вашего приложения.

Также могут понадобиться другие файлы

Кроме активностей и макетов, приложениям Android часто требуются дополнительные ресурсы: графические файлы, данные приложений и т. д. В приложение можно включить любые дополнительные файлы, которые вам понадобятся.

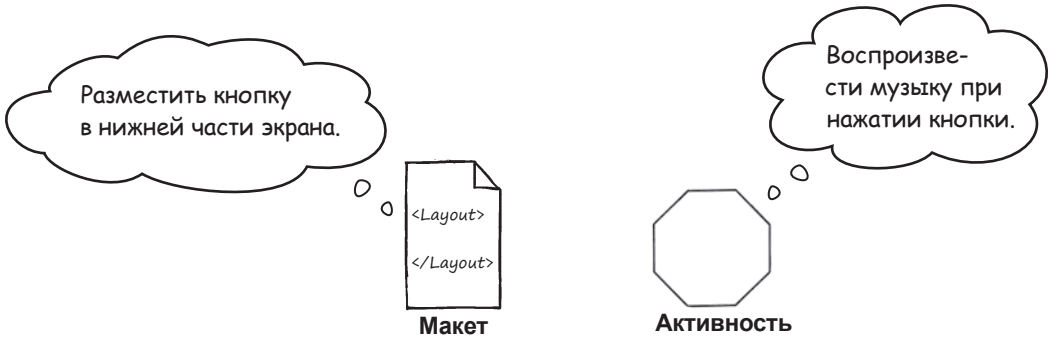


Также приложение может включать другие файлы (например, графические).

На самом деле приложение Android — всего лишь набор файлов в заранее определенных каталогах. При построении приложения все эти файлы собираются воедино, и вы получаете приложение, которое можно запустить на вашем устройстве.

Активности и макеты образуют костяк вашего приложения

В типичном приложении активности и макеты работают совместно для определения пользовательского интерфейса приложения. Макеты сообщают Android, как должны быть упорядочены различные элементы экрана, а активности управляют поведением приложения. Например, если в приложении присутствует кнопка, макет определяет ее позицию, а активность управляет тем, что происходит при нажатии этой кнопки пользователем.



А вот как активности и макеты работают совместно при запуске приложения на устройстве:

1. **Android запускает главную активность приложения.**
2. **Активность приказывает Android использовать конкретный макет.**
3. **Макет отображается на устройстве.**
4. **Пользователь взаимодействует с макетом.**
5. **Активность реагирует на эти взаимодействия и обновляет изображение...**
6. **...которое пользователь видит на устройстве.**

Теперь вы в общих чертах представляете, как строятся приложения Android, сделаем следующий шаг и построим простейшее приложение Android.

Вот что мы сейчас сделаем

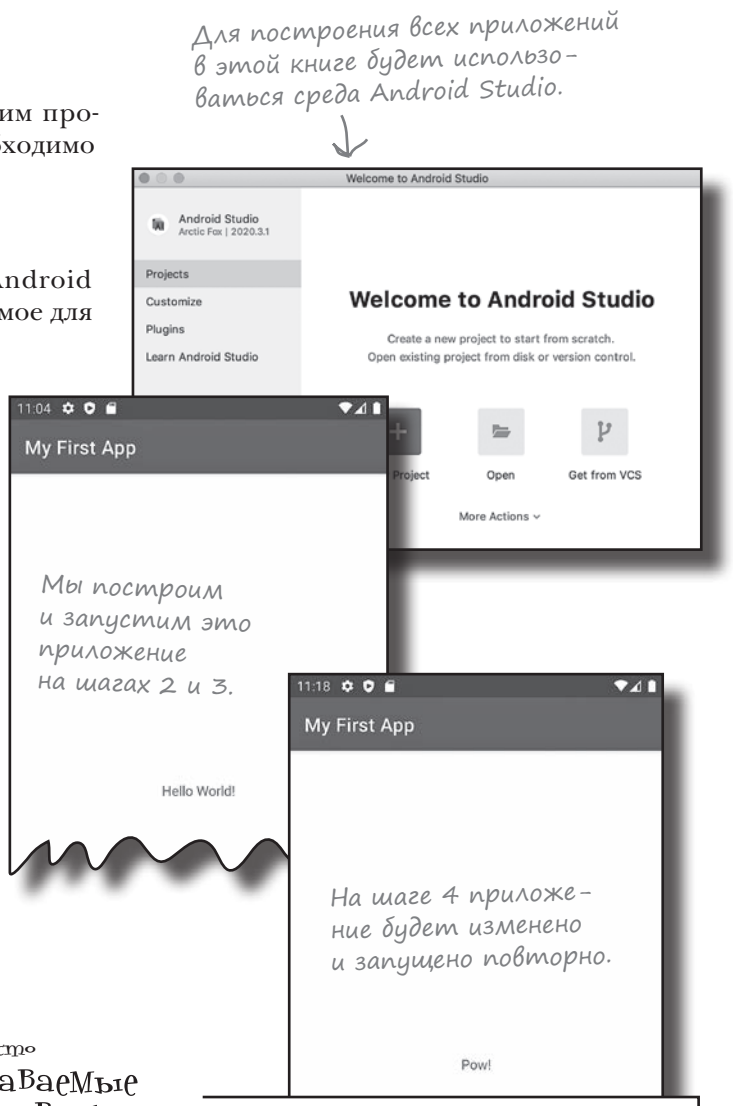
Давайте с ходу возьмемся за дело и построим простейшее Android-приложение. Для этого необходимо выполнить лишь несколько действий:

1. **Подготовка среды разработки.**
Необходимо установить среду Android Studio, включающую все необходимое для разработки Android-приложений.

2. **Построение простейшего приложения.**
Мы создадим в Android Studio простое приложение, которое будет выводить текст на экран.

3. **Запуск приложения.**
Мы запустим приложение на физическом и на виртуальном устройстве, чтобы увидеть его в действии.

4. **Изменение приложения.**
Наконец, мы внесем несколько изменений в созданное приложение и снова запустим его.



Часто Задаваемые Вопросы

В: Все Android-приложения пишутся на Kotlin?

О: Android-приложения также можно разрабатывать и на других языках (например, на Java), но лучше всего использовать Kotlin. Дело в том, что некоторые полезные возможности доступны только в Kotlin.

В: Как мне изучить Kotlin?

О: Наверное, мы необъективны, но на наш взгляд, Kotlin лучше всего изучать по нашей книге *Head First Kotlin**. В ней вы узнаете все, что необходимо знать о Kotlin, чтобы извлечь максимум пользы из книги.

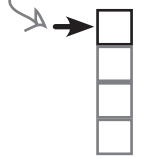
В: Вы упоминали, что для определения внешнего вида приложения вместо файлов макетов может использоваться код активности?

О: Да, с использованием инструментария, который называется Jetpack Compose.

Позднее в этой книге вы научитесь использовать Compose. А пока сосредоточимся на определении пользовательского интерфейса приложения в файлах макетов.

* *Гриффитс Дон, Гриффитс Дэвид.* Head First Kotlin.

Вы находитесь здесь.



Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Android Studio: среда разработки

Для разработки Android-приложений лучше всего использовать **Android Studio** — официальную интегрированную среду (IDE) для разработки Android-приложений.

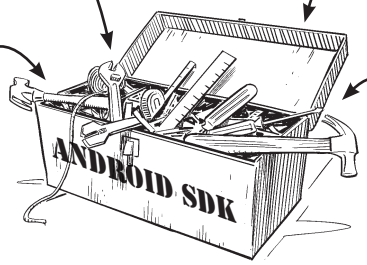
Android Studio базируется на среде IntelliJ IDEA — возможно, она вам уже знакома. В нее включены редакторы кода, UI-инструменты и шаблоны, упрощающие вашу жизнь в мире Android.

Также она включает **Android SDK** (Android Software Development Kit), необходимый для разработки Android-приложений. Android SDK включает исходные файлы Android и компилятор, используемый для компиляции кода в формат Android.

Основные компоненты Android SDK:

SDK Tools
Полный набор инструментов для разработки и отладки Android-приложений.

SDK Platform
Каждая версия Android содержит собственный пакет SDK Platform. Этот пакет позволяет компилировать приложения для данной версии Android и включает исходные файлы для этой версии.



SDK Build Tools
Инструменты, необходимые для построения Android-приложений.

SDK Platform Tools
Инструменты, взаимодействующие с платформой Android. Эти инструменты обладают обратной совместимостью, но некоторые возможности могут быть доступны только для новых версий Android.

Необходимо установить Android Studio

Так как Android Studio включает все необходимое для разработки Android-приложений, эта среда будет использоваться для построения всех приложений, представленных в книге.

Прежде чем двигаться дальше, **необходимо установить Android Studio на вашей машине**. О том, как это сделать, более подробно рассказано на следующей странице.

Установка Android Studio

Чтобы извлечь максимум пользы из книги, необходимо установить среду Android Studio. Мы не приводим здесь полные инструкции по установке, потому что они быстро устаревают, но если вы будете следовать инструкциям в интернете, все будет нормально.

Сначала проверьте системные требования Android Studio:

https://developer.android.com/studio#Requirements

Затем загрузите Android Studio по этому адресу:

https://developer.android.com/studio

Эти URL-адреса иногда изменяются. Если адреса не работают, поищите Android Studio в интернете — вы найдете соответствующие страницы.

и выполните инструкции по установке.

Когда установка будет завершена, откройте Android Studio и выполните инструкции по добавлению новейших инструментов SDK и библиотек поддержки.

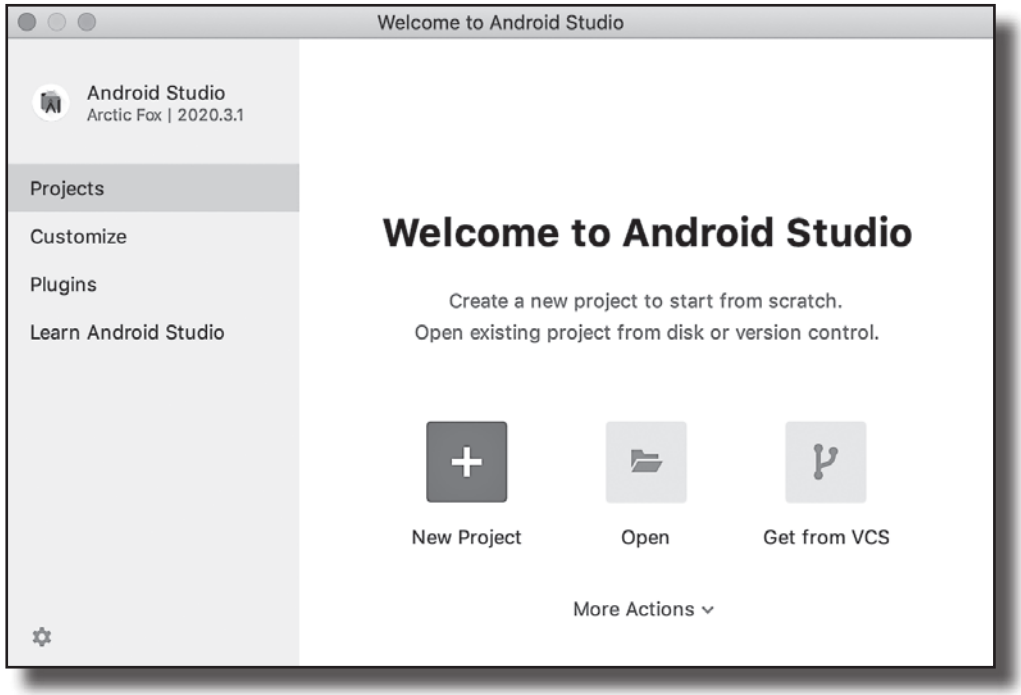
В этой книге используется версия Android Studio 2020.3.1 (также известная как Arctic Fox). Убедитесь в том, что вы установили эту или более позднюю версию.

Если ранее вы установили более старую версию Android Studio, мы рекомендуем восстановить настройки IDE по умолчанию. Для этого откройте меню File, выберите команду Manage IDE Settings, а затем Restore Default Settings.

Сбрасывает все старые настройки, которые могли храниться в Android Studio. Они могли бы помешать выполнению вашего кода.

Когда все будет сделано, на экране появляется заставка Android Studio:

Заставка Android Studio включает список возможных действий.

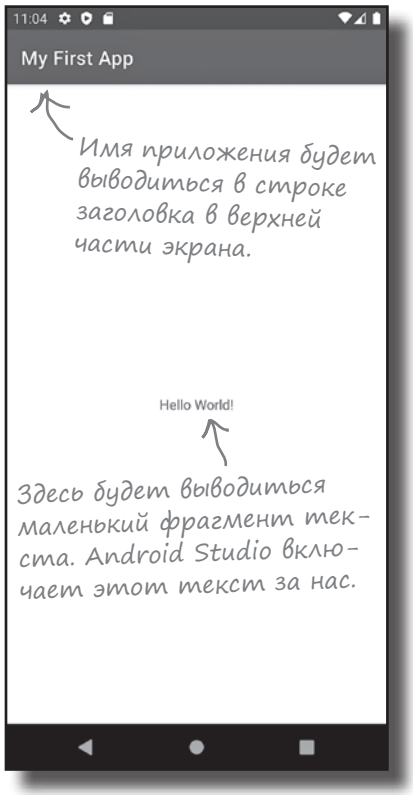


Этот шаг завершен, мы его вычеркиваем.

Построение простого приложения

Итак, среда разработки подготовлена, и можно переходить к созданию вашего первого Android-приложения. Вот как оно будет выглядеть:

Приложение, которое мы собираемся построить. Оно очень простое, но для первого Android-приложения этого вполне достаточно.



Давайте займемся построением приложения.

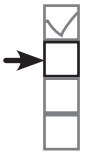
Часто Задаваемые Вопросы

В: А мне обязательно использовать Android Studio для построения Android-приложений? Нельзя ли использовать другую IDE?

О: Строго говоря, все, что вам нужно, — это инструменты для написания и компиляции Kotlin-кода, а также некоторые специализированные инструменты для преобразования откомпилированного кода в форму, которая может запускаться на Android-устройствах. Тем не менее Android Studio — официальная среда разработки Android, и команда Android рекомендует именно ее. Мы полагаем, что эта среда идеально подходит для разработки Android-приложений, поэтому мы остановили свой выбор на ней.

В: Возможно ли создавать Android-приложения без IDE?

О: Возможно, но это заметно увеличит объем работы. Гораздо быстрее и проще работать в IDE, поэтому мы рекомендуем использовать Android Studio.



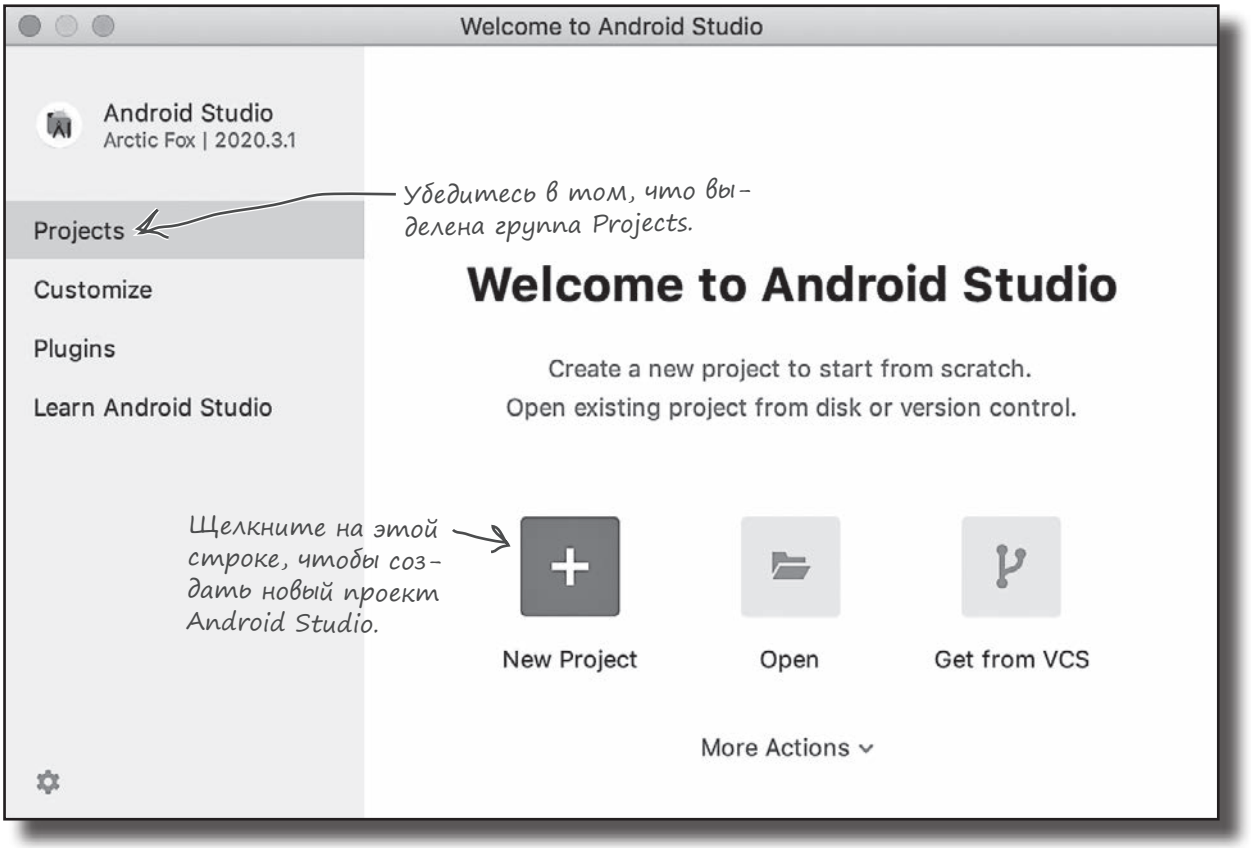
Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

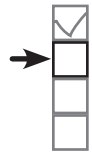
Как построить приложение

Каждый раз, когда вы создаете новое приложение, для него необходимо создать новый проект. Убедитесь в том, что среда Android Studio открыта, и повторяйте за нами.

1. Создание нового проекта.

На заставке Android Studio перечислены некоторые возможные операции. Мы хотим создать новый проект; убедитесь в том, что выбрана категория Projects, и щелкните на варианте «New Project».

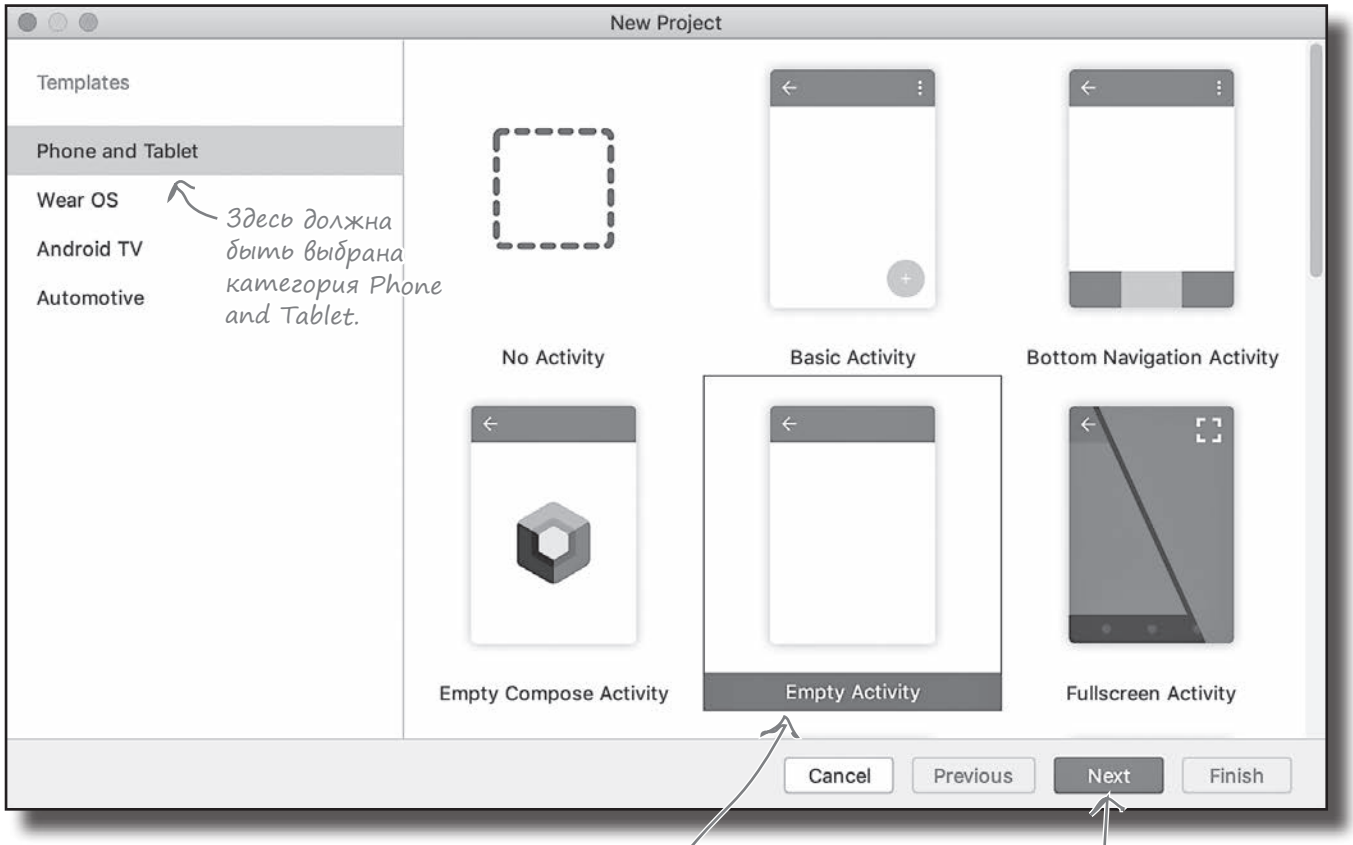


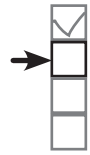


Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

2. Выбор шаблона приложения.

Далее необходимо задать тип создаваемого проекта Android Studio. Мы создадим приложение с пустой активностью, которое будет работать на телефоне или на планшете, поэтому проследите за тем, чтобы была выбрана категория Phone and Tablet, и выберите пустую активность (Empty Activity). Через несколько страниц вы узнаете, что дает вам шаблон Empty Activity, а пока перейдите к следующему шагу нажатием кнопки Next.





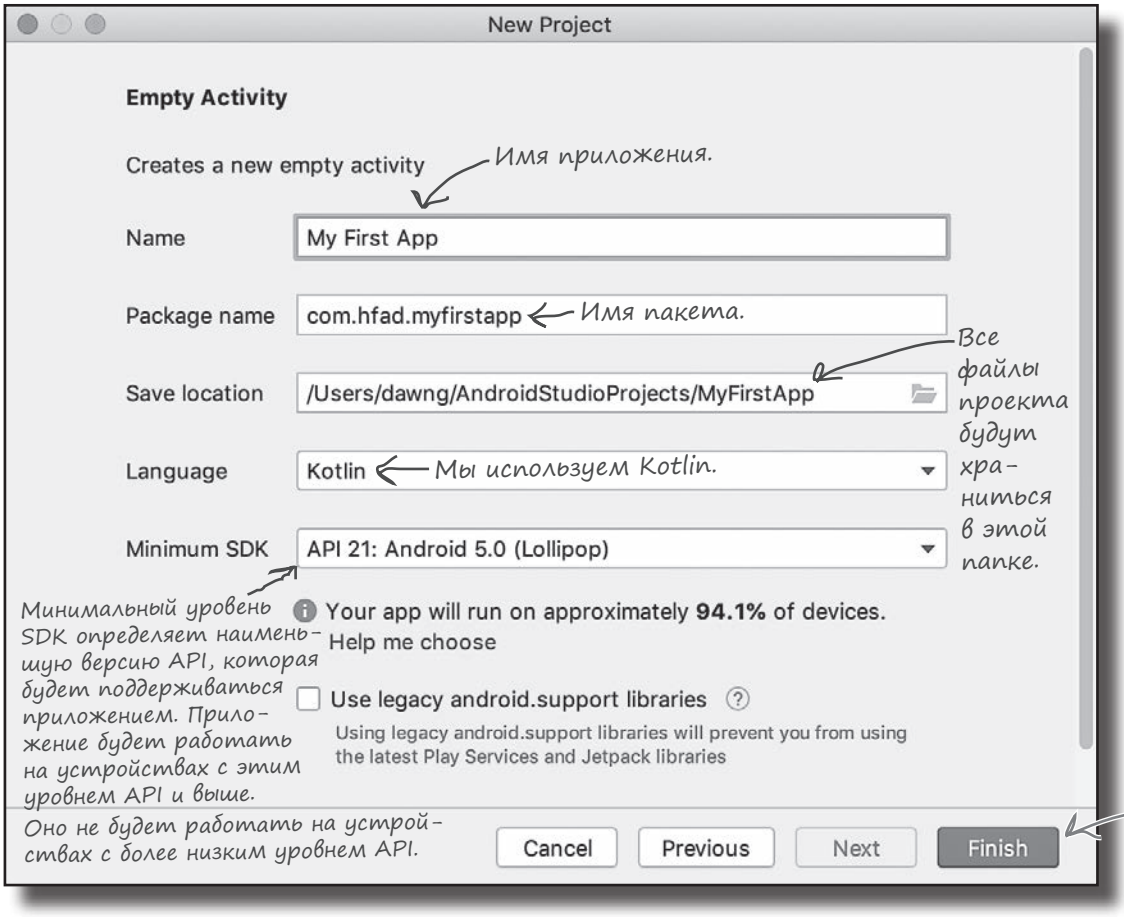
Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

3. Настройка проекта.

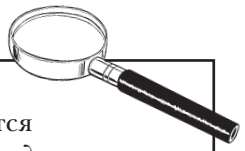
Теперь необходимо настроить конфигурацию приложения: указать его имя, имя пакета и где должны храниться его файлы. Введите имя «My First App» и имя пакета «com.hfad.myfirstapp», подтвердите место сохранения по умолчанию.

Также необходимо сообщить Android Studio, какой язык программирования вы будете использовать, и задать минимальную версию SDK. Этот параметр определяет наименьшую версию Android, которая будет поддерживаться приложением: уровни SDK более подробно рассматриваются на следующей странице.

Выберите язык Kotlin и минимальную версию API 21, чтобы приложение работало на большинстве устройств. Когда вы щелкнете на кнопке Finish, Android Studio создаст проект. Через пару страниц вы узнаете, что при этом происходит.



Версии Android под увеличительным стеклом



Вероятно, вам не раз доводилось слышать, как применительно к Android упоминаются разные «вкусные» названия: Nougat (нуга), Oreo и Pie (пирог). Что это за кондитерская?

Каждой версии Android присваивается номер и кодовое имя. Номер версии определяет конкретную версию Android (например, 9.0), тогда как кодовое имя представляет собой чуть более общее «дружественное» имя, которое может объединять сразу несколько версий Android (например, Pie). Под «уровнем API» понимается версия API, используемых приложением. Например, Android версии 9.0 соответствует уровень API 28.

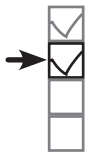
После Pie кодовые имена Android перестали быть «вкусными». Например, Android версии 10.0 называется просто «Android 10».

Версия	Кодовое имя	Уровень API
1.0		1
1.1	Petit Four	2
1.5	Cupcake	3
1.6	Donut	4
2.0–2.1	Eclair	5–7
2.2–2.2.3	Froyo	8
2.3–2.3.7	Gingerbread	9–10
3.0–3.2.6	Honeycomb	11–13
4.0–4.0.4	Ice Cream Sandwich	14–15
4.1–4.3.1	Jelly Bean	16–18
4.4–4.4.4	KitKat	19–20
5.0–5.1.1	Lollipop	21–22
6.0–6.0.1	Marshmallow	23
7.0–7.1.2	Nougat	24–25
8.0–8.1	Oreo	26–27
9.0	Pie	28
10.0	10	29
11.0	11	30
12.0	12	31

Сейчас эти версии уже никем не используются.

На большинстве устройств используется один из этих уровней API.

При разработке Android-приложений необходимо учитывать, с какими версиями Android должно быть совместимо ваше приложение. Если вы укажете, что приложение совместимо только с самой последней версией SDK, может оказаться, что оно не запускается на очень многих устройствах. Чтобы получить информацию о процентном распределении версий по устройствам, выберите вариант «Help me choose» при создании нового проекта.



Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Вы создали свой первый проект для Android

После завершения мастера New Project среда Android Studio создает новый проект, для чего ей потребуется около минуты. В это время происходит следующее:

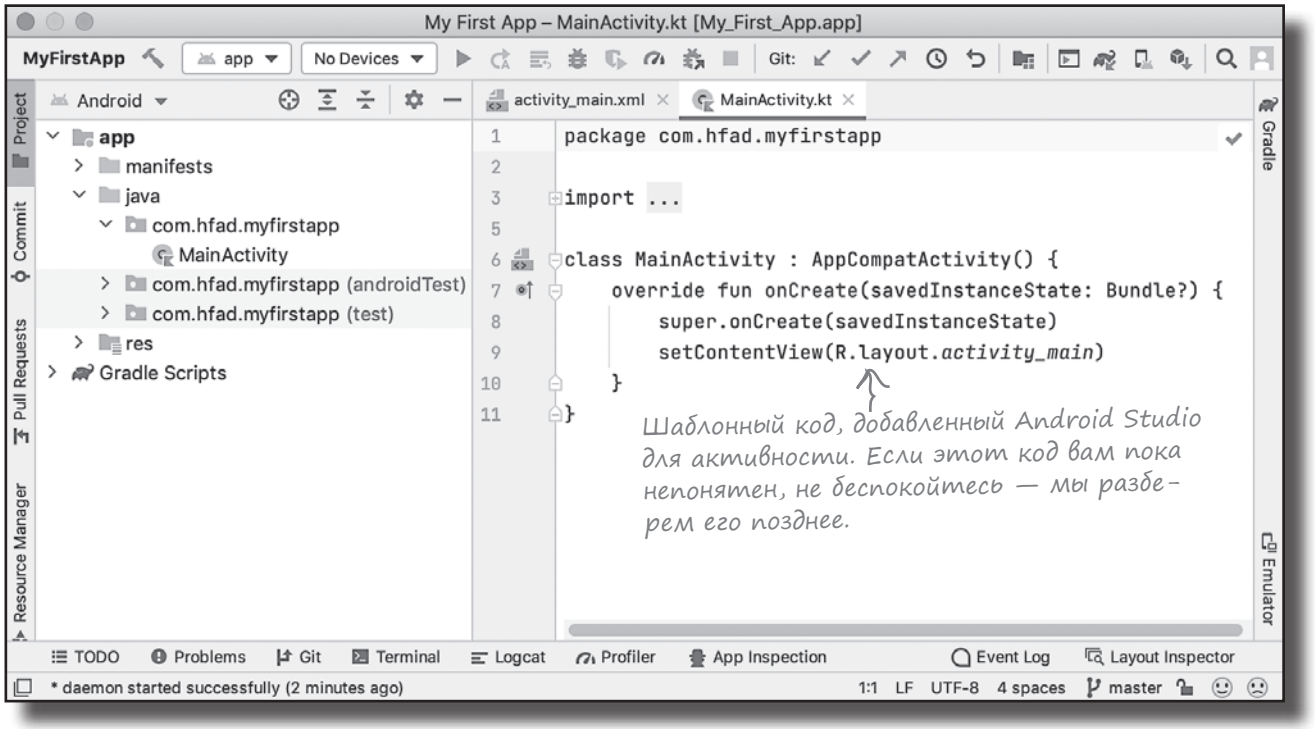
Проект настраивается в соответствии с вашими указаниями.
Android Studio проверяет минимальную версию SDK, которая должна поддерживаться приложением, и включает все файлы и папки, необходимые для минимального приложения. Также среда настраивает структуру пакета и задает имя приложения.

Добавляется шаблонный код.
Шаблонный код состоит из разметки макета, написанной на XML, и кода активности, написанного на Kotlin. В данной главе вы больше узнаете об этих двух разновидностях кода.

Когда среда Android Studio завершит создание проекта, она автоматически откроет его.

Вот как выглядит наш проект на текущий момент (не беспокойтесь, если сейчас все выглядит слишком сложно, — на нескольких ближайших страницах мы все объясним):

Так выглядит проект в Android Studio.



Структура нового проекта

Android-приложение в действительности представляет собой набор файлов, размещенных в четко определенной структуре папок; Android Studio создает все эти папки за вас при создании нового приложения. Эту структуру папок можно просмотреть на панели у левого края окна Android Studio.

В структуре папок присутствуют файлы разных типов

На панели отображаются все проекты, открытые в настоящий момент. В данном случае открыт один проект с именем MyFirstApp — тот, который мы только что создали.

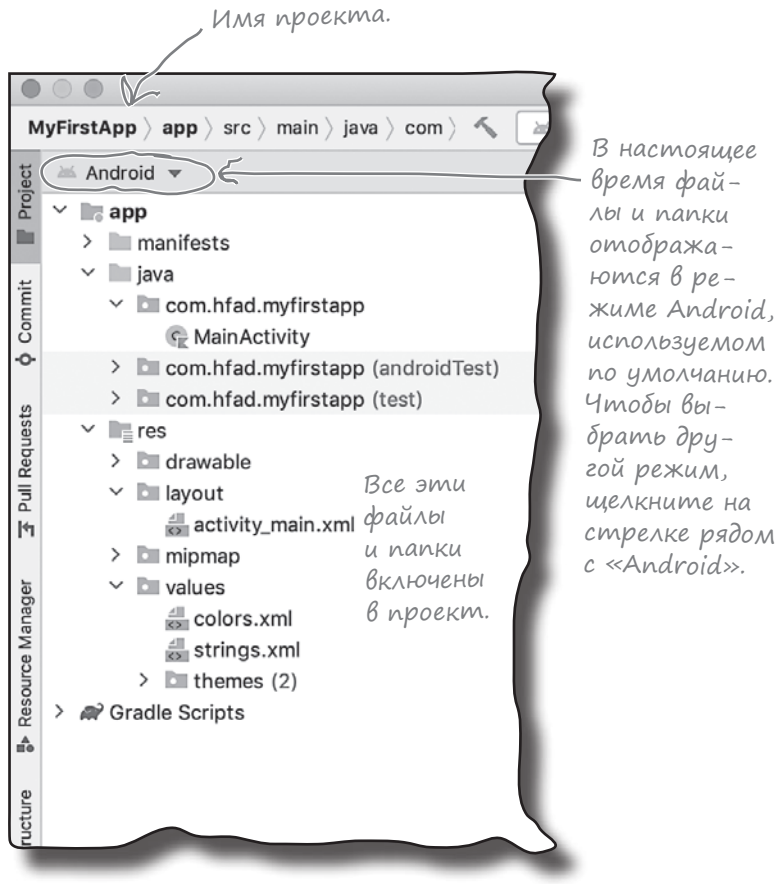
Просмотрев структуру папок, вы увидите, что мастер создал за вас папки и файлы разных типов:

Исходные файлы Kotlin и XML
Android Studio автоматически создает файл активности с именем *MainActivity.kt* и макет с именем *activity_main.xml*.

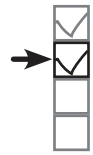
Файлы ресурсов
К этой категории относятся файлы изображений по умолчанию, темы, которые могут использоваться вашим приложением, и все общие строковые данные, к которым может обращаться приложение.

Библиотеки Android
В окне мастера вы указали минимальную версию SDK, с которой должно быть совместимо ваше приложение. Android Studio включает в приложение библиотеки Android, актуальные для этой версии.

Файлы конфигурации
Файлы конфигурации сообщают Android, что содержит приложение и как его следует выполнять.



Давайте повнимательнее присмотримся к некоторым ключевым файлам и папкам в мире приложений Android.

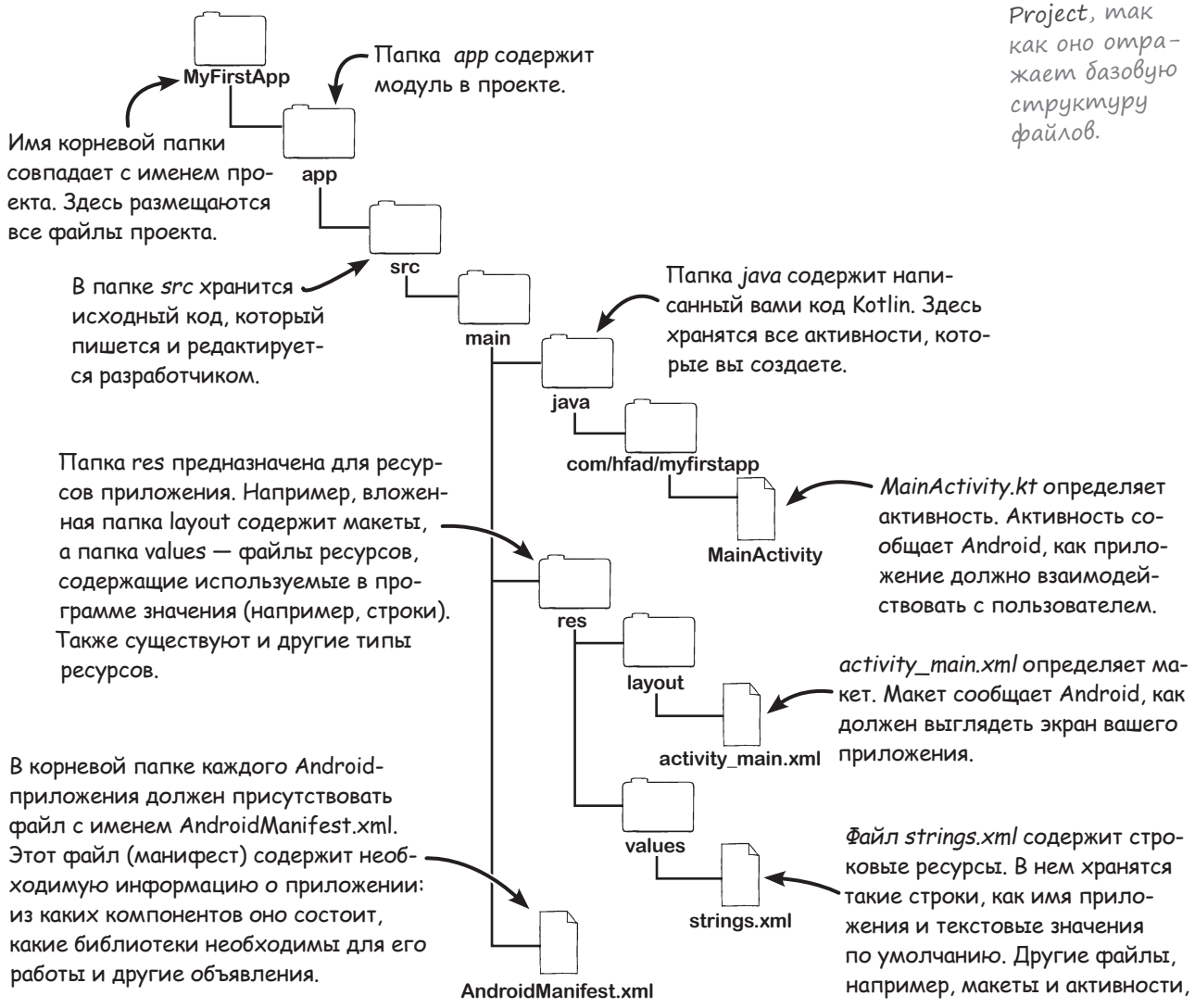


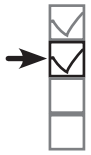
Ключевые файлы вашего проекта

Проекты Android Studio используют систему сборки Gradle для компиляции и развертывания приложений. Проекты gradle имеют стандартную структуру. Ниже описаны некоторые ключевые файлы и папки, с которыми вам предстоит работать.

Чтобы увидеть это представление структуры папок, смените представление панели Explorer в Android Studio с Android на Project. Для этого щелкните на стрелке в верхней части панели Explorer и выберите вариант Project.

Кнопка со стрелкой используется для изменения представления панели. Обычно выбирается представление Project, так как оно отражает базовую структуру файлов.





Редактирование кода в Android Studio

Для просмотра и изменения файлов используются различные редакторы Android Studio. Сделайте двойной щелчок на файле, с которым вы хотите работать; его содержимое появляется в середине окна Android Studio.

Редактор кода

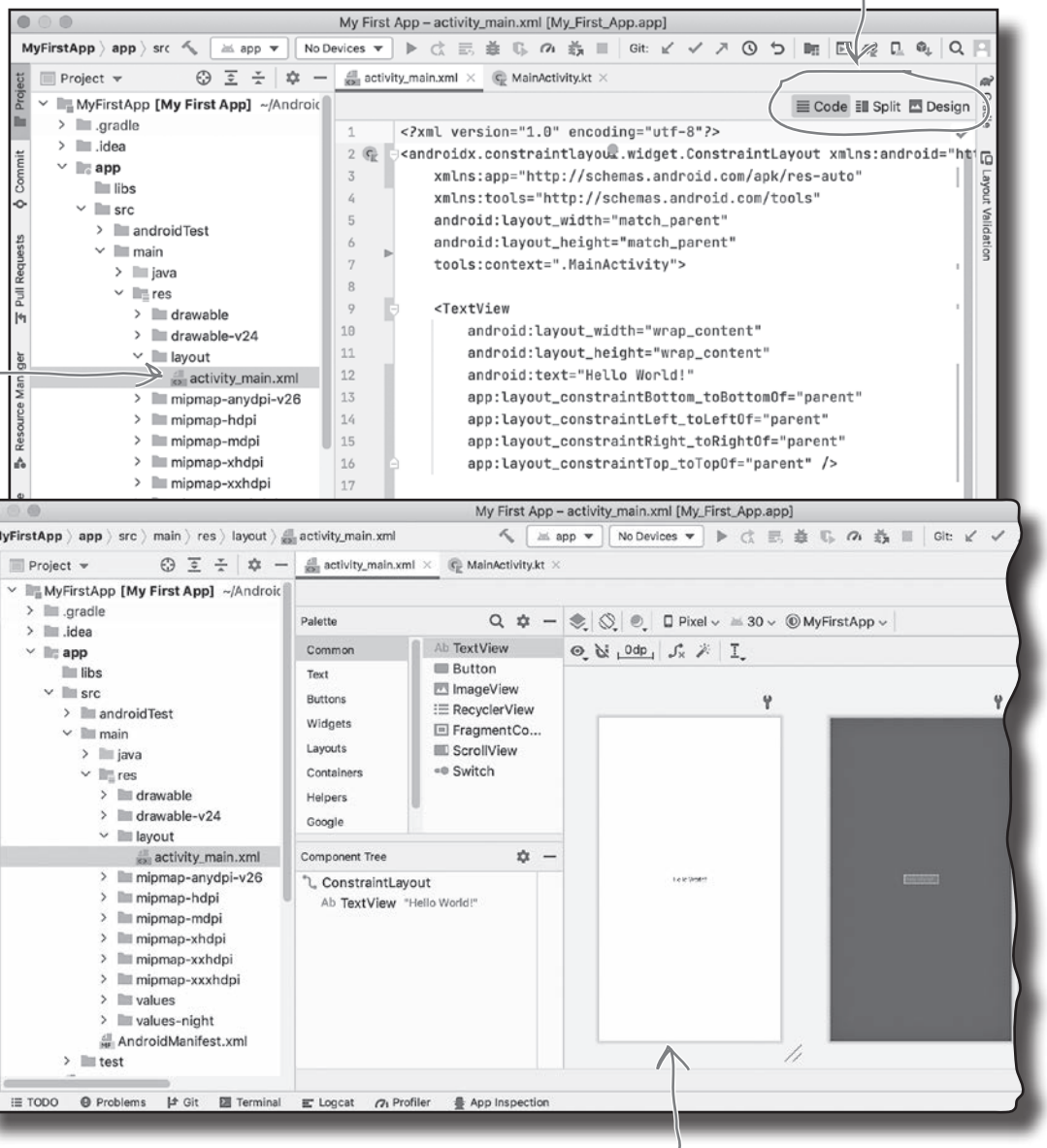
Большинство файлов отображается в редакторе кода. По сути, это обычный текстовый редактор, но с поддержкой таких дополнительных возможностей, как цветовое выделение синтаксиса и проверка кода.

Сделайте двойной щелчок на файле. Содержимое файла отображается на панели.

Визуальный редактор

При редактировании макета (например, *activity_main.xml*) появляется дополнительная возможность. Вместо того чтобы редактировать код, можно воспользоваться визуальным редактором. Он позволяет перетащить компоненты графического интерфейса на макет и расположить их так, как вы считаете нужным. Редактор кода и визуальный редактор обеспечивают разные представления одного файла, и вы можете переключаться между ними.

Эти кнопки используются для выбора редактора.



Макеты можно редактировать в визуальном редакторе, перетаскивая компоненты мышью.

Чего мы добились

К настоящему моменту мы решили две задачи:

1. **Мы настроили среду разработки.**
   Для разработки Android-приложений используется среда Android Studio, которую необходимо установить на вашей машине.

2. **Мы построили простое приложение.**
   Мы воспользовались Android Studio для создания нового проекта Android.

Вы получили некоторое представление о том, как выглядит приложение в Android Studio и как взаимодействуют между собой его части. Но ведь вам хотелось бы увидеть, как оно *по-настоящему* работает, верно?

Android Studio позволяет запускать приложения двумя способами: на физическом и на виртуальном Android-устройстве. Каждый способ будет рассмотрен через несколько страниц.

Часто Задаваемые Вопросы

В: Почему Android Studio хранит код Kotlin в папке с именем *java*?

О: Прежде чем команда Android приняла Kotlin как основной язык разработки, большинство Android-приложений разрабатывалось на Java. Имя папки *java* осталось с того времени, но оно может измениться в будущем.

В: Вы сказали, что проект включает активность с именем *MainActivity.kt* и макет с именем *activity_main.xml*. Откуда они взялись?

О: При создании проекта мы выбрали вариант создания проекта с пустой активностью. Среда Android Studio автоматически присвоила файлу активности имя *MainActivity.kt*, а также добавила соответствующий файл макета с именем *activity_main.xml*. Эти файлы будут более подробно рассмотрены позднее в этой главе.

В: Вы сказали, что проекты Android Studio используют Gradle. Напомните, что это такое?

О: Gradle — программа для компилирования, построения и распространения кода. Gradle используется во многих IDE, и возможности этой программы не ограничиваются разработкой Android-приложений.

Все проекты Gradle формируют стандартную структуру папок, а Android Studio использует эту структуру для всех своих проектов.

В: Почему Android Studio использует Gradle?

О: Когда вы разрабатываете Android-приложения, часто возникает необходимость во включении дополнительных библиотек, не входящих в Android SDK. Gradle может загрузить эти библиотеки за вас, что значительно упрощает вашу жизнь как программиста.

Gradle также использует Groovy и Kotlin как язык сценариев, а это означает, что с Gradle можно легко создавать достаточно сложные сборки.

В: Что мне необходимо знать о Gradle?

О: Для усвоения большей части материала опыт Gradle вообще не понадобится. Мы объясним все необходимое в тот момент, когда это потребуется.

Кто и что делает?

Ниже приведен фрагмент кода из файла активности *MainActivity.kt*. Да, мы знаем, что вы еще ни разу не видели код активности, и все же попробуйте соединить каждое из описаний в нижней части страницы с правильной строкой кода. Мы уже провели одну линию, чтобы вам было проще взяться за дело.

MainActivity.kt

```
package com.hfad.myfirstapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

}
```

Имя пакета.

Классы Android, используемые в `MainActivity`.

Выбор используемого макета.

Реализация метода `onCreate()` из класса `AppCompatActivity`. Этот метод вызывается при первом создании активности.

`MainActivity` расширяет класс `AppCompatActivity`.

Кто и что делает? Решение

Ниже приведен фрагмент кода из файла активности *MainActivity.kt*. Да, мы знаем, что вы еще ни разу не видели код активности, и все же попробуйте соединить каждое из описаний в нижней части страницы с правильной строкой кода. Мы уже провели одну линию, чтобы вам было проще взяться за дело.

MainActivity.kt

```
package com.hfad.myfirstapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

}
```

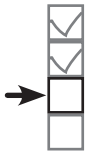
Имя пакета.

Классы Android, используемые в `MainActivity`.

Выбор используемого макета.

Реализация метода `onCreate()` из класса `AppCompatActivity`. Этот метод вызывается при первом создании активности.

`MainActivity` расширяет класс `AppCompatActivity`.



Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Запуск приложения на физическом устройстве

Если у вас имеется Android-устройство с версией Lollipop и выше, вы сможете использовать его для запуска только что созданного приложения.

Версию Android можно проверить в настройках устройства. Lollipop — версия 5.0, поэтому должна использоваться версия 5.0 и выше.

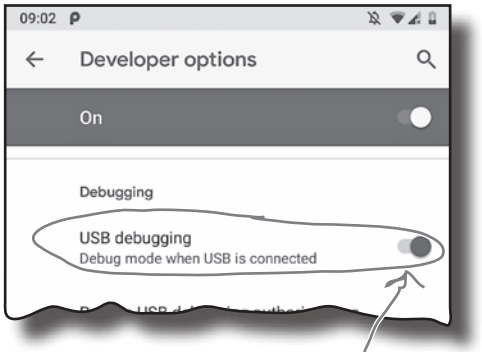
Чтобы запустить приложение на физическом устройстве, выполните следующие действия:

1. Включите отладку USB на вашем устройстве.

Чтобы разрешить Android Studio выполнять приложения на вашем устройстве, необходимо разрешить режим отладки USB. Эта возможность включается в разделе «Developer options», отключенном по умолчанию.

Да, серьезно.

На своем устройстве выберите команду Settings → About Phone и коснитесь номера сборки семь раз. Тем самым вы включите доступ к настройкам разработчика. Выберите команду Settings → System → Advanced → Developer и включите отладку USB.



Необходимо включить отладку USB.

2. Настройте ваш компьютер для обнаружения устройства.

Если вы работаете на Mac, этот шаг можно пропустить.

Если вы используете Windows, необходимо установить драйвер USB, если он еще не установлен. Обновленные инструкции доступны по адресу:

https://developer.android.com/studio/run/oem-usb

Если вы используете Ubuntu Linux, необходимо создать файл правил `udev`. Обновленные инструкции по выполнению этой операции доступны по следующему адресу:

https://developer.android.com/studio/run/device#setting-up

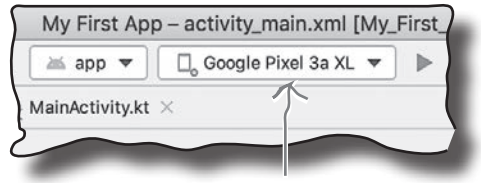
3. Подключите устройство к компьютеру кабелем USB.

Вероятно, вам будет предложено разрешить отладку USB. В таком случае выберите вариант «Always allow from this computer» и нажмите кнопку OK.

4. Запустите приложение.

Наконец, выберите устройство в списке устройств на верхней панели инструментов Android Studio (если его там нет, на своем устройстве выберите команду Settings → Connected devices, выберите USB, а затем вариант «File transfer»). Затем запустите приложение командой «Run 'app'» из меню Run. Android Studio построит проект, установит приложение на ваше устройство и запустит его.

При создании приложения был выбран минимальный уровень API 21 (Lollipop). Чтобы приложение работало, на устройстве должна быть установлена эта версия Android или выше.



Перед запуском приложения проследите за тем, чтобы было выбрано ваше устройство. В данном случае используется Pixel 3a XL.

Приложение будет рассмотрено через несколько страниц — после описания его запуска на виртуальном устройстве.

Как запустить приложение на виртуальном устройстве

Если у вас под рукой нет Android-устройства или на устройстве установлена неподходящая версия Android, приложение можно запустить на виртуальном устройстве. Запуск приложения на виртуальном устройстве может быть полезен, если вы хотите видеть, как приложение выглядит на устройстве, которое вам недоступно, или протестировать его в другой версии Android.

Android SDK содержит встроенный эмулятор, который можно использовать для создания одного или нескольких **виртуальных устройств** (**Android Virtual Devices**, AVDs). После этого приложение можно запустить на AVD так, словно оно выполняется на физическом устройстве.

С системными требованиями для использования эмулятора можно ознакомиться по адресу https://developer.android.com/studio/run/emulator#requirements

Эмулятор точно воссоздает аппаратное окружение устройства Android: от центрального процессора и памяти до звуковых микросхем и экрана. Эмулятор построен на базе существующего эмулятора QEMU, похожего на другие виртуальные машины, с которыми вам, возможно, доводилось работать, такие как VirtualBox или VMWare.

Внешний вид и поведение AVD зависят от заданных вами параметров. Если, допустим, создать AVD для Pixel 3 с Android 11, эмулятор будет выглядеть и вести себя так, словно на вашем компьютере имеется внутреннее устройство Pixel 3 с нужной версией Android.

AVD в эмуляторе Android. Виртуальное устройство основано на устройстве Pixel 3 с версией Android 11. Оно имеет те же спецификации устройства и работает так же, как на виртуальном устройстве.

Давайте создадим AVD, чтобы приложение можно было запустить в эмуляторе.

Создание Android Virtual Device (AVD)

Процесс создания AVD в Android Studio состоит из нескольких шагов. Мы создадим AVD для Pixel 3 с уровнем API 30 (Android 11), чтобы вы могли видеть, как ваше приложение выглядит и ведет себя на устройствах этого типа. Последовательность действий остается более или менее постоянной для всех типов виртуальных устройств.

Откройте AVD Manager

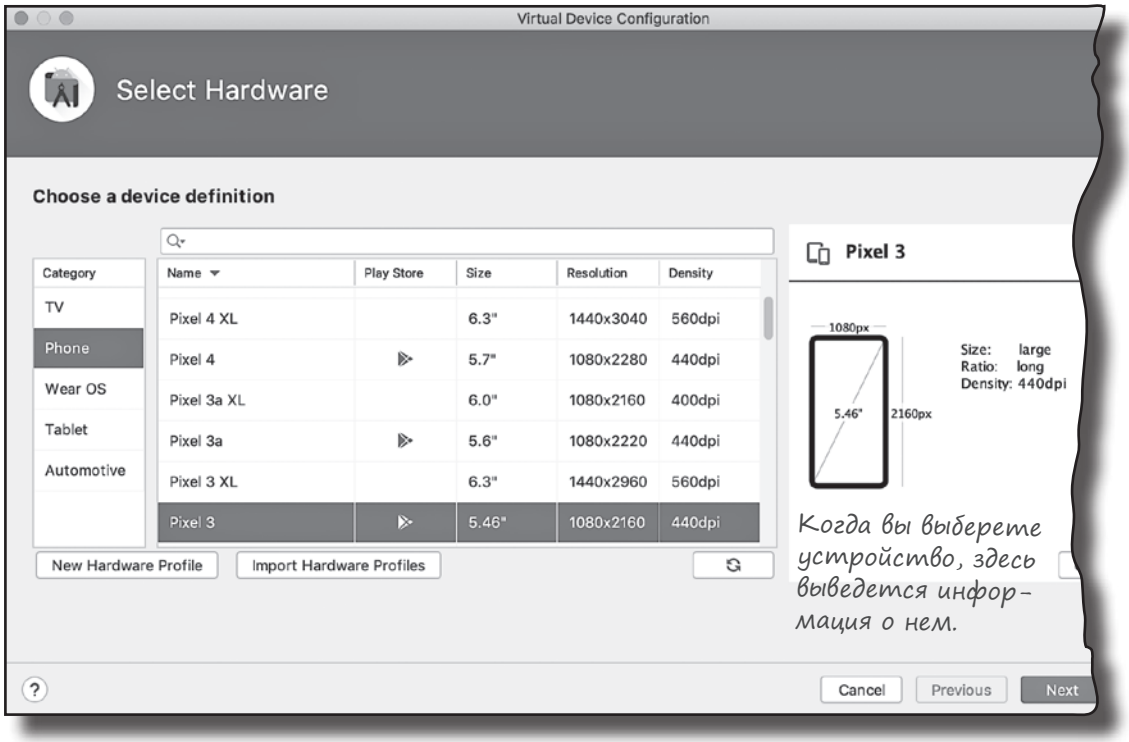
В диспетчере AVD Manager можно создавать новые AVD, а также просматривать и редактировать уже созданные виртуальные устройства. Чтобы запустить его, выберите в меню Tools пункт AVD Manager.

Если вы еще не создали ни одного виртуального устройства, открывается окно с предложением создать его. Щелкните на кнопке Create Virtual Device.

Щелкните на этой кнопке, чтобы создать AVD.

Выберите устройство

На следующем экране вам будет предложено выбрать определение устройства, то есть тип устройства, который будет эмулировать AVD. В вашем распоряжении различные телефоны, планшеты, носимые или телевизионные устройства. Давайте посмотрим, как будет выглядеть наше приложение на телефоне Pixel 3. Выберите в меню Category пункт Phone, затем выберите в списке Pixel 3. Щелкните на кнопке Next.



Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

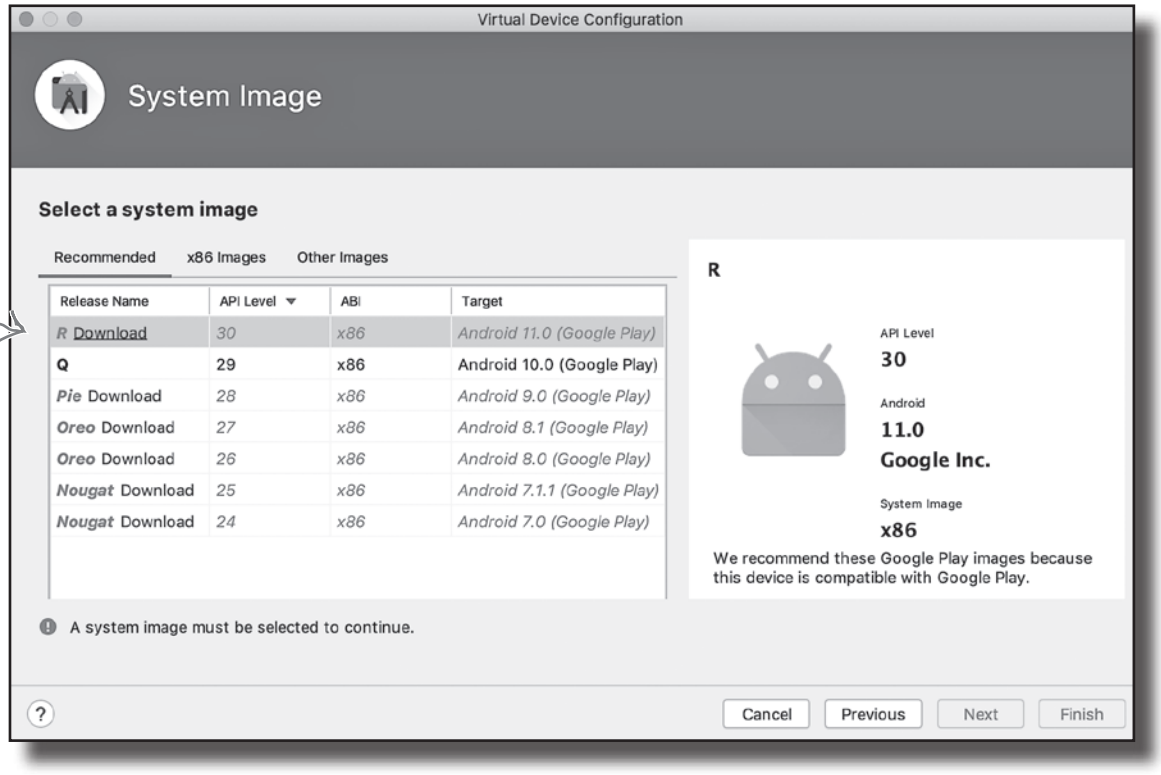
Выбор образа системы

Далее следует выбрать образ системы, то есть версию операционной системы Android, которая должна работать на AVD. Необходимо выбрать версию Android, совместимую с создаваемым приложением. Версия должна быть *не ниже* минимальной версии SDK, поддерживаемой приложением.

При создании нашего проекта Android была выбрана минимальная версия SDK – API уровня 21. Это означает, что необходимо выбрать образ системы с уровнем API не менее 21 (Lollipop). Если будет выбрана более старая версия, приложение не будет работать на таком устройстве.

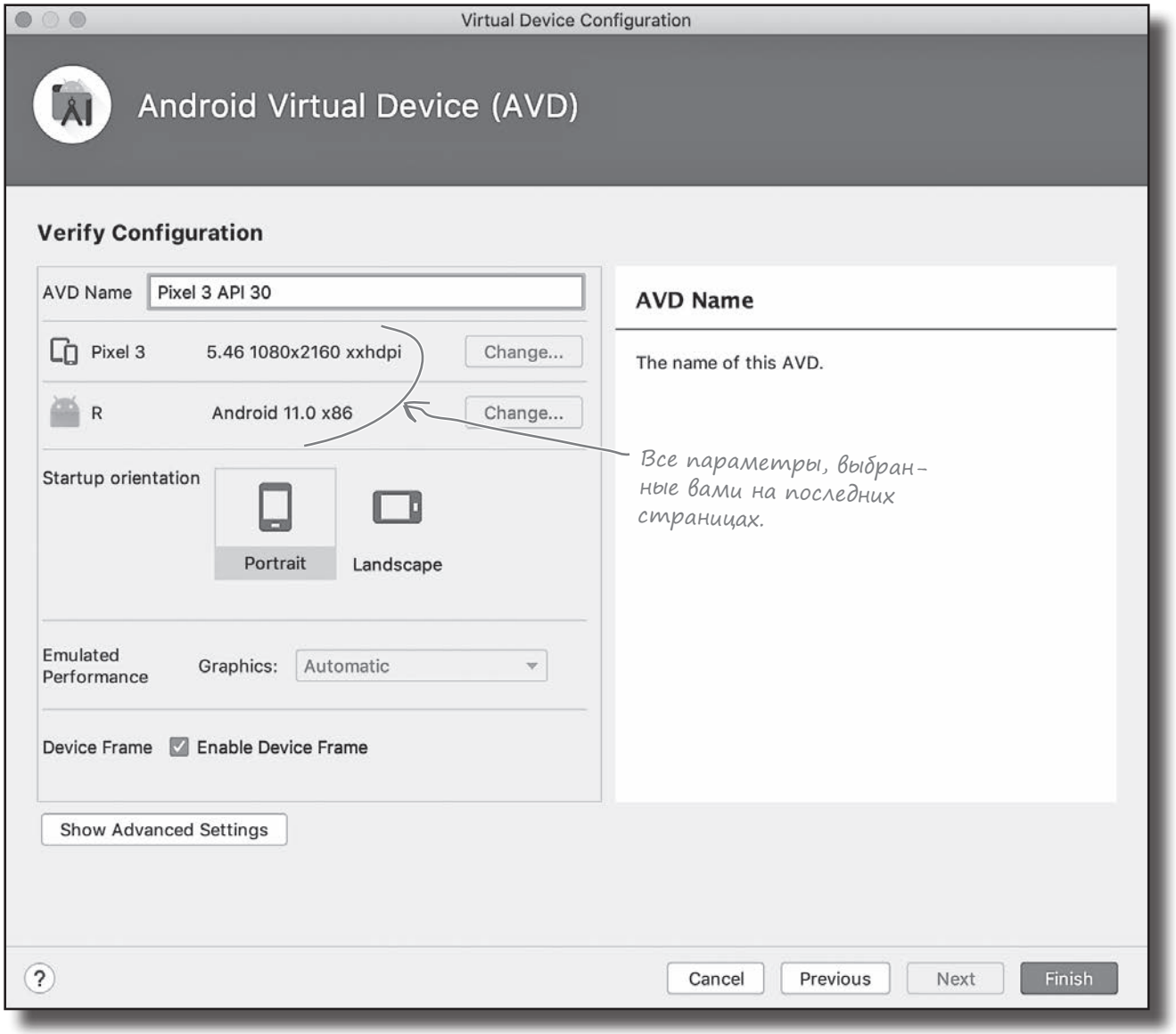
Здесь мы хотим посмотреть, как будет выглядеть приложение в относительно новой версии Android, поэтому выберите образ системы с именем R и целевой версией Android 11.0 (API уровня 30), после чего щелкните на кнопке Next.

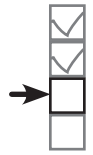
Если этот образ системы у вас не установлен, появляется ссылка для загрузки Download. Щелкните на ссылке, чтобы загрузить и установить образ системы.



Проверка конфигурации AVD

На следующем экране вам будет предложено подтвердить конфигурацию AVD. На нем приведена сводка параметров, выбранных вами на нескольких последних экранах, а также предоставляется возможность изменить их. Подтвердите значения и щелкните на кнопке Finish.





Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Создание виртуального устройства

При нажатии кнопки Finish Device Manager создает виртуальное устройство и включает его в список виртуальных устройств AVD Manager:

Проверьте, есть ли в списке новый AVD, затем закройте AVD Manager.

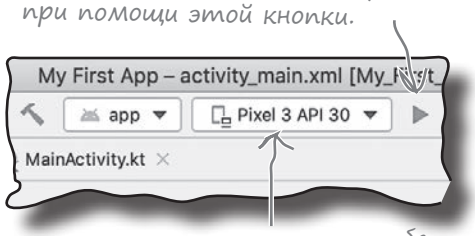
Запуск приложения на AVD

После того как устройство AVD будет создано, на нем можно запустить приложение.

Чтобы запустить приложение, убедитесь в том, что виртуальное устройство выбрано в списке устройств на верхней панели инструментов Android Studio, затем запустите приложение командой «Run 'app'» меню Run.

Загрузка AVD потребует времени. Пока вы ожидаете, посмотрим, что происходит под капотом при выполнении команды Run.

Также можно запустить приложение при помощи этой кнопки.



Перед запуском приложения убедитесь в том, что выбрано это устройство AVD. Здесь используется устройство Pixel 3.

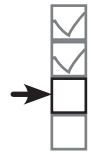
Часто Задаваемые Вопросы

В: Мне придется создавать новое устройство AVD каждый раз, когда я создаю новое приложение?

О: Нет. После того как вы создадите AVD, виртуальное устройство можно будет использовать для запуска любого приложения, работающего в этой версии Android.

В: Могу ли я создать несколько AVD?

О: Да! Вы можете создать AVD для эмуляции разных версий Android, включая бета-версии Android. Это позволяет вам протестировать, как приложение будет работать в каждой версии. Например, можно создать AVD для планшетного устройства, чтобы увидеть, как приложение будет выглядеть и вести себя на устройствах с большим размером экрана.



Компиляция, упаковка, развертывание, запуск

Команда Run не просто запускает ваше приложение. Она также выполняет все подготовительные операции, необходимые для запуска приложения.

Краткий обзор того, что при этом происходит:

APK-файл представляет собой пакет с приложением Android. По сути, это ZIP- или JAR-файл для приложений Android.

1. **Исходные файлы Kotlin компилируются в байт-код.**

2. **Создается пакет приложения Android (APK-файл).**
   APK-файл включает откомпилированные файлы Kotlin вместе со всеми библиотеками и ресурсами, необходимыми приложению.

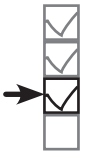
3. **APK устанавливается на устройстве.**
   Для виртуального устройства Android Studio запускает эмулятор и ожидает, пока AVD не станет активным, после чего устанавливает APK.

   Для физического устройства Android Studio просто устанавливает APK.

4. **Устройство запускает главную активность приложения.**
   Приложение появляется на экране устройства. И вы можете с ним поэкспериментировать.

Теперь вы знаете, что происходит при выполнении команды Run.
Давайте посмотрим, как выглядит построенное нами приложение.

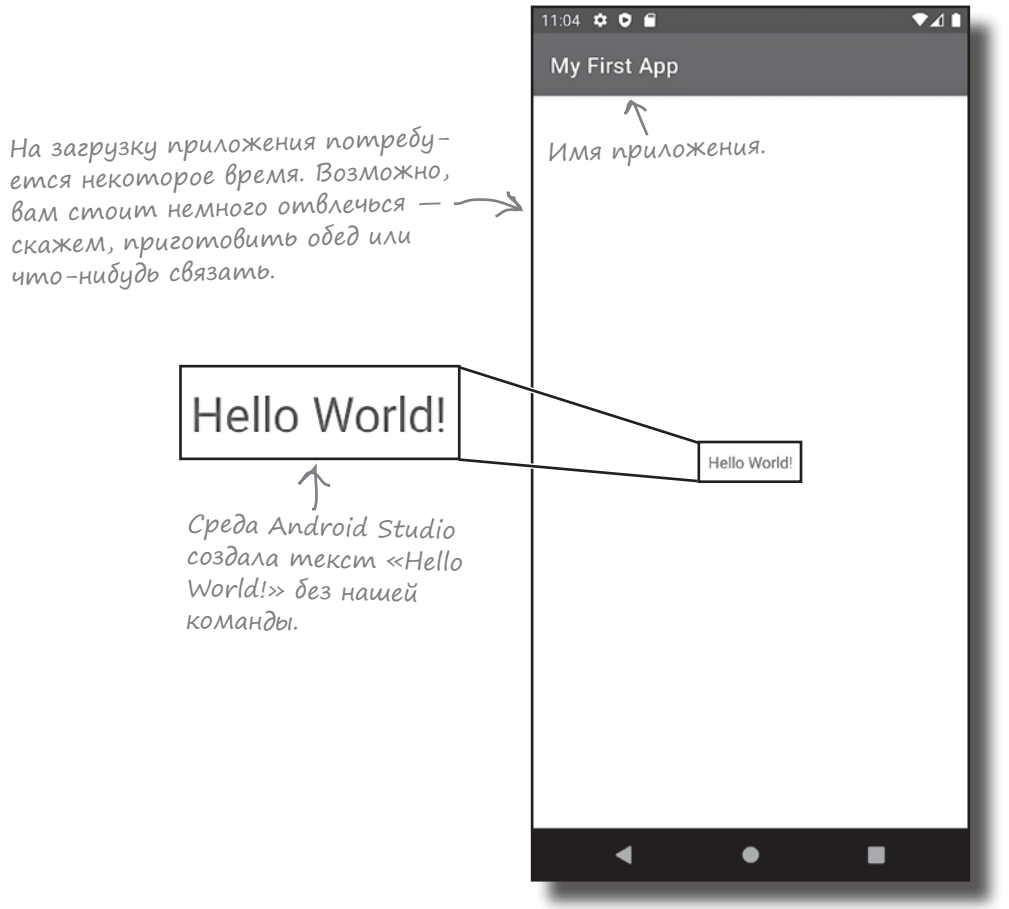
Тест-драйв



Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Запустите приложение на физическом или виртуальном устройстве командой «Run 'app'» из меню Run.

Android Studio загружает приложение на устройство и запускает его. Название приложения «My First App» появляется в верхней части экрана, а в центре выводится текст «Hello World!».



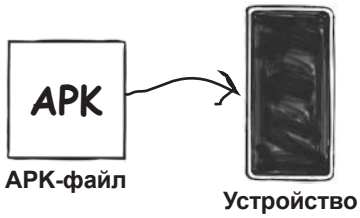
А теперь разберемся, что же только что произошло.

Настройка среды
Построение приложения
Запуск приложения
Изменение приложения

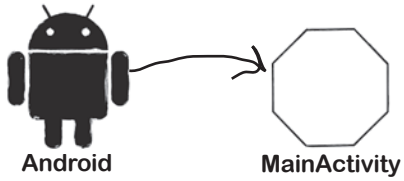
Что только что произошло?

Давайте разберемся по шагам, что происходит при запуске приложения:

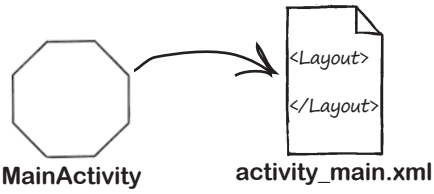
1. **Android Studio устанавливает приложение на устройство.**
Если устройство является виртуальным, оно ожидает запуска эмулятора, прежде чем устанавливать приложение.



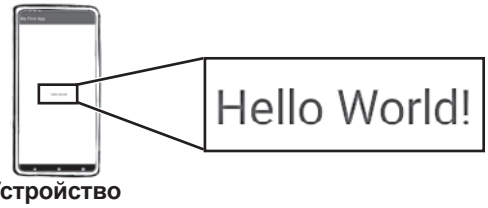
2. **Android запускает главную активность приложения.**
При этом для создания объекта `MainActivity` используется код *MainActivity.kt* (который был автоматически включен в проект средой Android Studio).



3. **MainActivity указывает, что должен использоваться макет activity_main.xml.**



4. **Макет отображается на экране.**
Текст «Hello World!» выводится в середине экрана.



Часто Задаваемые Вопросы

В: И Kotlin, и Java — языки виртуальной машины Java. Означает ли это, что приложения Android выполняются в JVM?

О: Нет. Каждое приложение выполняется в собственном процессе, использующем исполнительную среду Android (ART). При этом приложения выполняются намного быстрее и эффективнее.

Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Доработка приложения

До настоящего момента мы построили базовое приложение Android и увидели, как оно выполняется на физическом или виртуальном устройстве. Затем мы займемся доработкой приложения.

На данный момент приложение выводит временный текст «Hello World!», который был сгенерирован мастером. Давайте заменим этот текст чем-то другим. Что же для этого нужно сделать?

Чтобы ответить на этот вопрос, сделаем шаг назад и посмотрим, из каких частей состоит приложение.

Приложение состоит из одной активности и одного макета

Когда мы строили приложение, мы сообщили Android Studio его конфигурацию, а мастер сделал все остальное: сгенерировал активность и макет по умолчанию.

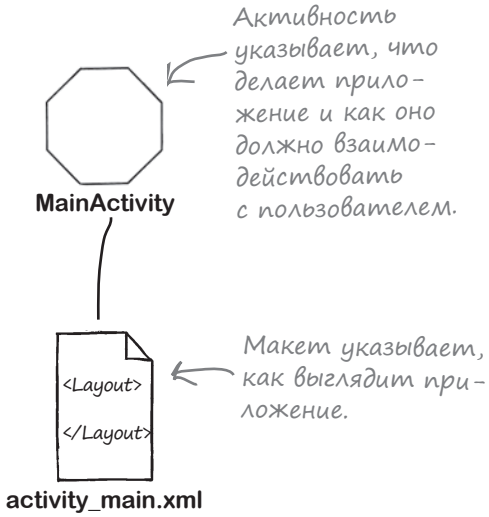
Активность управляет тем, что делает приложение

Android Studio создает за нас активность с именем *MainActivity.kt*. Активность определяет, что **делает** приложение и как оно должно реагировать на действия пользователя.

Макет управляет внешним видом приложения

MainActivity.kt использует макет с именем *activity_main.xml*, который также был создан средой Android Studio. Макет определяет **внешний вид** приложения.

Мы собираемся изменить внешний вид приложения — обновить выводимый текст. Для этого нужно изменить файл, управляющий внешним видом приложения, а следовательно, нужно поближе познакомиться с *макетом*.

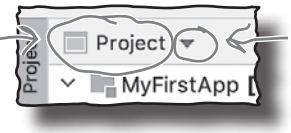


Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Что содержит макет?

Мы хотим изменить текст «Hello World!», сгенерированный Android Studio, поэтому начнем с файла макета *activity_main.xml*. Откройте этот файл (если он еще не был открыт ранее). Для этого найдите файл в папке *app/src/main/res/layout* на панели и сделайте на нем двойной щелчок.

Если вы не видите структуру папок на панели, попробуйте переключиться в режим Project.



Щелкните на этой стрелке, чтобы изменить режим отображения файлов и папок.

Визуальный редактор

Как вы узнали ранее, существуют два режима просмотра и редактирования файлов макетов в Android Studio: **визуальный редактор** и **редактор кода**.

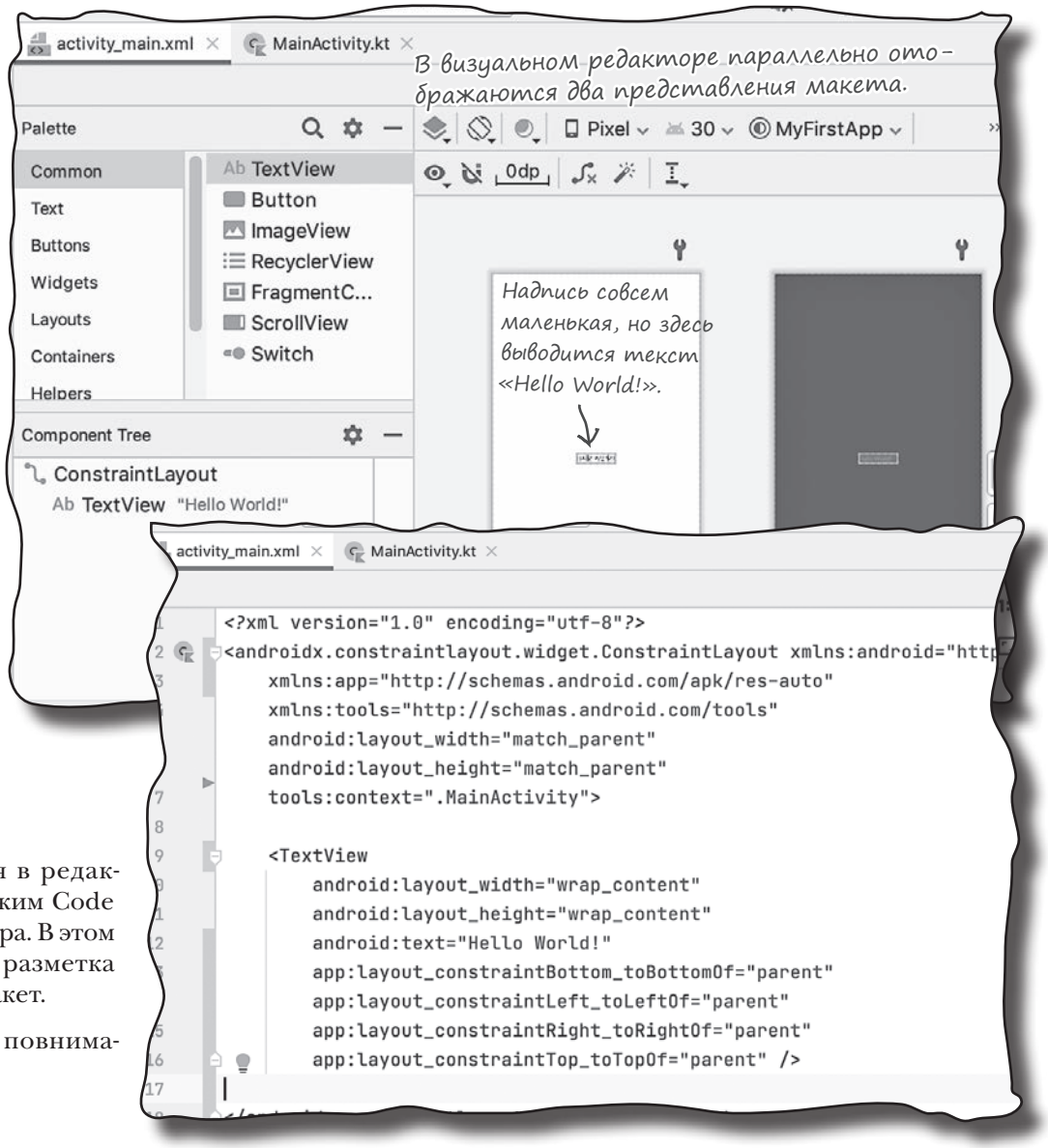
При выборе визуального редактора мы видим, что в макете выводится текст «Hello World!», как и следовало ожидать. Но как выглядит разметка XML, определяющая макет?

Чтобы увидеть ее, нужно переключиться в редактор кода.

Редактор кода

Чтобы переключиться в редактор кода, выберите режим Code в верхней части редактора. В этом режиме отображается разметка XML, определяющая макет.

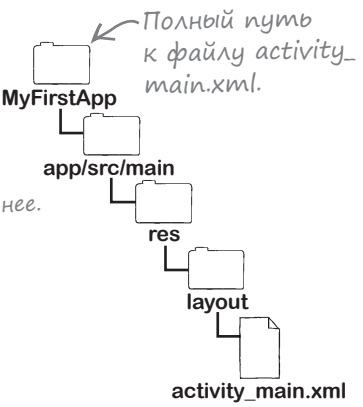
Присмотримся к коду повнимательнее.



activity_main.xml состоит из двух элементов

Ниже приведен код из файла *activity_main.xml*, который был сгенерирован средой Android Studio за нас. В нем опущены подробности, на которые вам пока не нужно отвлекаться; они еще будут рассмотрены в других главах книги.

Разметка выглядит так:



Этот элемент определяет, как должны отображаться компоненты (в данном случае текст «Hello World!»).

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ... >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        ... />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Здесь Android Studio генерирует дальнейшую разметку XML, но пока мы не будем отвлекаться на нее.

Элемент <TextView>.

Часть разметки XML <TextView> также опущена.

Как видно из листинга, разметка состоит из двух элементов.

Первый элемент <...ConstraintLayout> сообщает Android, как отображать компоненты на экране устройства. Существуют различные варианты макетов, которые вы можете использовать в своих приложениях; больше о них вы узнаете позднее в этой книге.

Самым важным на данный момент является второй элемент <TextView>. Этот элемент используется для вывода текста, в данном случае текста «Hello World!».

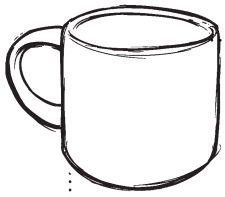
Ключевая часть кода в элементе <TextView> — строка, начинающаяся с `android:text`. Это атрибут `text`, который описывает выводимый текст:

Элемент <TextView> описывает текст в макете.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    ... />
```

Выводимый текст.

Мы заменим этот текст после того, как вы проверите свои силы на следующем упражнении.



Расслабьтесь

Если ваша разметка отличается от нашей, не беспокойтесь.

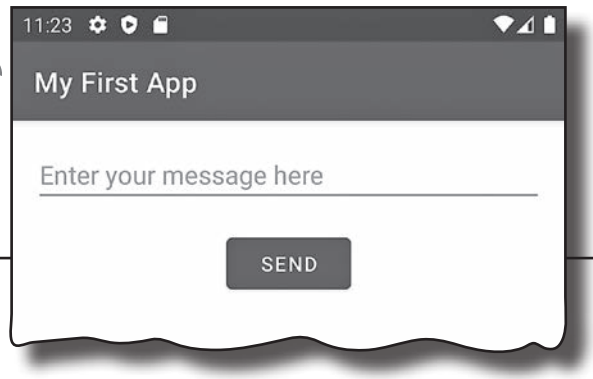
Android Studio может генерировать слегка отличающуюся разметку XML в зависимости от используемой версии. Вам не нужно об этом беспокоиться, потому что, начиная со следующей главы, вы научитесь писать макеты самостоятельно и заменять большую часть той разметки, которую для вас генерирует Android Studio.

Возьмите в руку карандаш

Да, мы еще не приводили большую часть разметки, и все же попробуйте угадать, что делает каждая из строк следующей разметки XML. Мы заполнили несколько пропусков, чтобы вам было проще взяться за дело.

```
<?xml version="1.0" encoding="utf-8"?>          Объявление XML в начале файла.
<LinearLayout    Выстраивает компоненты в линию, один за другим.
    xmlns:android="http://schemas.android.com/apk/res/android"  Определяет пространство имен.
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:hint="Enter your message here"
        android:inputType="text" />

    <Button
        android:id="@+id/send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="SEND" />
</LinearLayout>
```

Вот что появится на экране при запуске приложения с макетом, приведенным выше.

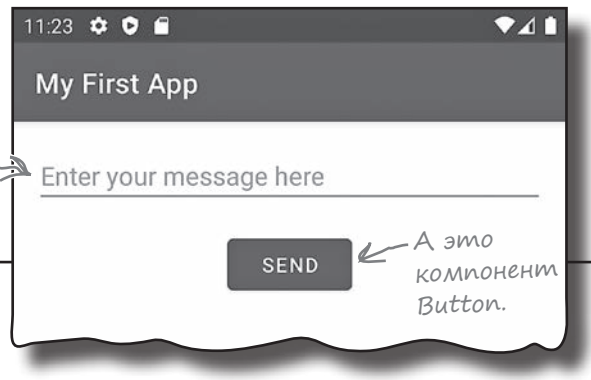


Возьмите в руку карандаш

Решение

Да, мы еще не приводили большую часть разметки, и все же попробуйте угадать, что делает каждая из строк следующей разметки XML. Мы заполнили несколько пропусков, чтобы вам было проще взяться за дело.

```
<?xml version="1.0" encoding="utf-8"?>          Объявление XML в начале файла.
<LinearLayout    Выстраивает компоненты в линию, один за другим.
    xmlns:android="http://schemas.android.com/apk/res/android"  Определяет пространство имен.
    xmlns:tools="http://schemas.android.com/tools"  Определяет второе пространство имен.
    android:layout_width="match_parent"  Приводит ширину в соответствие с размером экрана.
    android:layout_height="match_parent"  Приводит высоту в соответствие с размером экрана.
    android:orientation="vertical"       Выстраивает компоненты по вертикали.
    tools:context=".MainActivity">  Сообщает, что макет используется активностью MainActivity.

    <EditText  Определяет компонент EditText (текстовое поле с возможностью редактирования).
        android:id="@+id/message"       Присваивает идентификатор «message».
        android:layout_width="match_parent"  Приводит в соответствие с шириной макета.
        android:layout_height="wrap_content"     Выделяет место по высоте содержимого.
        android:layout_margin="16dp"     Назначает компоненту поля с размером 16dp.
        android:hint="Enter your message here"     Включает текст подсказки.
        android:inputType="text" />     Использует тип ввода с клавиатуры.

    <Button    Определяет компонент Button.
        android:id="@+id/send"      Присваивает идентификатор «send».
        android:layout_width="wrap_content"     Выделяет место по ширине содержимого.
        android:layout_height="wrap_content"     Выделяет место по высоте содержимого.
        android:layout_gravity="center_horizontal"  Выравнивает кнопку горизонтально по центру.
        android:text="SEND" />      Добавляет к кнопке текст «SEND».
</LinearLayout>    Закрывает элемент LinearLayout.
```



Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

Обновление текста в макете

Наша задача — изменить текст в файле *activity_main.xml*, чтобы при запуске приложения вместо «Hello World!» выводился другой текст. Для этого нужно изменить атрибут `text` в элементе `<TextView>` макета:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    ... />
```

Чтобы изменить текст, обновите этот атрибут.

Атрибут `text` определяется внутри элемента `<TextView>` кодом `android:text`. Он определяет, какой текст должен выводиться, — в данном случае «Hello World!»

Выводит текст... *...«Hello World!»*

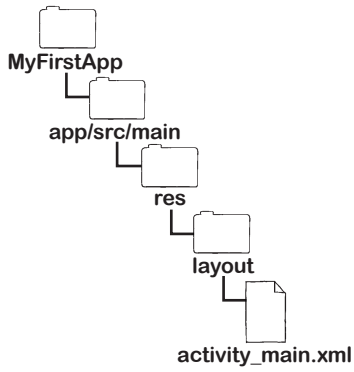
```
android:text="Hello World!" />
```

Чтобы изменить текст, выводимый в макете, просто измените значение атрибута `text` `"Hello World!"` каким-то другим текстом, например `"Pow!"`. Обновленный код элемента `<TextView>` должен выглядеть так:

Часть кода пропущена, так как мы всего лишь изменяем выводимый текст.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ... >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="~~Hello World!~~Pow!"
        ... />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Замените «Hello World!» на «Pow!»



Это единственное изменение, которое необходимо внести для обновления текста. Давайте посмотрим, что произойдет при выполнении кода.

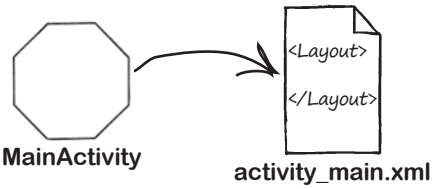
Что делает код

Подготовка среды
Построение приложения
Запуск приложения
Изменение приложения

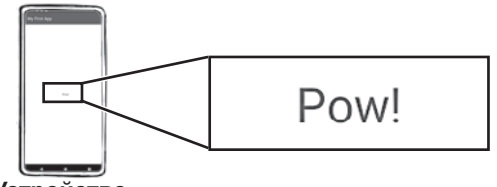
Прежде чем опробовать приложение, еще раз пройдемся по тому, что делает код.

1. **Android использует MainActivity.kt для создания объекта активности MainActivity.**

2. **MainActivity указывает, что активность использует макет activity_main.xml.**



3. **Макет выводит текст «Pow!» в центре приложения на устройстве.**



Часто Задаваемые Вопросы

В: Мой код макета отличается от вашего. Это нормально?

О: Да, это нормально. Android Studio может сгенерировать несколько различающийся код, если вы используете другую версию, но это роли не играет. В дальнейшем вы научитесь создавать собственный код макета и будете заменять большую часть кода, который вам предоставляет Android Studio.

В: Я правильно понимаю, что мы жестко фиксируем выводимый текст?

О: Да, но только чтобы вы увидели, как изменять текст в макете. Существует более эффективный способ вывода текстовых значений, чем запись их прямо в макете, но чтобы узнать его, вам придется дождаться следующей главы.

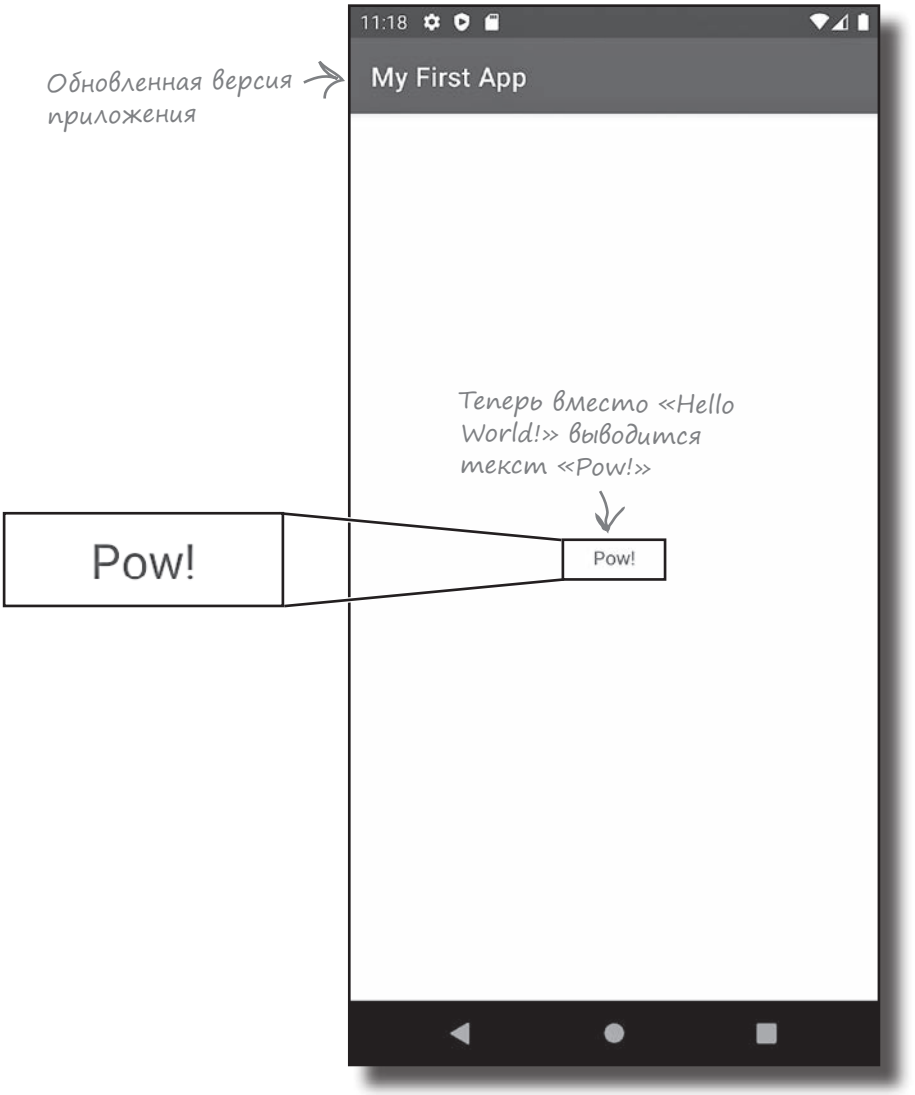
В: Структура папок на моей панели Explorer отличается от вашей. Почему?

О: Android Studio позволяет выбрать альтернативные представления для отображения иерархии папок; по умолчанию используется представление Android. Мы предпочитаем представление Project, так как оно отражает структуру папок. Чтобы переключить панель в представление Project, щелкните на стрелке в верхней части панели Explorer и выберите вариант Project.

Используйте эту стрелку для выбора представления (режима) панели Explorer. Обычно используется представление Project.

Тест-драйв

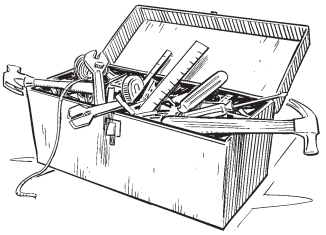
После того как файл будет отредактирован, попробуйте снова запустить файл в эмуляторе — выберите команду «Run 'app'» меню Run или щелкните на кнопке Run. Теперь в приложении вместо текста «Hello World!» должен выводиться текст «Pow!».



Поздравляем! Вы построили и обновили свое первое приложение, а попутно узнали, как строятся приложения Android. В следующей главе мы пойдем дальше и создадим приложение, с которым вы сможете взаимодействовать.

ГЛАВА 1

Ваш инструментарий Android



Глава 1 подходит к концу. В ней ваш инструментарий дополнился основными концепциями программирования для Android.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Версии Android характеризуются номером версии, уровнем API и кодовым именем.
- Android Studio — специализированная версия среды IntelliJ IDEA. Она взаимодействует с пакетом Android Software Development Kit (SDK) и системой сборки Gradle.
- Типичное Android-приложение состоит из активностей, макетов и файлов ресурсов.
- Макеты описывают внешний вид приложения. Они хранятся в папке *app/src/main/res/layout*.
- Активности описывают то, что делает приложение и как оно взаимодействует с пользователем. Созданные вами активности хранятся в папке *app/src/main/java*.
- Файл *AndroidManifest.xml* содержит информацию о самом приложении. Этот файл находится в папке *app/src/main*.
- AVD — виртуальное устройство Android (Android Virtual Device). AVD выполняется в эмуляторе Android и моделирует физическое устройство Android.
- APK — пакет приложения Android. Файл содержит байт-код приложения, библиотеки и ресурсы. Установка приложения на устройстве осуществляется установкой его пакета APK.
- Приложения Android выполняются в отдельных процессах с использованием исполнительной среды Android (ART).
- Элемент `<TextView>` используется для вывода текста.
- Атрибут `text` элемента `<TextView>` определяет текст, который должен выводиться в элементе.

Приложения, которые что-то делают

Обычно приложение должно реагировать на действия пользователя. В этой главе вы узнаете, как существенно повысить **интерактивность** ваших приложений. Вы узнаете, как добавить в код активности метод ***OnClickListener***, чтобы приложение могло **прослушивать** действия пользователя и соответствующим образом на них реагировать. Также вы научитесь **конструировать макеты** и поймете, как каждый UI-компонент, добавляемый в макет, происходит от общего **предка View**. Попутно вы узнаете, почему **строковые ресурсы** настолько важны для гибких, хорошо спроектированных приложений.

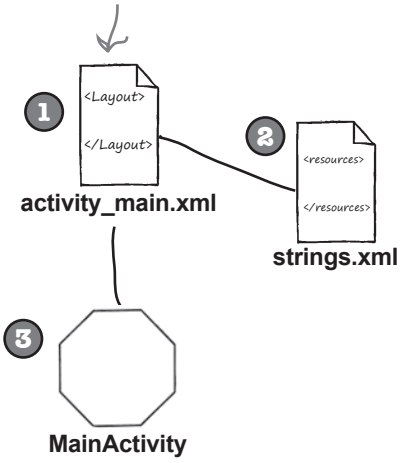
В этой главе мы построим приложение для выбора пива

Когда вы строите приложение Android, обычно предполагается, что это приложение должно что-то *делать*.

В этой главе мы покажем, как создать приложение, взаимодействующее с пользователем. В приложении Beer Adviser пользователь выбирает вид пива, который он предпочитает, щелкает на кнопке и получает список рекомендуемых сортов.

Так должен выглядеть макет.

Приложение имеет следующую структуру:



1. **Макет activity_main.xml определяет, как будет выглядеть приложение.**
   Он состоит из трех компонентов графического интерфейса:
   - раскрывающегося списка значений, в котором пользователь выбирает нужный вид пива;
   - кнопки, которая при нажатии возвращает подборку сортов пива;
   - надписи для вывода сортов пива.

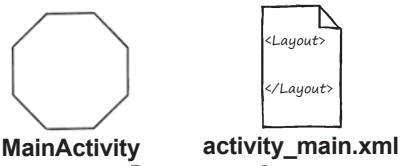
2. **Файл strings.xml включает все строковые ресурсы, необходимые макету,** например текст надписи на кнопке и виды пива.

3. **Активность MainActivity определяет, как приложение должно взаимодействовать с пользователем.**
   Она получает вид пива, выбранный пользователем, и использует его для вывода списка сортов, которые могут представлять интерес для пользователя.

Что нужно сделать

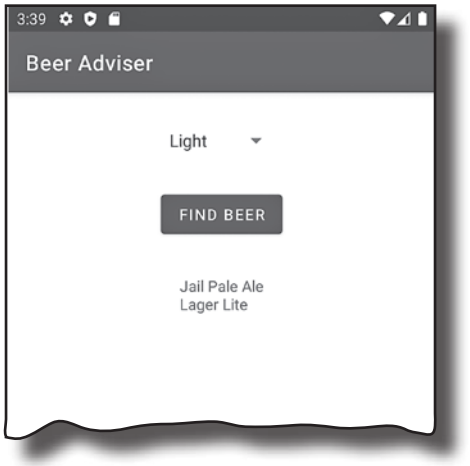
Итак, приступим к построению приложения Beer Adviser. Работа состоит из нескольких шагов (все они будут подробно рассмотрены в этой главе):

1. **Создание проекта.**
Мы создаем совершенно новое приложение, для которого нужно будет создать новый проект, включающий пустую активность и макет.

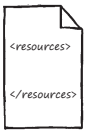


Android Studio автоматически добавляет эти компоненты при создании нового проекта.

2. **Обновление макета.**
Когда основная структура приложения будет готова, следует отредактировать макет и включить в него все компоненты графического интерфейса, необходимые для работы приложения.



3. **Добавление строковых ресурсов.**
Весь жестко запрограммированный текст заменяется строковыми ресурсами, чтобы весь текст, используемый приложением, содержался в одном файле.



4. **Программирование реакции кнопки на щелчки.**
Макет определяет только внешний вид приложения. Чтобы кнопка что-то *делала* по щелчку, необходимо написать код активности.

5. **Программирование логики приложения.**
Мы добавим в приложение новый метод, который будет возвращать пользователю правильные сорта пива в зависимости от его выбора.

Для шагов 4 и 5 необходимо обновить MainActivity.

Начнем с создания проекта.

Создание проекта

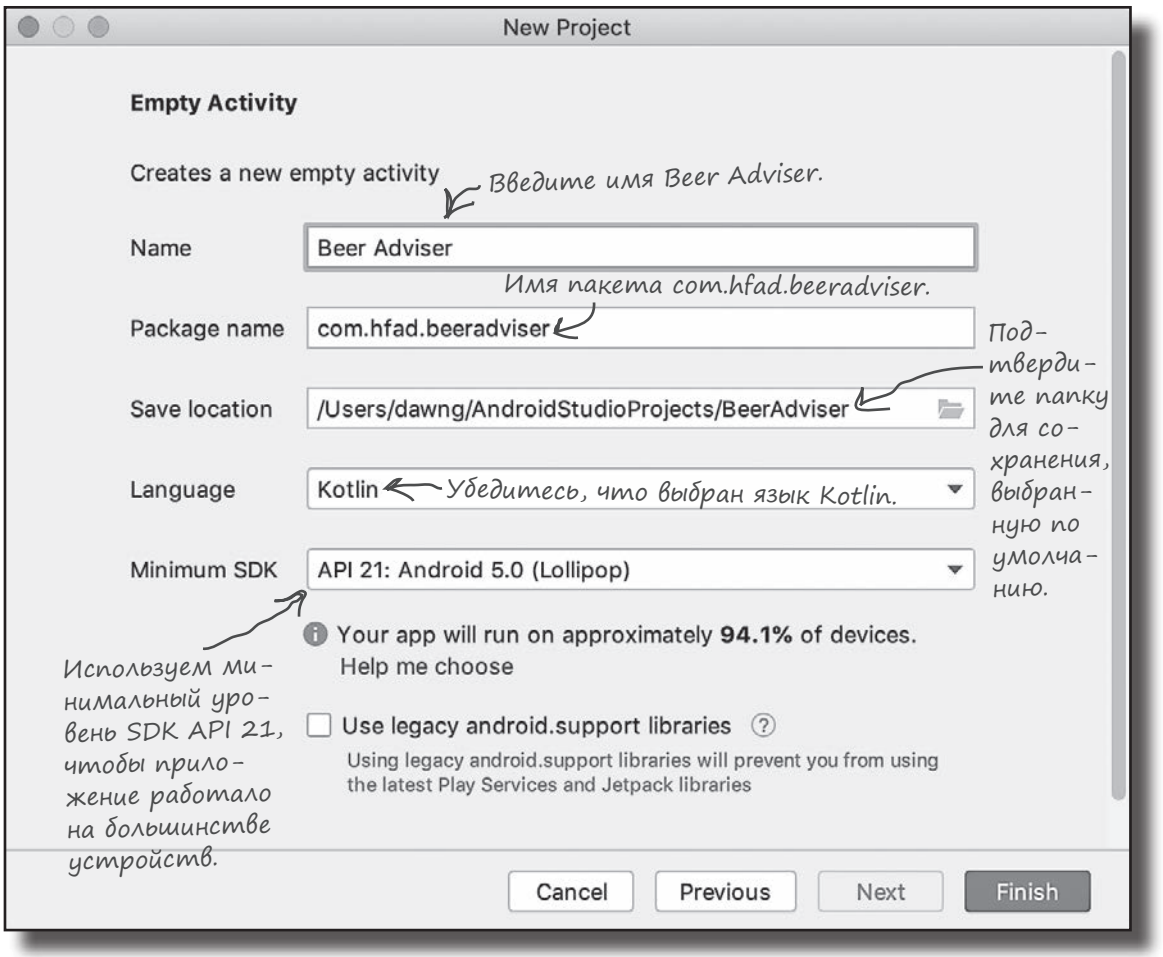
Создание проекта
Обновление макета
Добавление ресурсов
Реакция на щелчки
Программирование логики

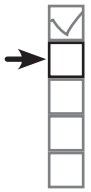
Последовательность действий для создания нового проекта почти не отличается от той, которая использовалась в предыдущей главе:

1. Откройте Android Studio, закройте все открытые проекты и выберите на заставке строку «New Project». Запускается мастер, уже знакомый вам по главе 1.

2. Проследите за тем, чтобы флажок Phone and Tablet был установлен. Выберите вариант Empty Activity.

3. Введите имя приложения «Beer Adviser» и имя пакета «com.hfad.beeradviser», подтвердите место сохранения по умолчанию. Убедитесь в том, что для приложения выбран язык Kotlin, а для минимального уровня SDK выбрано значение API 21, чтобы приложение работало на большинстве устройств Android. Щелкните на кнопке Finish.





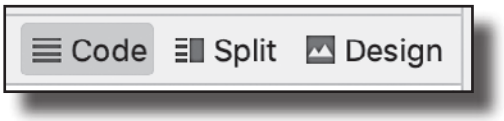
- Создание проекта
- **Обновление макета**
- Добавление ресурсов
- Реакция на щелчки
- Программирование логики

Мы создали макет и активность по умолчанию

Когда вы щелкнете на кнопке Finish, среда Android Studio создаст новый проект, содержащий активность *MainActivity.kt* и макет *activity_main.xml* (как и при создании проекта в главе 1). Необходимо изменить эти файлы, чтобы придать приложению требуемый внешний вид и поведение.

Начнем с редактирования файла макета *activity_main.xml* для изменения внешнего вида приложения. Макет будет построен на нескольких ближайших страницах, а пока переключитесь на представление Project панели Explorer в Android Studio, перейдите в папку *app/src/main/res/layout* и откройте файл *activity_main.xml*. Затем переключитесь в редактор кода и **замените весь код** из *activity_main.xml* следующим кодом:

Щелкните на вкладке Code, чтобы открыть редактор кода.

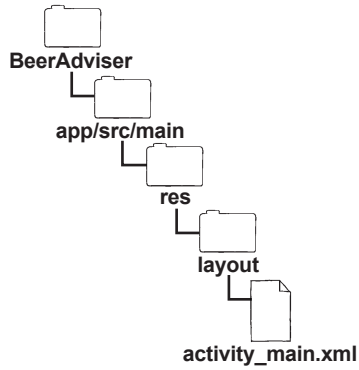


Мы заменили код, который был сгенерирован средой Android Studio.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Beer types" />
</LinearLayout>
```

Эти элементы относятся к макету в целом. Они определяют его ширину и высоту, а также величину отступов от краев макета и способ размещения (по вертикали или по горизонтали).

Используется для вывода текста.



Этот код определяет линейный макет (он задается элементом <LinearLayout>) и текст (элемент <TextView>). Позднее в этой главе вы больше узнаете об этих элементах, а пока достаточно знать, что линейный макет используется для размещения компонентов графического интерфейса в столбец, а в надписи выводится текст «Beer types».

Все изменения, вносимые в разметку XML макета, отражаются в визуальном редакторе. Переключитесь на него, щелкнув на вкладке Design в верхней части панели редактора.

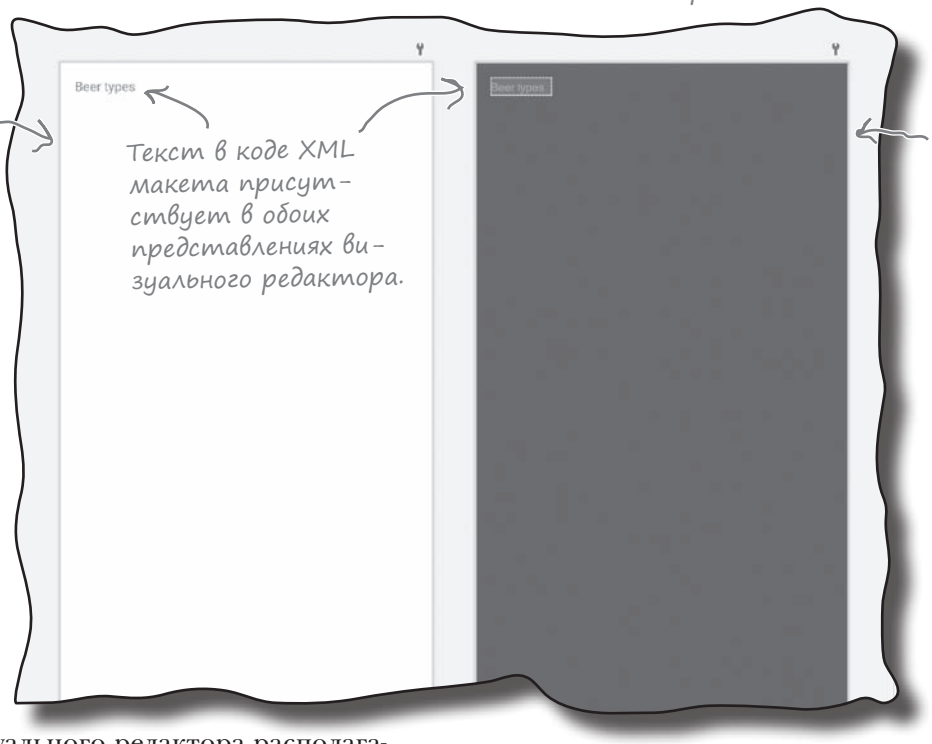
Щелкните на вкладке Design, чтобы открыть визуальный редактор.

- [x] Создание проекта
- [ ] **Обновление макета**
- [ ] Добавление ресурсов
- [ ] Реакция на щелчки
- [ ] Программирование логики

Подробнее о визуальном редакторе

Как вы узнали в главе 1, визуальный редактор предоставляет средства редактирования макетов более наглядные, чем при прямом редактировании разметки XML. В нем имеется два разных представления структуры макета. Одно показывает, как макет будет выглядеть на реальном устройстве, а другое — эскиз его структуры:

Если в Android Studio не отображаются оба представления макета, щелкните на этой кнопке панели инструментов визуального редактора и выберите вариант «Design and Blueprint».

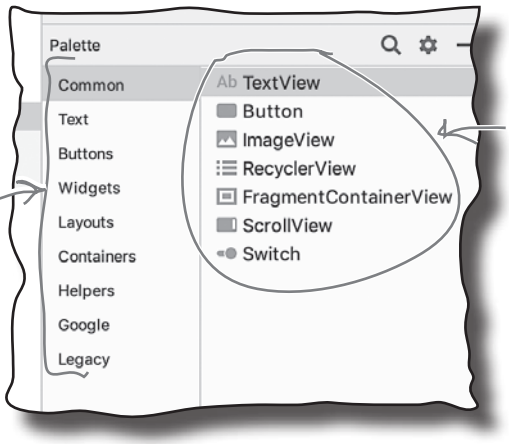


Это представление макета показывает, как он будет выглядеть на реальном устройстве.

Текст в коде XML макета присутствует в обоих представлениях визуального редактора.

Представление Blueprint ориентировано на структуру макета.

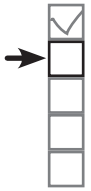
Слева от визуального редактора располагается палитра с компонентами графического интерфейса, которые можно перетаскивать мышью на макет. Мы воспользуемся ею на следующей странице для добавления в макет кнопки, которая позднее будет использоваться для обновления текста, выводимого приложением.



В этом списке содержатся разные категории компонентов, которые можно добавить в макет.

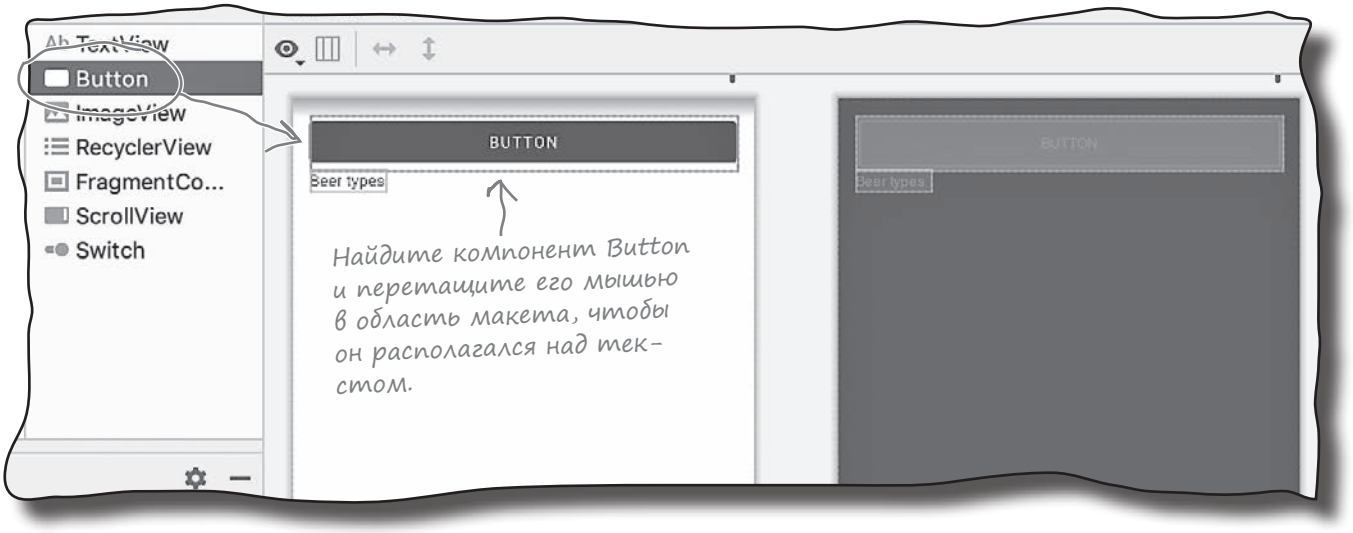
Компоненты выбранной категории. Они будут более подробно описаны позднее в книге.

Добавление кнопки в визуальном редакторе



- Создание проекта
- **Обновление макета**
- Добавление ресурсов
- Реакция на щелчки
- Программирование логики

Начнем с добавления кнопки в макет. Найдите компонент Button в палитре, щелкните на нем и перетащите в область визуального редактора, чтобы он располагался над текстом. Кнопка появляется в изображении макета:



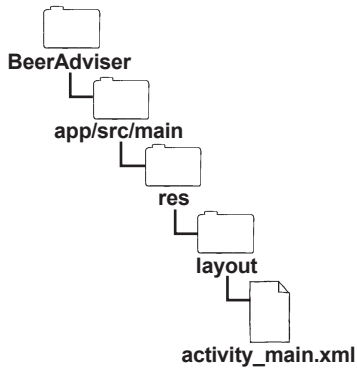
Изменения, внесенные в визуальном редакторе, отражаются в XML

Такое перетаскивание компонентов графического интерфейса является удобным способом обновления макета. Переключившись на редактор кода, вы увидите, что в результате добавления кнопки в визуальном редакторе в файле XML появилось несколько строк кода:

Код, добавленный визуальным редактором, зависит от того, в каком месте была размещена кнопка. Не беспокойтесь, если ваш код отличается от нашего.

Новый элемент <Button>, описывающий новую кнопку, которую вы перетащили в макет. На нескольких ближайших страницах он будет рассмотрен более подробно.

```
...
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button" />

<TextView
    android:id="@+id/brands"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Beer types" />
...
```



- Создание проекта
- **Обновление макета**
- Добавление ресурсов
- Реакция на щелчки
- Программирование логики

В activity_main.xml появилась новая кнопка

Как вы уже видели, визуальный редактор добавил новый элемент `<Button>` в файл *activity_main.xml*:

```
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button" />
```

В мире Android пользователь нажимает кнопку, чтобы инициировать какое-либо действие. Элемент `<Button>` включает атрибуты, управляющие ее размером и внешним видом. Эти атрибуты существуют не только у кнопок — они есть и у других компонентов графического интерфейса, включая надписи.

Кнопки и надписи — субклассы одного класса Android View

Тот факт, что кнопки и надписи имеют так много общих свойств, вполне логичен: оба компонента наследуют от одного класса Android **`View`**. Более подробные описания свойств будут приведены ниже, а пока рассмотрим несколько типичных примеров.

android:id

Имя, по которому идентифицируется компонент. Это имя используется кодом активности для обращения к компоненту и управления его поведением:

```
android:id="@+id/button"
```

android:layout_width, android:layout_height

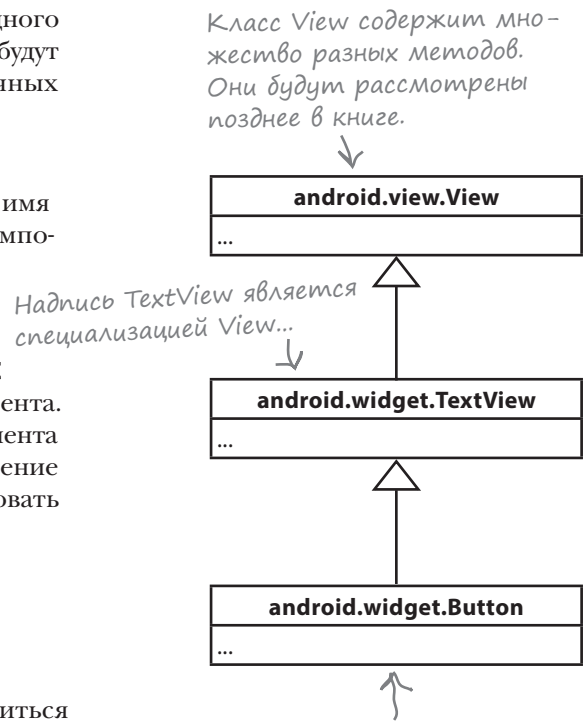
Эти атрибуты задают базовую ширину и высоту компонента. Значение «wrap_content» означает, что размеры компонента должны подбираться по размерам содержимого, а значение «`match_parent`» — что его ширина должна соответствовать ширине макета, который его содержит:

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

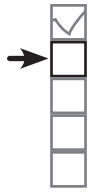
android:text

Атрибут сообщает Android, какой текст должен выводиться в компоненте, например, текст на кнопке:

```
android:text="Button"
```



Подробнее о коде макета



Давайте внимательнее рассмотрим код макета и разобьем его так, чтобы происходящее стало более понятным (не беспокойтесь, если ваш код выглядит немного иначе, просто следите за логикой):

Элемент <LinearLayout>

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">
```

Кнопка

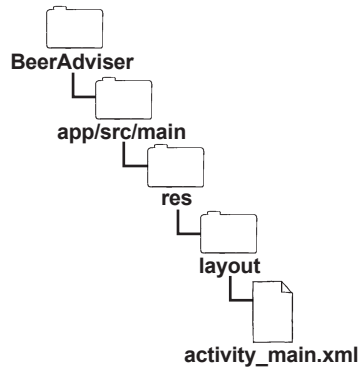
```
    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button" />
```

Надпись

```
    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Beer types" />
```

```
</LinearLayout>
```

Закрывает элемент <LinearLayout>.



Элемент <LinearLayout>

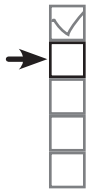
Код начинается с элемента `<LinearLayout>`. Элемент сообщает Android, что компоненты графического интерфейса в макете должны следовать один за другим в строку или столбец.

Ориентация задается атрибутом `android:orientation`. В нашем примере:

```
android:orientation="vertical"
```

компоненты выводятся в один столбец.

Подробнее о коде макета (продолжение)



- Создание проекта
- **Обновление макета**
- Добавление ресурсов
- Реакция на щелчки
- Программирование логики

Элемент <LinearLayout> (с предыдущей страницы) содержит еще два элемента: <Button> и <TextView>.

Элемент <Button>

На первом месте стоит элемент <Button>:

```
...
<Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button" />
...
```

Использование LinearLayout означает, что компоненты интерфейса отображаются по вертикали в столбец или по горизонтали в строку.

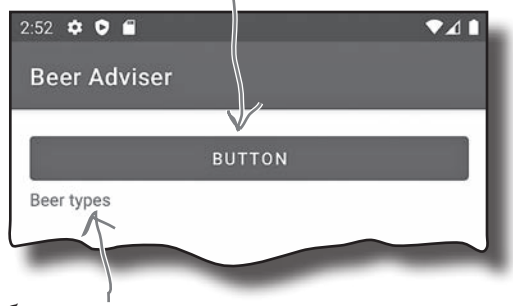
Так как это первый элемент внутри <LinearLayout>, он первым выводится в макете у верхнего края экрана. Его атрибут `layout_width` содержит значение `"match_parent"`, с которым его ширина выбирается по ширине родительского элемента, то есть <LinearLayout>. Атрибут `layout_height` содержит значение `"wrap_content"`, это означает, что элемент должен иметь минимальную высоту, необходимую для вывода его текста.

Элемент <TextView>

Внутри элемента <LinearLayout> последним идет элемент надписи <TextView>:

```
...
<TextView
    android:id="@+id/brands"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Beer types" />
...
```

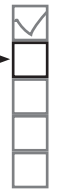
Кнопка выводится наверху, потому что она является первым элементом в XML.



Надпись выводится под кнопкой, так как она следует за кнопкой в <LinearLayout>.

Так как это второй элемент, а для элемента <LinearLayout> выбрана вертикальная ориентация, он выводится под кнопкой (первым элементом). Его атрибутам `layout_width` и `layout_height` присвоено значение `"wrap_content"`, чтобы элемент занимал минимальное место, необходимое для вывода его текста.

Вы уже видели, как добавление компонентов в визуальном редакторе приводит к их включению в разметку XML макета. Также справедливо и обратное: все изменения, вносимые в разметке XML макета, отражаются в визуальном редакторе. Посмотрим, как работает этот механизм.

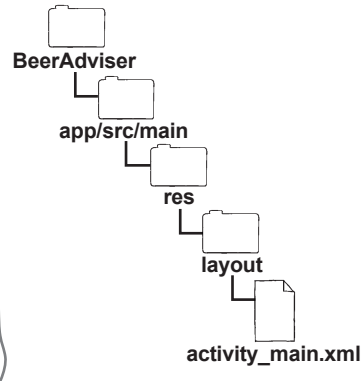


Обновление разметки XML макета

Давайте обновим макет: мы добавим в него новый компонент **Spinner** и настроим уже существующие компоненты надписи и кнопки. Компонент Spinner представляет раскрывающийся список значений. Если щелкнуть на нем, он раскроется и выведет список, из которого вы сможете выбрать одно значение.

Внесите в файл *activity_main.xml* следующие изменения (выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <Spinner
        android:id="@+id/beer_color"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp" />

    <Button
        android:id="@+id/button find_beer"
        android:layout_width="match_parent wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="Button Find Beer" />

    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="Beer types" />
</LinearLayout>
```



Новый компонент Spinner позволяет выбрать одно значение из набора вариантов.

Замените идентификатор кнопки значением «find_beer». Новое значение будет использоваться позднее в этой главе.

Подгоняет ширину кнопки по ширине содержимого.

Кнопка выравнивается горизонтально по центру и снабжается отступами.

Изменяет текст кнопки.

Надпись выравнивается по центру и снабжается отступами.

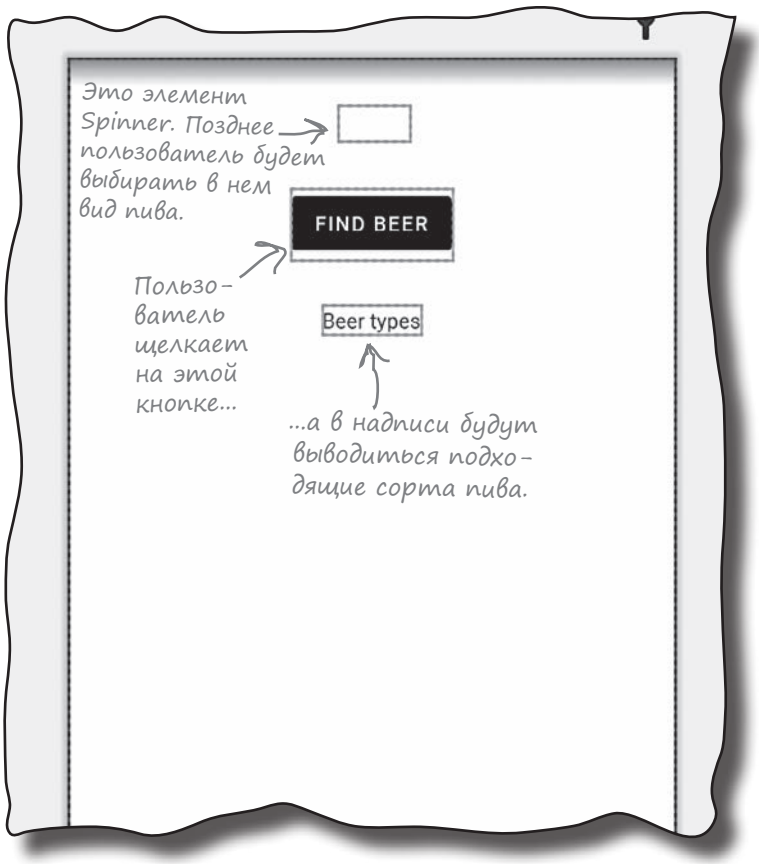
Не забудьте!

Обязательно внесите в содержимое activity_main.xml эти изменения.

- [x] Создание проекта
- [ ] **Обновление макета**
- [ ] Добавление ресурсов
- [ ] Реакция на щелчки
- [ ] Программирование логики

Изменения XML отражаются в Визуальном редакторе

После изменения XML макета переключитесь в визуальный редактор. Вместо макета с кнопкой и расположенной под ней надписью визуальный редактор теперь отображает список Spinner, кнопку и надпись, выровненные горизонтально по центру в один столбец:



Все компоненты, необходимые для макета приложения Beer Adviser, были включены в файл *activity_main.xml*. Нам все еще предстоит немало потрудиться, но давайте опробуем приложение на практике и посмотрим, как оно выглядит на устройстве.

Элемент Spinner предоставляет раскрывающийся список значений. Он позволяет выбрать одно значение из заданного набора.

Все компоненты графического интерфейса, добавляемые в файлы макетов, — кнопки, надписи и т. д. — используют похожие наборы атрибутов, потому что все они являются разновидностями View. Во внутренней реализации все они наследуют от одного класса Android View.

Эти компоненты часто называются представлениями (views), потому что они наследуют от одного класса View.

Тест-драйв

- [x] Создание проекта
- [x] **Обновление макета**
- [ ] Добавление ресурсов
- [ ] Реакция на щелчки
- [ ] Программирование логики

Запустите приложение командой «Run 'app'» меню Run или кнопкой Run. Подождите, пока приложение загрузится.

Когда приложение появится на устройстве, вы увидите пустой раскрывающийся список, кнопку и надпись, выстроенные в один столбец.

Часто Задаваемые Вопросы

В: Вы сказали, что представления можно добавлять как в визуальном редакторе, так и прямым редактированием XML. Какой вариант лучше?

О: В основном это вопрос личных предпочтений. Для простых макетов лучше выбрать обновление XML, но в главе 4 будет показано, что некоторые типы макетов проще обновлять в визуальном редакторе.

В большинстве случаев мы будем предлагать вам обновлять разметку XML, так как это позволит вам легко убедиться в том, что ваш код совпадает с нашим.

В: Почему мы заменили разметку по умолчанию, предоставленную средой Android Studio?

О: По двум причинам. Во-первых, линейный макет лучше всего подходит для размещения компонентов в одну строку или столбец. Из-за этого он хорошо вписывается в приложение, которое строится в этой главе.

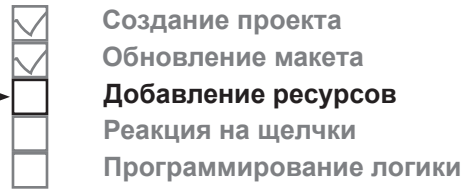
Во-вторых, линейный макет — самый простой из макетов, которые вы научитесь использовать. Позднее в книге будут представлены более сложные виды макетов, а пока мы сосредоточимся на основах.

В: Почему текст на кнопке выводится в верхнем регистре?

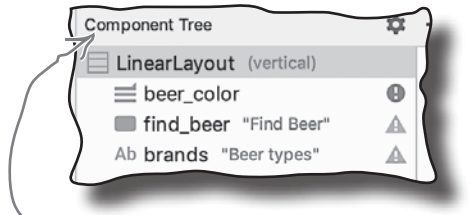
О: Потому что тема приложения по умолчанию преобразует текст кнопки к верхнему регистру. Темы — и возможности переопределения их поведения — более подробно рассматриваются в главе 8.

А пока скажем, что для предотвращения вывода текста кнопки в верхнем регистре можно добавить следующий код к каждому элементу `<Button>` в вашем макете: `android:textAllCaps="false"`

Для макета выдаются предупреждения...



Когда вы разрабатываете макеты в Android Studio, IDE автоматически проверяет ваш код на наличие ошибок и предлагает возможные улучшения. Чтобы просмотреть предупреждения или рекомендации, можно переключить макет в представление визуального редактора, а затем проверить панель дерева компонентов. Эта панель обычно располагается под палитрой, и на ней выводится иерархическое дерево компонентов в макете. Если у Android Studio есть рекомендации по улучшению вашего кода, справа от соответствующего компонента появляется значок (ярлык). Например, значки предупреждений выводятся рядом с компонентами `find_beer` и `brands`. Если навести указатель мыши на каждый из них, появляются сообщения, предупреждающие о жестко запрограммированном тексте:



Дерево компонентов располагается под палитрой в визуальном конструкторе макета.

Также выводится предупреждение о раскрывающемся списке, но мы займемся им позднее.

Одно из предупреждений.

...потому что он содержит жестко запрограммированный текст.

При определении макета мы жестко запрограммировали текст, который должен выводиться в надписи и на кнопке. Для этого использовался код следующего вида:

```
android:text="Find Beer"

android:text="Beer types"
```

Два компонента в макете используют жестко запрограммированные текстовые значения.

Хотя жесткое программирование текста в макете нормально работает, пока вы только учитесь, вряд ли это лучший вариант.

Допустим, вы написали приложение, которое произвело фурор в вашем локальном магазине Google Play Store. Вы не хотите ограничиваться одной страной или языком — приложение должно стать доступным для пользователей всего мира и поддерживать разные языки. Но если весь текст жестко запрограммирован в файлах макетов, вывести приложение на глобальный уровень будет непросто.

Также заметно усложняется внесение в текст изменений уровня приложения. Представьте, что начальник просит вас изменить текст приложения, потому что название компании изменилось. Если весь текст жестко зафиксирован в макетах, вам придется редактировать множество файлов.

Нет ли другого пути?

Размещение текста в файле строковых ресурсов

Лучше разместить текстовые значения в **файле строковых ресурсов**. Это значительно упрощает внесение глобальных изменений в текст, используемый в приложении. Вместо того чтобы изменять жестко запрограммированные текстовые значения в множестве файлов активностей и макетов, достаточно отредактировать текст в файле ресурсов.

Такой подход также значительно упрощает локализацию приложения. Вместо того чтобы фиксировать текст на одном языке, можно предоставить разные файлы строковых ресурсов для каждого языка, который вы хотите поддерживать. Это позволяет приложению менять язык, используемый приложением, чтобы он соответствовал локальному контексту устройства.

Android Studio помогает извлекать строковые ресурсы

Если ваш макет содержит жестко запрограммированный текст, Android Studio предоставляет простой способ извлечения текста и добавления его в файл строковых ресурсов. Просто щелкните (или сделайте двойной щелчок) на каждом значке предупреждения о жестко запрограммированном тексте, после чего щелкните на кнопке Fix, чтобы решить проблему.

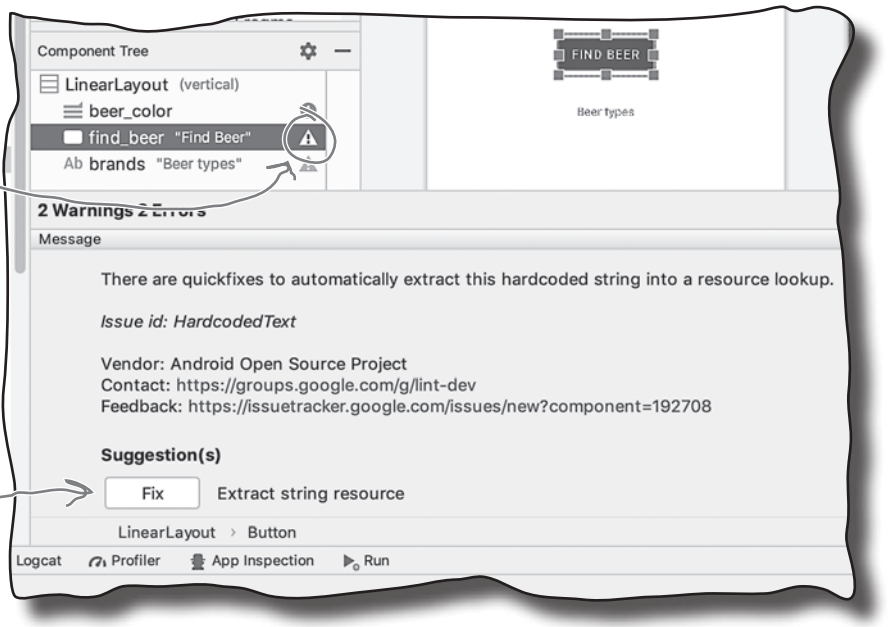
Возможно, содержимое окна придется прокрутить вниз, чтобы увидеть эту кнопку.

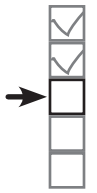
Опробуем этот способ с одним из компонентов макета. Убедитесь в том, что вы используете режим визуального редактора *activity_main.xml*, и щелкните на значке предупреждения рядом с компонентом `find_beer`.

Вы получите объяснение того, почему жестко запрограммированный текст создает проблемы. Прокрутите описание до конца и щелкните на кнопке Fix:

Щелкните на значке предупреждения рядом с компонентом find_beer...

...затем щелкните на кнопке Fix в конце пояснительного сообщения.



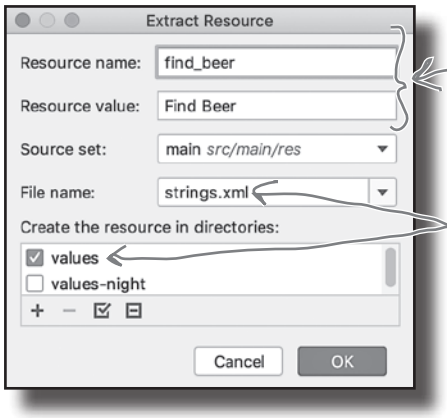


Создание проекта
Обновление макета
Добавление ресурсов
Реакция на щелчки
Программирование логики

Извлечение строкового ресурса

Когда вы щелкаете на кнопке Fix, открывается окно Extract Resource. В нем можно ввести имя ресурса, его значение и имя файла строковых ресурсов. Убедитесь в том, что в окне выбрано имя ресурса «find_beer», имя файла «strings.xml» и исходный набор «main» и выбран каталог values. Затем щелкните на кнопке OK.

Когда вы щелкаете на кнопке OK, Android Studio добавляет жестко запрограммированный текст компонента `find_beer` в файл строковых ресурсов с именем *strings.xml* и изменяет разметку XML макета, чтобы в ней использовался строковый ресурс. Мы рассмотрим оба изменения, начиная с файла строковых ресурсов.

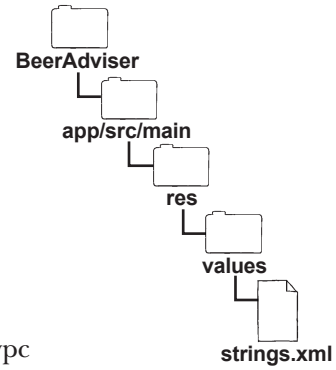


Убедитесь в том, что ресурсу присвоено имя «find_beer» и что он содержит значение «Find Beer».

Ресурс добавляется в файл strings.xml в папке values.

Строковый ресурс был добавлен в файл strings.xml

strings.xml — файл строковых ресурсов приложения по умолчанию. Android Studio автоматически создает этот файл при создании нового проекта. Откройте файл *strings.xml* на панели Explorer среды Android Studio: файл *strings.xml* находится в папке *app/src/main/res/values*.



Содержимое файла должно выглядеть примерно так:

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer</string>
</resources>
```

Приведенный выше код описывает два строковых ресурса, каждый ресурс представлен в виде пары «имя–значение». Первый ресурс с именем `app_name` содержит значение «Beer Adviser», а второй ресурс с именем `find_beer` содержит значение «Find Beer». Второй ресурс был добавлен при извлечении жестко запрограммированного текста для компонента `find_beer`:

Означает, что это строковый ресурс.

```
<string name="find_beer">Find Beer</string>
```

Имя ресурса «find_beer».

И значение Find Beer.

Строковые ресурсы будут более подробно рассмотрены через несколько страниц, а пока посмотрим, какие изменения были внесены в *activity_main.xml*.

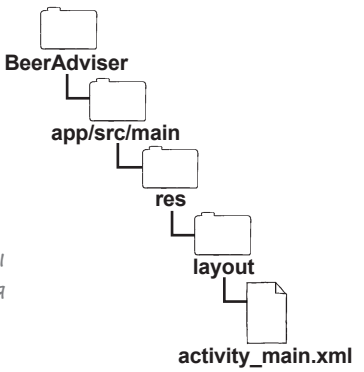
activity_main.xml использует строковый ресурс

Когда мы отдаем команду Android Studio извлечь жестко запрограммированный текст, Android Studio автоматически обновляет кнопку `find_beer` в файле *activity_main.xml*, чтобы в нем использовался извлеченный строковый ресурс.

Ниже приведен обновленный код кнопки:

```
<Button
    android:id="@+id/find_beer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="@string/find_beer" />
```

Атрибут text ранее содержал значение «Find Beer». Теперь он был изменен, чтобы в нем использовался строковый ресурс.



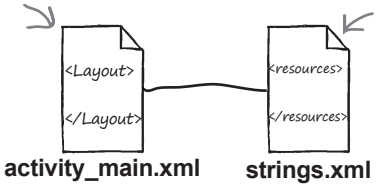
Как вы вскоре увидите, атрибуту `text` кнопки `find_beer` было присвоено значение `"@string/find_beer"`. Что это означает?

Начнем с первой части, `@string`. Она всего лишь приказывает Android искать текстовое значение из файла строковых ресурсов. В данном случае это файл *strings.xml*, который был приведен ранее.

Вторая часть, `find_beer`, приказывает Android **найти значение ресурса с именем `find_beer`**. Таким образом, `"@string/find_beer"` фактически говорит: «Найти строковый ресурс с именем `find_beer` и использовать связанное с ним текстовое значение».

Макет получает текст...

...из файла строковых ресурсов.



Выводимый текст...

...для строкового ресурса find_beer.

```
android:text="@string/find_beer" />
```

Строковые ресурсы также можно извлекать вручную

Вы уже знаете, какие изменения вносит среда Android Studio в код, когда вы отдаете ей команду извлечь жестко закодированный текст в строковый ресурс. Эти изменения также можно внести самостоятельно, напрямую редактируя код в файлах *strings.xml* и *activity_main.xml*.

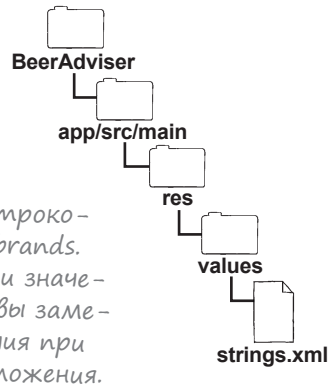
Давайте посмотрим, как это делается. Для примера возьмем жестко закодированный текст «Beer types», используемый атрибутом `text` компонента `brands`. Обновленный компонент должен использовать строковый ресурс.

Скорее всего, в своих проектах вы будете просто пользоваться мастером. Мы показываем, как вручную редактировать XML, чтобы вы были уверены в том, что созданный вами код совпадает с нашим. Для этого проще всего воспользоваться прямым обновлением XML.

Добавление и использование нового строкового ресурса

Начнем с создания нового строкового ресурса с именем `brands`. Откройте файл *strings.xml* (находящийся в папке *app/src/main/res/values*) и добавьте новую строку для строкового ресурса. В следующем листинге она выделена жирным шрифтом:

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer</string>
    <string name="brands">No beer selected</string>
</resources>
```

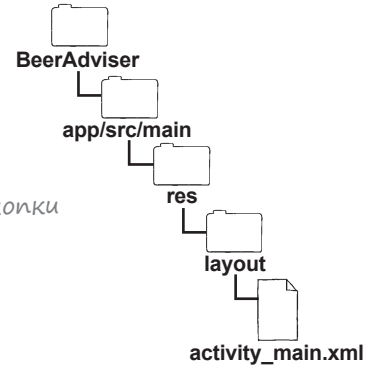


Добавьте строковый ресурс brands. Мы изменили значение, чтобы вы заметили отличия при запуске приложения.

После добавления строкового ресурса откройте файл *activity_main.xml* и обновите код надписи `brands`, чтобы в нем использовался новый ресурс (изменения выделены жирным шрифтом):

```
...
<Spinner
    android:id="@+id/beer_color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp" />

<Button
    android:id="@+id/find_beer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="@string/find_beer" />

<TextView
    android:id="@+id/brands"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="~~Beer types~~@string/brands" />
...
```

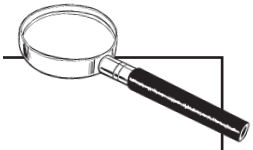
Код списка и кнопки не изменялся.



Обновите атрибут text надписи, чтобы вместо жестко запрограммированного текста в нем использовался строковый ресурс brands. В результате в надписи будет выводиться значение ресурса.

На следующей странице приведена краткая сводка использования строковых ресурсов. Далее мы опробуем приложение на практике.

Строковые ресурсы под увеличительным стеклом



Файл *strings.xml* является файлом ресурсов по умолчанию, используемым для хранения пар строк «имя–значение», к которым можно обращаться из приложения. Файл имеет следующий формат:

Элемент <string> определяет пары «имя-значение» как строки.

Элемент <resources> определяет содержимое файла как набор ресурсов.

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer!</string>
    <string name="brands">No beer selected</string>
</resources>
```

Android распознает *strings.xml* как файл строковых ресурсов по двум признакам:

Файл хранится в папке *app/src/main/res/values*.
Файлы XML, хранящиеся в этой папке, содержат значения, которые могут использоваться в приложении, например строки и цвета.

Файл имеет элемент <resources>, который содержит один или несколько элементов <string>.
Формат самого файла указывает, что это файл ресурсов, содержащий строки. Элемент `<resources>` сообщает Android, что файл содержит ресурсы, а элемент `<string>` идентифицирует каждый строковый ресурс.

Это означает, что файл строковых ресурсов не обязательно называть *strings.xml*; при желании вы можете присвоить ему другое имя или разбить строки на несколько файлов.

Каждая пара «имя–значение» определяется в следующей форме:

```
<string name="string_name">string_value</string>
```

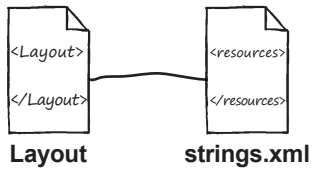
где `string_name` — идентификатор строки, а `string_value` — само строковое значение.

Для получения строкового значения макет использует следующий синтаксис:

```
"@string/string_name"
```

Имя строки, значение которой должно быть возвращено.

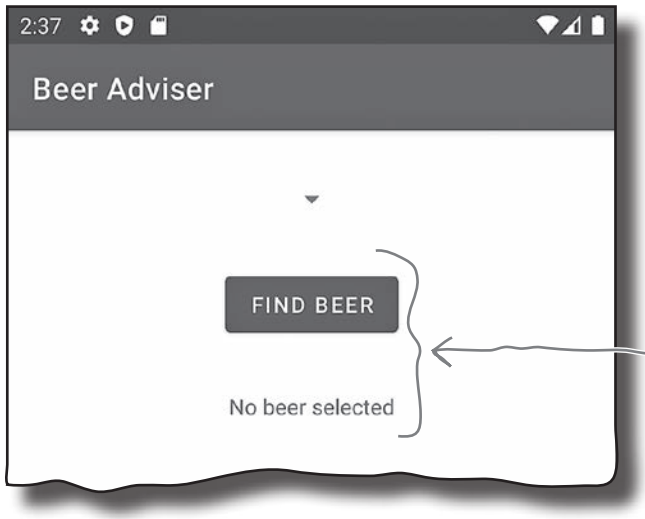
«@string» приказывает Android найти строковый ресурс с указанным именем.



Тест-драйв

После того как макеты были обновлены для использования строковых ресурсов вместо жестко запрограммированных текстовых значений, запустите приложение снова и посмотрите, как оно выглядит. Как и прежде, приложение запускается командой «Run 'app'» из меню Run.

При запуске приложения текст, выводимый на кнопке и в надписи, изменяется. Теперь он использует строковые значения, добавленные в файл *strings.xml*:



Теперь атрибуты text кнопки и надписи используют строковые ресурсы вместо жестко запрограммированного текста.

- [x] Создание проекта
- [x] Обновление макета
- [x] **Добавление ресурсов**
- [ ] Реакция на щелчки
- [ ] Программирование логики

Расслабьтесь

Если при повторном запуске приложения Android Studio спросит, хотите ли вы завершить процесс «app», не беспокойтесь.

Среда просто хочет знать, может ли она завершить старую версию приложения, которая выполняется в текущий момент, чтобы запустить новую. Выберите вариант Terminate и подождите, пока ваше приложение загрузится.

Часто Задаваемые Вопросы

В: Текстовые значения обязательно хранить в файле строковых ресурсов?

О: Не обязательно, но если вы будете жестко программировать текстовые значения, Android выдает предупреждающие сообщения. Поначалу может показаться, что файлы строковых ресурсов только создают лишние хлопоты, но они заметно упрощают многие операции, например локализацию. Также проще использовать строковые ресурсы с самого начала, вместо того чтобы переходить на них впоследствии.

В: Как извлечение строковых значений упрощает локализацию?

О: Предположим, приложение должно использовать английский язык по умолчанию, но если на устройстве выбран французский язык, приложение должно переключаться на французский. Вместо того чтобы жестко программировать разные языки в приложении, можно использовать один файл строковых ресурсов для английского текста и другой файл ресурсов для французского текста.

В: Откуда приложение знает, какой файл строковых ресурсов нужно использовать?

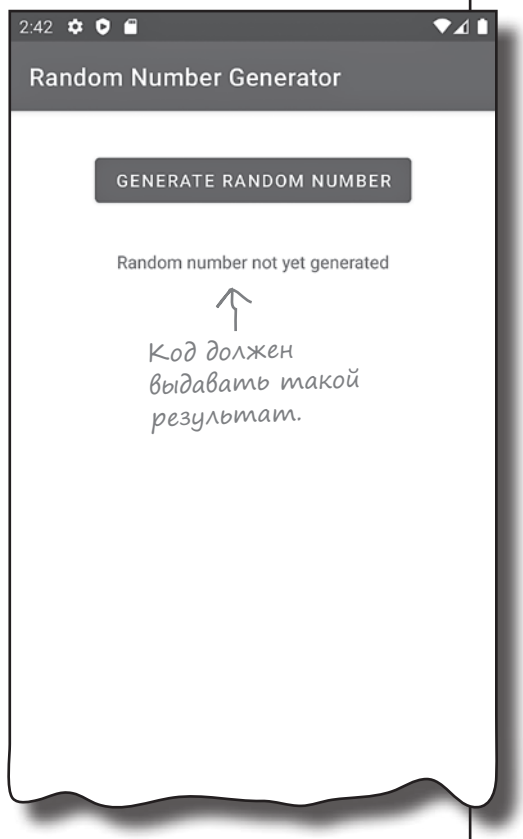
О: Поместите файл строковых ресурсов с английским текстом (используемый по умолчанию) в папку *app/src/main/res/values* как обычно, а файл для французского языка — в новую папку с именем *app/src/main/res/values-fr*. Если на устройстве выбран французский язык, то будут использоваться строковые ресурсы из папки *app/src/main/res/values-fr*. Если на устройстве выбран любой другой язык, то будет использоваться файл строковых ресурсов из папки *app/src/main/res/values*.

Возьмите в руку карандаш

Следующий код (файл макета и файл строковых ресурсов) позаимствован из приложения, предназначенного для генерирования случайных чисел. Завершите код, чтобы он выдавал показанный результат.

activity_main.xml

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    < .....
        android:id="@+id/generate_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android: .....="@string/button_text" />

    < .....
        android:id="@+id/random_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text=" ....." />
</LinearLayout>
```



strings.xml

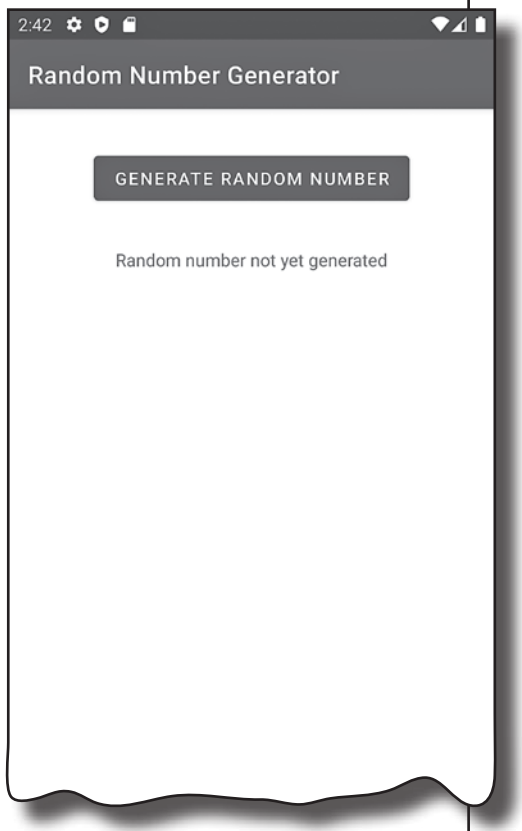
```
<resources>
    <string name="app_name">Random Number Generator</string>
    <string name=" .....">Generate random number</string>
    <string name="no_number">Random number not yet generated</string>
</resources>
```

Возьмите в руку карандаш
Решение

Следующий код (файл макета и файл строковых ресурсов) позаимствован из приложения, предназначенного для генерирования случайных чисел. Завершите код, чтобы он выдавал показанный результат.

activity_main.xml

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    < Button                                   <-- Выводит кнопку.
        android:id="@+id/generate_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:  text  ="@string/button_text" />
                                   <-- Обновляет атрибут text.
    < TextView                     <-- Выводит надпись.
        android:id="@+id/random_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="  @string/no_number " />
</LinearLayout>
```



Использовать строковый ресурс no_number.

strings.xml

```
<resources>
    <string name="app_name">Random Number Generator</string>
    <string name="  button_text  ">Generate random number</string>
    <string name="no_number">Random number not yet generated</string>
</resources>
```

Кнопка использует строковый ресурс с именем button_text.

- Создание проекта
- Обновление макета
- **Добавление ресурсов**
- Реакция на щелчки
- Программирование логики

Добавление значений в список

На данный момент макет включает раскрывающийся список, но в этом списке нет никаких данных. Чтобы список приносил пользу, нужно сообщить ему, какие значения в нем должны отображаться. Каждый раз, когда вы используете раскрывающийся список в коде макета, необходимо указать сопутствующий набор значений, иначе в списке не будет данных и Android Studio выдаст предупреждение.

Набор значений для раскрывающегося списка можно определить практически так же, как мы определим текст на кнопке и надписи: нужно добавить в *strings.xml* ресурс и включить ссылку на этот ресурс в макет. Но вместо того чтобы задавать одиночное значение как строковый ресурс, нужно добавить несколько строк в ресурс-*массив* и использовать этот массив для заполнения списка.

Ресурсы — непрограммные данные (например, графика или строки), используемые в приложении.

Добавление ресурса массива очень похоже на добавление строкового ресурса

Как вы уже знаете, добавление строкового ресурса в файл strings.xml происходит так:

```
<string name="имя_строки">значение</string>
```

где `имя_строки` — имя, а `значение` — значение, выводимое в приложении.

Синтаксис добавления массива строк выглядит так:

```
<string-array name="имя_массива">
    <item>значение1</item>
    <item>значение2</item>
    <item>значение3</item>
    ...
</string-array>
```

Имя массива.

Значения в массиве. Добавьте столько, сколько потребуется.

где `имя_массива` — имя массива, а `значение1`, `значение2`, `значение3` — отдельные строковые значения, входящие в массив.

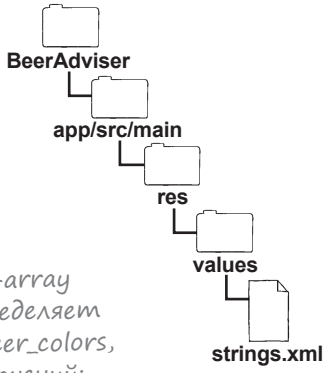
В нашем приложении следует использовать ресурс массива, каждый элемент которого представляет вид пива. Затем массив связывается с раскрывающимся списком, чтобы при щелчке на списке выводился набор видов пива.

Итак, добавим массив строк.

Добавление элемента string-array в файл strings.xml

Чтобы добавить массив строк, откройте файл *strings.xml* и вставьте приведенный ниже код (изменения выделены жирным шрифтом). Следующий фрагмент добавляет ресурс `string-array` с именем `beer_colors`, который будет присоединен к раскрывающемуся списку:

```
<resources>
    <string name="app_name">Beer Adviser</string>
    <string name="find_beer">Find Beer</string>
    <string name="brands">No beer selected</string>
    <string-array name="beer_colors">
        <item>Light</item>
        <item>Amber</item>
        <item>Brown</item>
        <item>Dark</item>
    </string-array>
</resources>
```



Добавьте элемент string-array в файл strings.xml. Он определяет массив строк с именем beer_colors, состоящий из четырех значений: Light, Amber, Brown и Dark.

Вывод значений из массива в раскрывающемся списке

Для ссылки на ресурс массива строк в макете используется синтаксис, сходный с синтаксисом обращения к значению строкового ресурса. Только вместо

```
"@string/имя"
```

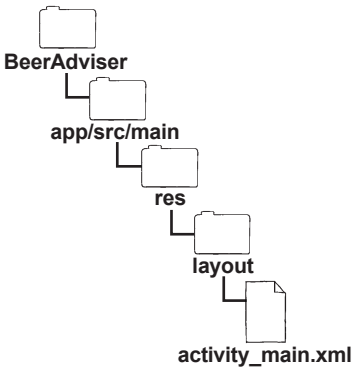
@string используется для ссылок на строки, а @array — для ссылок на массивы.

используется синтаксис

```
"@array/имя_массива"
```

где `имя_массива` — имя массива.

Воспользуемся этим синтаксисом в макете. Перейдите к файлу макета *activity_main.xml* и добавьте к раскрывающемуся списку атрибут `entries` (выделен жирным шрифтом):




```
...
<Spinner
    android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:entries="@array/beer_colors" />
...
```

Это означает «данные для раскрывающегося списка берутся из массива beer_colors».

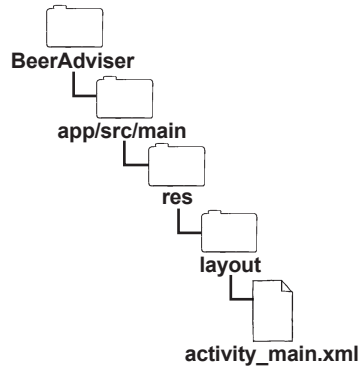
Полный код *activity_main.xml* приведен на следующей странице.

Создание проекта
Обновление макета
Добавление ресурсов
Реакция на щелчки
Программирование логики

Полный код файла activity_main.xml

Ниже приведен полный код *activity_main.xml*. Убедитесь в том, что этот файл содержит весь код, показанный ниже.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <Spinner
        android:id="@+id/beer_color"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:entries="@array/beer_colors" />

    <Button
        android:id="@+id/find_beer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="@string/find_beer" />

    <TextView
        android:id="@+id/brands"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        android:text="@string/brands" />
</LinearLayout>
```



Раскрывающийся список beer_color получает свои данные из массива beer_colors.

Кнопка find_beer получает свой текст из строкового ресурса find_beer.

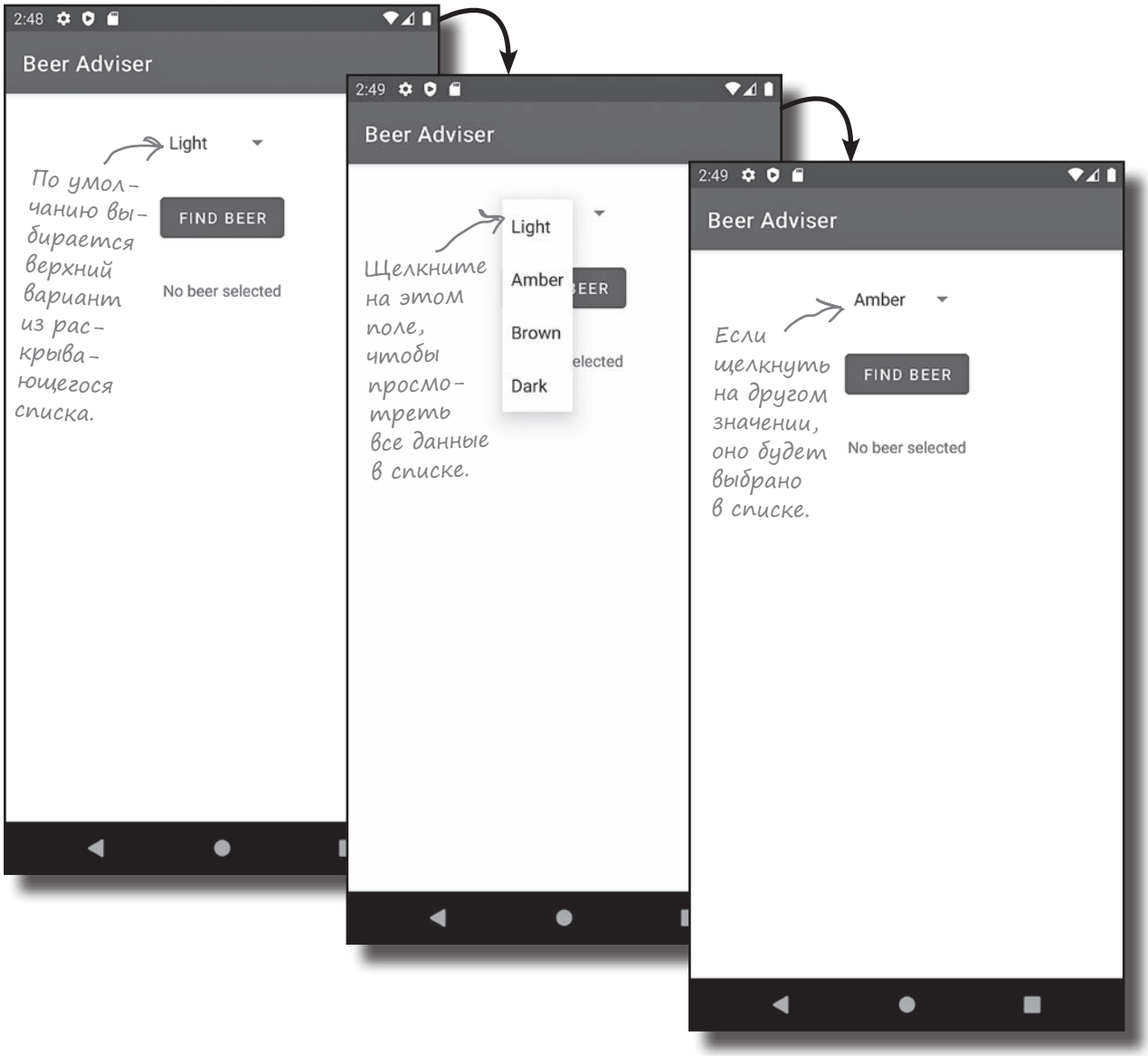
Надпись brands получает свой текст из строкового ресурса brands.

Опробуем новую версию приложения на практике.

Тест-драйв

- [x] Создание проекта
- [x] Обновление макета
- [x] **Добавление ресурсов**
- [ ] Реакция на щелчки
- [ ] Программирование логики

Посмотрим, как эти изменения отразились на приложении. Результат должен выглядеть примерно так:



К настоящему моменту мы создали макет (*activity_main.xml*), который включает раскрывающийся список, кнопку и надпись. Эти представления используют файл строковых ресурсов (*strings.xml*) для получения строковых данных и данных массивов.

Следующее, что необходимо сделать, — заставить приложение обновлять надпись `brands` каждый раз, когда пользователь щелкает на кнопке.

- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

Приложение должно быть интерактивным

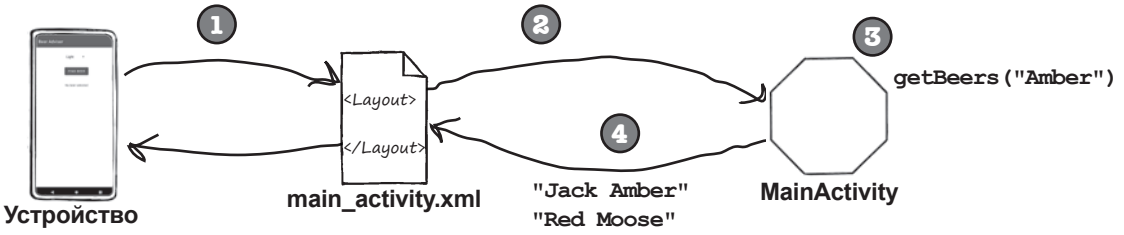
Приложение Beer Adviser выглядит как предполагалось и включает все необходимые представления, но оно еще не выдает никаких рекомендаций. Чтобы приложение было интерактивным, необходимо, чтобы оно что-то делало при нажатии кнопки `find_beer`. Приложение должно работать приблизительно так:

1. **Пользователь выбирает вид пива в раскрывающемся списке и щелкает на кнопке.**

2. **Код MainActivity реагирует на нажатие кнопки.**

3. **MainActivity передает выбранный вид пива созданному нами методу с именем getBeers.** Метод `getBeers()` находит сорта пива, соответствующие заданному виду.

4. **MainActivity обновляет надпись brands, чтобы на устройстве выводился перечень рекомендуемых сортов.**



Чтобы приложение реагировало на действия пользователя подобным образом, необходимо обновить код *MainActivity.kt*, так как этот код отвечает за поведение приложения. Android Studio создает этот файл при создании проекта. Давайте посмотрим, как выглядит текущий код.

- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

Как выглядит код MainActivity

Android Studio создает файл *MainActivity.kt* за вас при создании проекта. Откройте этот файл (если он не был открыт ранее); для этого перейдите в папку *app/src/main/java* и сделайте на файле двойной щелчок.

Код *MainActivity.kt*, который был создан средой Android Studio:

```
package com.hfad.beeradviser

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Имя пакета, которое было указано при создании проекта.

Класс расширяет AppCompatActivity.

Метод onCreate() вызывается при исходном создании активности.

setContentView() сообщает Android, какой макет используется активностью. В данном случае это макет activity_main.xml.

Приведенный выше код — все, что вам необходимо для создания базовой активности. Как видите, это класс, который расширяет класс `AppCompatActivity` и переопределяет его метод `onCreate()`.

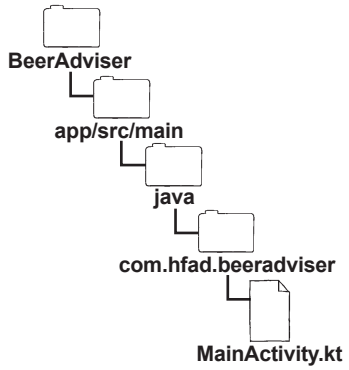
Все активности (не только эта) должны расширять класс активности, например `AppCompatActivity`. Эта тема более подробно рассматривается в главе 5, а пока достаточно знать, что когда класс расширяет `AppCompatActivity`, обычный класс Kotlin преобразуется в полноценную активность Android.

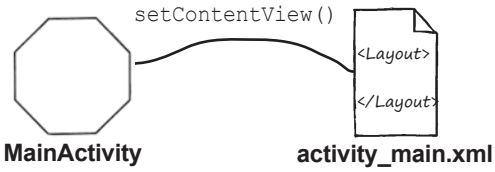
Все активности также должны реализовать метод `onCreate()`. Этот метод вызывается при создании объекта активности, и он используется для выполнения подготовки, например назначения макета, с которым связывается активность. Эта задача решается вызовом `setContentView()`. В приведенном выше примере код

```
setContentView(R.layout.activity_main)
```

сообщает Android, что эта активность использует макет *activity_main.xml*.

Теперь вы знаете, что делает текущая версия кода `MainActivity`. Как же заставить этот метод реагировать на нажатие пользователем кнопки `find_beer`?





Метод setContentView() связывает активность с ее макетом.

Кнопка может прослушивать события on-click...

- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

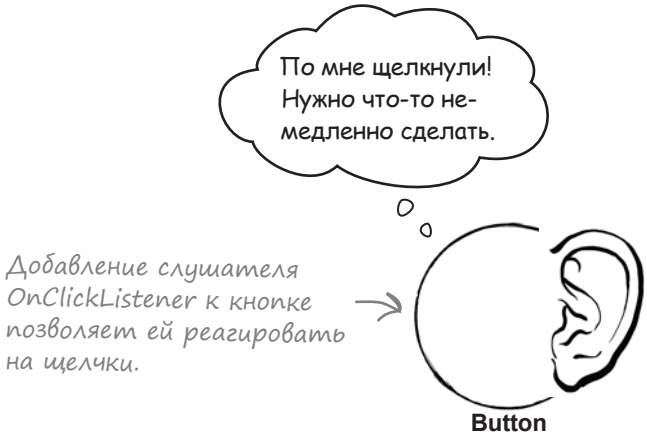
Каждый раз, когда пользователь делает что-то в приложении, происходит некоторое **событие**. В мире Android существует множество разных типов событий: щелчки на кнопках, смахивание по экрану или нажатие физической клавиши на устройстве.

В своем приложении мы хотим знать, когда пользователь щелкает на кнопке `find_beer`, чтобы приложение могло сделать что-то в ответ. Для этого приложение может прослушивать событие **on-click** кнопки, чтобы при каждом возникновении этого события происходило обновление текста в надписи `brands`.

Чтобы кнопка реагировала на события щелчков, следует добавить к ней слушателя OnClickListener.

...используя OnClickListener

Чтобы приложение прослушивало события on-click кнопки, добавьте к кнопке слушатель **`OnClickListener`**. Каждый раз, когда пользователь щелкает на кнопке, `OnClickListener` «слышит» щелчок и реагирует на него.



Чтобы указать, какие действия должны выполняться слушателем `OnClickListener`, передайте блок кода — лямбда-выражение. Оно указывает, что должно происходить при щелчке на кнопке.

Если вы подзабыли, что такое лямбда-выражения в Kotlin, мы рекомендуем обратиться к книге «Head First. Kotlin».

В нашем примере надпись `brands` должна обновляться при каждом щелчке на кнопке `find_beer`. Это означает, что для кнопки `find_beer` нужно добавить слушатель `OnClickListener` и передать ему лямбда-выражение, определяющее, как должна обновляться надпись.

Посмотрим, как это делается.

- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

Получение ссылки на кнопку...

Чтобы добавить слушатель `OnClickListener` к кнопке `find_beer`, сначала необходимо получить ссылку на кнопку в коде активности. Для этого используется метод с именем **`findViewById`**.

Метод `findViewById` позволяет получить ссылку на любое представление в макете, обладающее идентификатором. Просто укажите тип и идентификатор представления, а метод вернет ссылку на него.

В нашем приложении нужно получить ссылку на кнопку с идентификатором «find_beer», поэтому мы используем следующий код:

```
val findBeer = findViewById<Button>(R.id.find_beer)
```

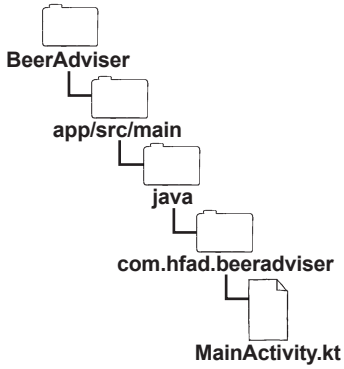
Здесь указывается тип — в данном случае Button.

Нужна кнопка с идентификатором find_beer.

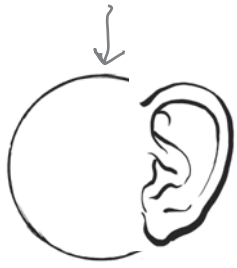
...и вызов ее метода setOnClickListener

После получения ссылки на кнопку можно добавить к ней слушатель `OnClickListener` вызовом метода `setOnClickListener()`:

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            //Код, выполняемый по щелчку на кнопке
        }
    }
}
```

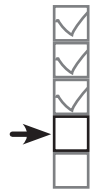


Добавляет слушатель OnClickListener к кнопке find_beer, чтобы кнопка реагировала на события on-click.



Обратите внимание: код `OnClickListener` добавляется в методе `onCreate()` активности `MainActivity`. Метод `onCreate()` выполняется при создании активности, так что добавление вызова `setOnClickListener()` означает, что кнопка `find_beer` начнет реагировать на щелчки как можно раньше.

После добавления слушателя `OnClickListener` к кнопке нужно позаботиться о том, чтобы при щелчке на кнопке в программе что-то происходило.



- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

Передача лямбда-выражения методу setOnClickListener

Чтобы определить, что должна делать кнопка по щелчку, следует передать лямбда-выражение ее методу `setOnClickListener()`. Лямбда-выражение указывает, что должно происходить при каждом щелчке на кнопке.

```
findBeer.setOnClickListener {
    //Здесь размещается код лямбда-выражения
}
```

Код, добавляемый в лямбда-выражение, сообщает слушателю OnClickListener кнопки, что нужно делать по щелчку.

Таким образом, если вы хотите, чтобы кнопка обновляла какой-нибудь текст или выполняла другое действие, включите соответствующий код в лямбда-выражение:

В будущем активность должна выводить список рекомендованных сортов, но пока пусть кнопка обновляет надпись `brands` значением, выбранным в списке `beer_color`. Это позволит нам убедиться в том, что слушатель `OnClickListener` кнопки нормально работает, и только потом переходить к получению *настоящих* рекомендаций.

Но для этого необходимо знать еще две вещи: как редактировать текст в надписи и как получить значение, выбранное в раскрывающемся списке.

- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

Как редактировать текст надписи

Как вы уже знаете, для изменения текста, выводимого в надписи, можно обновить атрибут `text` в разметке XML макета. Таким образом, чтобы обновить текст с использованием кода активности, можно получить ссылку на надпись и обновить ее свойство `text`.

Чтобы изменить текст, выводимый в надписи `brands` (например, заменить его текстом "Gottle of geer"), можно использовать следующий код:

Находит TextView с идентификатором brands.

```
val brands = findViewById<TextView>(R.id.brands)
brands.text = "Gottle of geer"
```

Присваивает текст «Gottle of geer».

В приложении Beer Adviser требуется обновить свойство `text` надписи значением, которое было выбрано в раскрывающемся списке `beer_color`. Но для этого сначала необходимо понять, как получить это значение.

Как получить текущее значение из списка

Для получения текущего значения раскрывающегося списка можно воспользоваться его свойством **`selectedItem`**. Например, следующий код возвращает текущее значение списка `beer_color`:

Находит элемент Spinner с идентификатором beer_color.

```
val beerColor = findViewById<Spinner>(R.id.beer_color)
```

Свойство `selectedItem` может содержать значение любого типа, не только `String`, поэтому значение необходимо преобразовать перед использованием.

Например, в приложении Beer Adviser мы знаем, что раскрывающийся список `beer_color` содержит массив значений `String`, поэтому выбранный пользователем элемент должен относиться к типу `String`. Следовательно, чтобы использовать это значение, необходимо преобразовать его к типу `String` кодом следующего вида:

```
val color = beerColor.selectedItem.toString()
```

Свойство selectedItem может содержать значение произвольного типа, и его необходимо преобразовать к правильному типу перед использованием. В данном случае оно преобразуется к типу String.

Также можно использовать его в строковом шаблоне с кодом следующего вида:

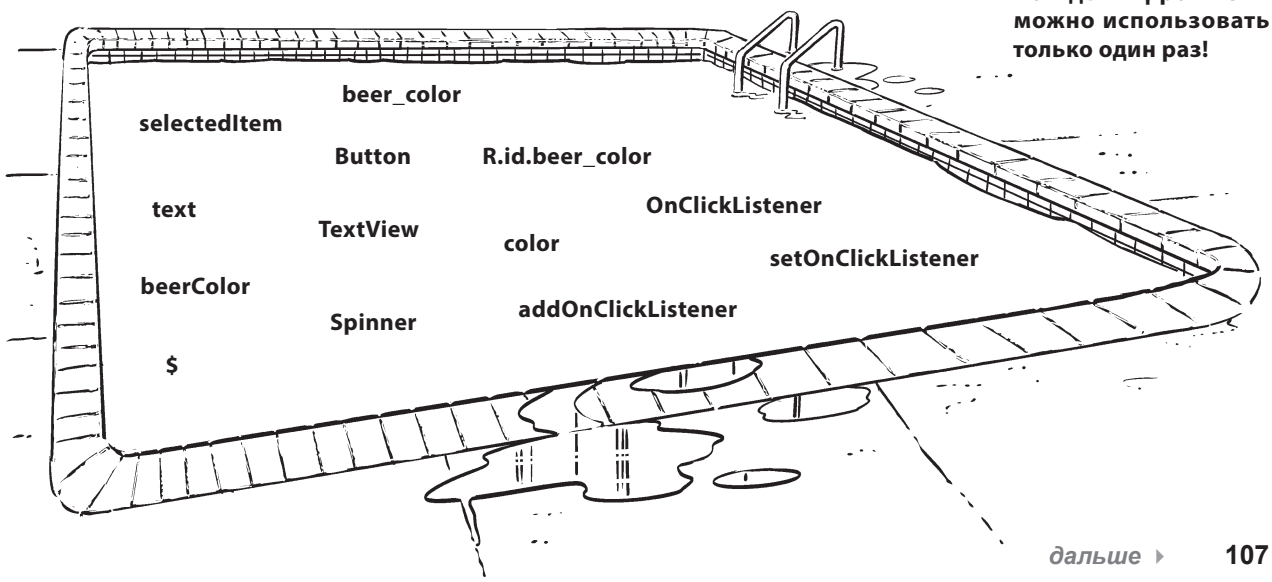
```
val color = "${beerColor.selectedItem}"
```

Теперь вы знаете достаточно, чтобы кнопка `find_beer` могла получить значение раскрывающегося списка `beer_color` и вывести его в надписи `brands`. Но прежде чем вы увидите полный код, попробуйте собрать его самостоятельно в следующем упражнении.

У бассейна

Возьмите фрагменты кода из бассейна и разместите их в пустых строках метода `onCreate()` активности `MainActivity`. Каждый фрагмент можно использовать не более одного раза, причем в бассейне есть и лишние фрагменты. Нужно добиться того, чтобы кнопка `find_beer` реагировала на щелчки и обновляла надпись `brands` значением, выбранным в раскрывающемся списке `beer_color`.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val findBeer = findViewById<Button>(R.id.find_beer)
    findBeer. ..........{
        val beerColor = findViewById<Spinner>( ..........)
        val color = beerColor. ..........
        val brands = findViewById< ..........>(R.id.brands)
        brands. ..... = "Beer color is ..........."
    }
}
```



У бассейна. Решение

Возьмите фрагменты кода из бассейна и разместите их в пустых строках метода `onCreate()` активности `MainActivity`. Каждый фрагмент можно использовать не более одного раза, причем в бассейне есть и лишние фрагменты. Нужно добиться того, чтобы кнопка `find_beer` реагировала на щелчки и обновляла надпись `brands` значением, выбранным в раскрывающемся списке `beer_color`.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val findBeer = findViewById<Button>(R.id.find_beer)
    findBeer.  setOnClickListener  {
        val beerColor = findViewById<Spinner>(  R.id.beer_color  )
        val color = beerColor.  selectedItem
        val brands = findViewById<  TextView  >(R.id.brands)
        brands.  text  = "Beer color is  $ color  "
    }
}
```

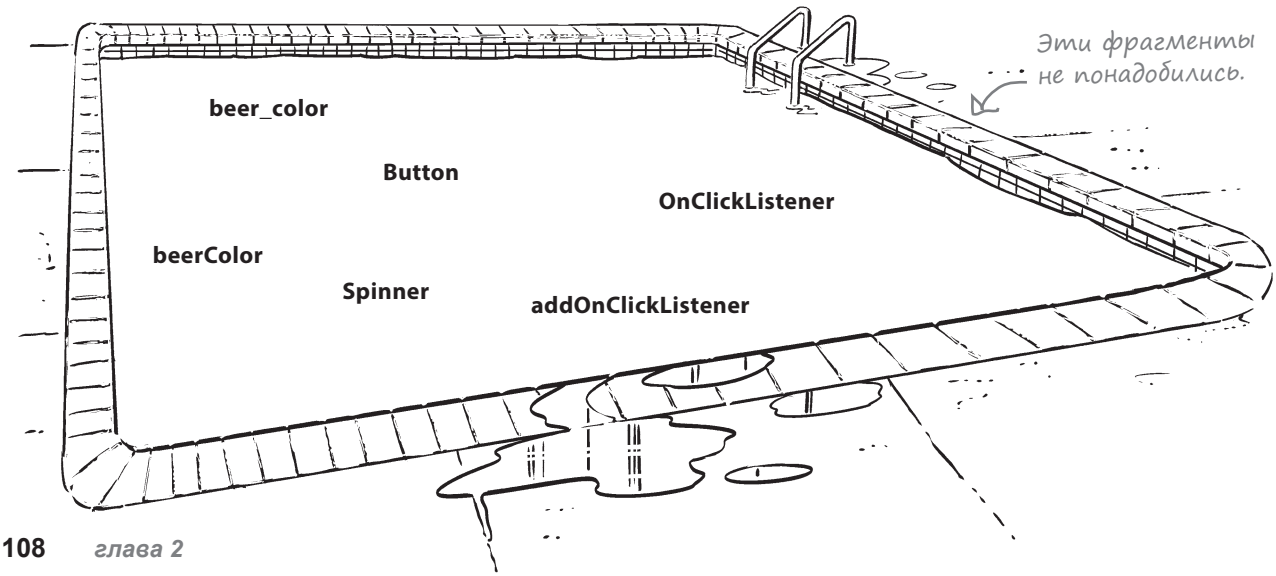
Добавляет OnClickListener.

Получает ссылку на раскрывающийся список с идентификатором beer_color.

brands — элемент TextView.

Обновляет свойство text надписи brands.

Использует цвет в строковом шаблоне.

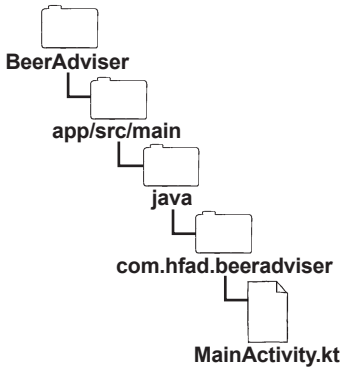


- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

Обновленный код MainActivity.kt

Чтобы кнопка `find_beer` реагировала на щелчки, необходимо обновить `MainActivity`. При каждом щелчке на кнопке должен обновляться текст, отображаемый в надписи `brands`, чтобы выводился вид пива, выбранный в раскрывающемся списке.

Ниже приведен обновленный код *MainActivity.kt* с кодом, собранным воедино в предыдущем упражнении. Обновите файл *MainActivity.kt* и включите в него изменения, выделенные жирным шрифтом:




```
package com.hfad.beeradviser

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.Spinner
import android.widget.TextView

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem
            val brands = findViewById<TextView>(R.id.brands)
            brands.text = "Beer color is $color"
        }
    }
}
```

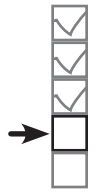
Импортирует классы Button, Spinner и TextView.

Добавляет слушатель OnClickListener к кнопке.

Получает выбранный элемент раскрывающегося списка.

Обновляет свойство text надписи, чтобы оно включало значение из раскрывающегося списка.

Прежде чем опробовать этот код на практике, давайте разберемся в том, что же происходит в этом коде при его выполнении.



- Создание проекта
- Обновление макета
- Добавление ресурсов
- **Реакция на щелчки**
- Программирование логики

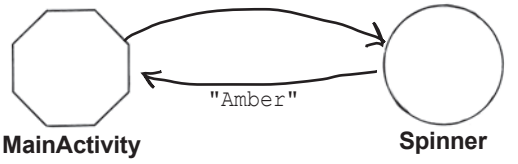
Что происходит при выполнении кода

При запуске приложения пользователь выбирает вид пива и щелкает на кнопке Find Beer. Вот что при этом происходит:

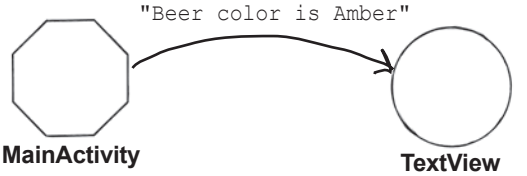
1. **Пользователь выбирает вид пива из раскрывающегося списка и щелкает на кнопке Find Beer.**

2. **Слушатель OnClickListener кнопки узнает, что на кнопке был сделан щелчок.**

3. **Код OnClickListener из MainActivity получает значение, выбранное в раскрывающемся списке (в данном случае Amber).**



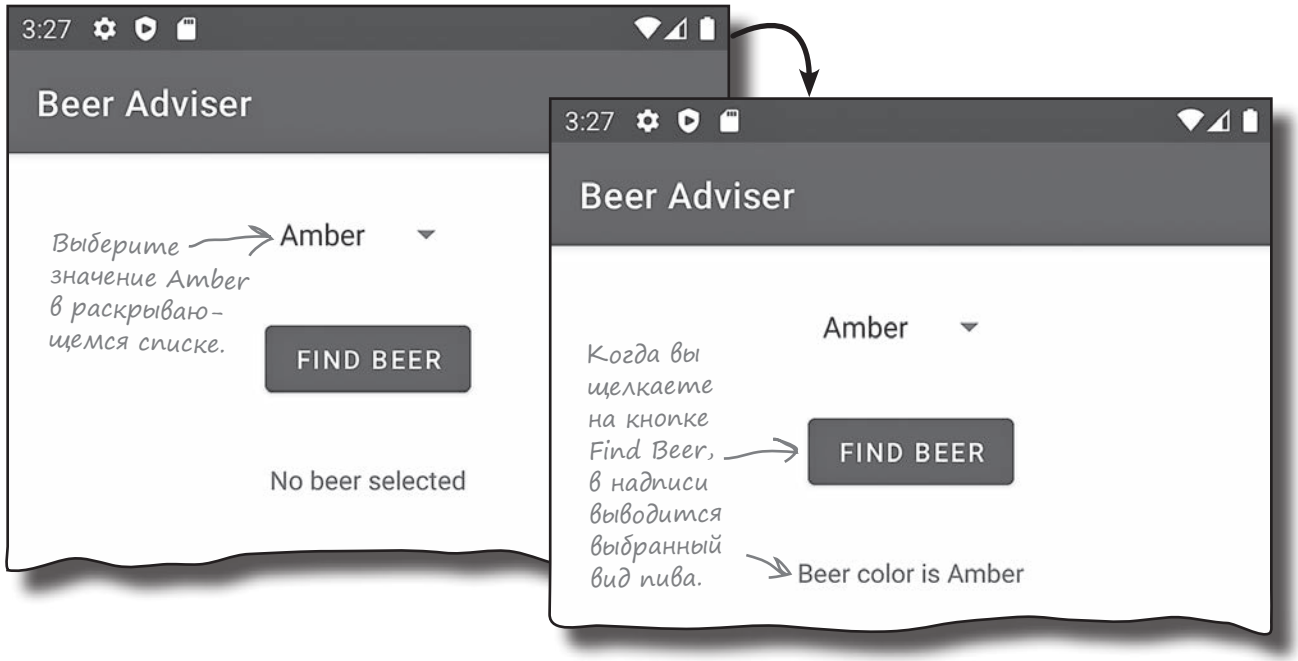
4. **Он обновляет свойство text надписи, сообщая о том, что пользователь выбрал в раскрывающемся списке вариант Amber.**



Тест-драйв

Обновите файл *MainActivity.kt* и запустите приложение.

Когда вы выбираете вид пива из раскрывающегося списка и щелкаете на кнопке Find Beer, выбранное значение появляется в надписи.



Часто Задаваемые Вопросы

В: Вы сказали, что мы получаем ссылку на надпись `brands`, передавая методу `findViewById` значение `R.id.brands`. Также вы сказали, что мы отдаем команду `MainActivity` использовать макет *`activity_main.xml`*, передавая методу `setContentView` значение `R.layout.activity_main`. И в `R.id.brands`, и в `R.layout.activity_main` упоминается что-то с именем R. Так что же такое R?

О: R относится к *R.java*. Это специальный файл, который автоматически генерируется при построении приложения. Android использует *R.java* для отслеживания любых ресурсов, используемых в приложении, таких как макеты, строковые ресурсы и представления.

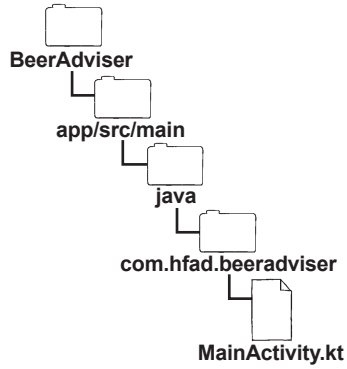
Когда вы передаете методу `findViewById` значение `R.id.brands`, Android ищет идентификатор `brands` в `R` и использует его для возвращения ссылки на представление `brands`. Аналогичным образом при передаче методу `setContentView` значения `R.layout.activity_main` Android обращается к `R` для получения правильного макета и связывает его с активностью.

Вам не придется изменять код *R.java* самостоятельно: Android Studio автоматически генерирует его за вас.

Добавление метода getBeers()

Итак, теперь вы знаете, как кнопка `find_beer` реагирует на щелчки, и мы можем изменить ее поведение. Каждый раз, когда пользователь щелкает на кнопке, кнопка должна выводить рекомендации в зависимости от значения, выбранного в раскрывающемся списке. Для этого в файл *MainActivity.kt* будет добавлен новый метод `getBeers()`, который вызывается в коде `OnClickListener` кнопки.

Метод `getBeers()` состоит из чистого кода Kotlin. Он получает один строковый параметр, в котором передается вид пива, и возвращает `List<String>` с рекомендациями. Добавьте метод `getBeers()` (приведен ниже жирным шрифтом) в файл *MainActivity.kt*:

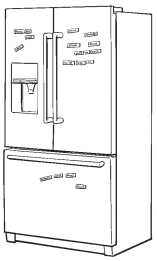



```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            ...
        }
    }

    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

Добавьте в MainActivity.kt метод getBeers(). Этот метод относится исключительно к Kotlin; в нем нет ничего связанного с Android.

Далее необходимо обновить лямбда-выражение, передаваемое методу `setOnClickListener()` кнопки `find_beer`. Оно должно передавать вид пива, выбранный в раскрывающемся списке, методу `getBeers()` и обновлять надпись `brands` возвращаемым значением.

Попробуйте собрать воедино код для решения этой задачи. В этом вам поможет следующее упражнение.



Развлечения с магнитами

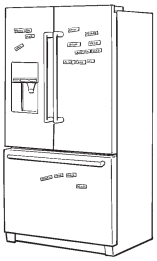
Кто-то выложил код `MainActivity` на холодильнике, заменяя пустые места магнитами. К сожалению, от сквозняка некоторые магниты упали на пол. Сможете ли вы снова собрать код?

Код должен вызывать метод `getBeers()` и выводить все возвращенные элементы в надписи `brands`. Каждый элемент должен выводиться в новой строке.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem

            val ..... = getBeers( ..... )

            val beers = beerList.reduce { str, item -> str + '\n' + item}
            val brands = findViewById<TextView>(R.id.brands)

            brands.text = .....
        }
    }

    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

Функция Kotlin reduce инициализирует строку str первым элементом beerList. Затем она перебирает оставшиеся элементы beerList и добавляет каждый элемент к str, отделяя его разрывом строки.

beers **color** **.toString()** **beerList**



Развлечения с магнитами. Решение

Кто-то выложил код `MainActivity` на холодильнике, заменяя пустые места магнитами. К сожалению, от сквозняка некоторые магниты упали на пол. Сможете ли вы снова собрать код?

Код должен вызывать метод `getBeers()` и выводить все возвращенные элементы в надписи `brands`. Каждый элемент должен выводиться в новой строке.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem

            val beerList = getBeers( color .toString() )

            val beers = beerList.reduce { str, item -> str + '\n' + item}
            val brands = findViewById<TextView>(R.id.brands)

            brands.text = beers
        }
    }

    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

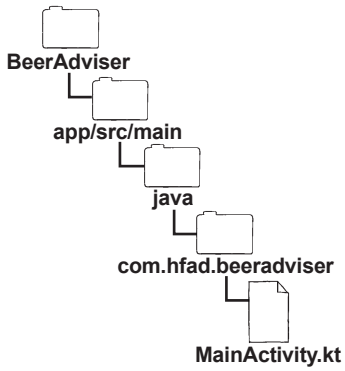
Преобразует выбранный элемент в строку и передает его методу getBeers.

Выводит строку beers в надписи brands.

Создание проекта
Обновление макета
Добавление ресурсов
Реакция на щелчки
Программирование логики

Полный код MainActivity.kt

Ниже приведен полный код `MainActivity`. Внесите изменения, выделенные жирным шрифтом, в *MainActivity.kt*.




```
package com.hfad.beeradviser

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.Spinner
import android.widget.TextView

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val findBeer = findViewById<Button>(R.id.find_beer)
        findBeer.setOnClickListener {
            val beerColor = findViewById<Spinner>(R.id.beer_color)
            val color = beerColor.selectedItem
            val beerList = getBeers(color.toString())
            val beers = beerList.reduce { str, item -> str + '\n' + item }
            val brands = findViewById<TextView>(R.id.brands)
            brands.text = "Beer color is $color" beers
        }
    }

    fun getBeers(color: String): List<String> {
        return when (color) {
            "Light" -> listOf("Jail Pale Ale", "Lager Lite")
            "Amber" -> listOf("Jack Amber", "Red Moose")
            "Brown" -> listOf("Brown Bear Beer", "Bock Brownie")
            else -> listOf("Gout Stout", "Dark Daniel")
        }
    }
}
```

Метод getBeers используется для получения списка сортов пива.

Выводит рекомендуемые сорта в надписи Brands.

Формирует строку результата с выводом каждого сорта пива в новой строке.

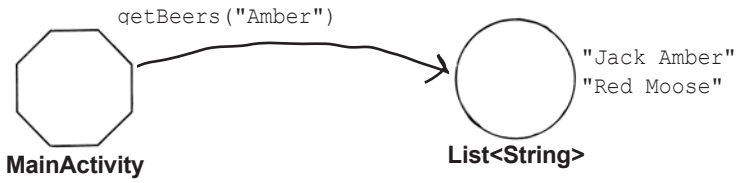
Прежде чем запускать итоговую версию приложения, разберемся, что же происходит в коде при запуске.

Что происходит при запуске кода

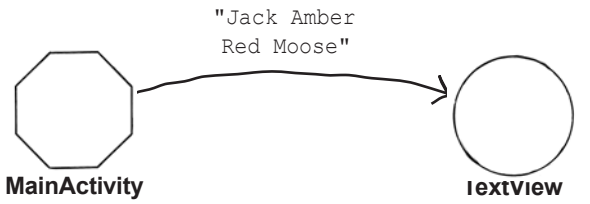
При запуске приложения происходит следующее:

1. **Когда пользователь щелкает на кнопке Find Beer, слушатель OnClickListener кнопки узнает о щелчке.**

2. **Код OnClickListener из MainActivity вызывает метод getBeers() и передает ему вид пива, выбранный в раскрывающемся списке.** Метод `getBeers()` возвращает список сортов пива, который `MainActivity` сохраняет в отдельной переменной.



3. **MainActivity форматирует список сортов пива и присваивает его свойству text надписи.**



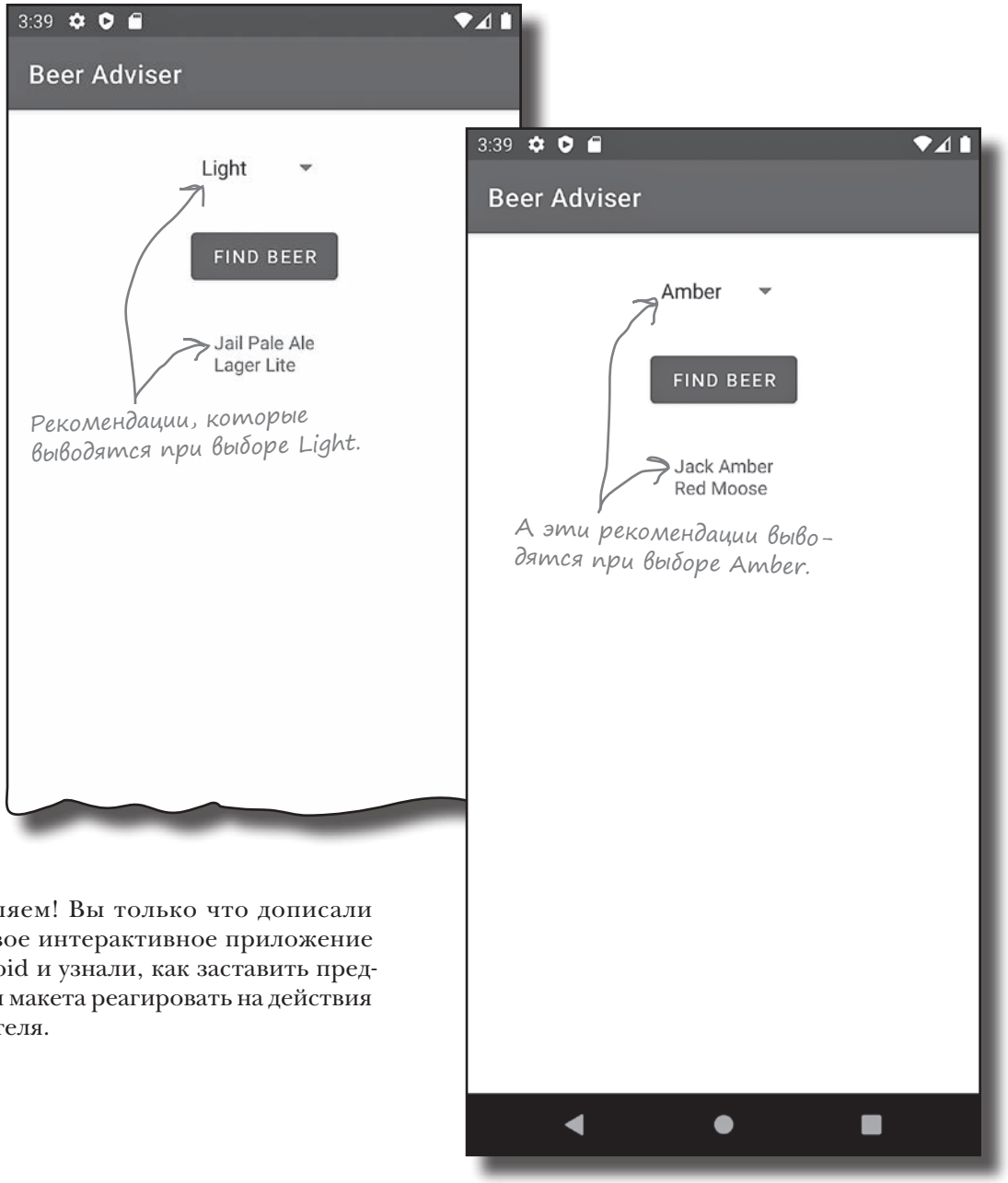
А теперь давайте опробуем приложение на практике.

Тест-драйв

- [x] Создание проекта
- [x] Обновление макета
- [x] Добавление ресурсов
- [x] Реакция на щелчки
- [x] **Программирование логики**

Внесите все изменения в приложение и запустите его.

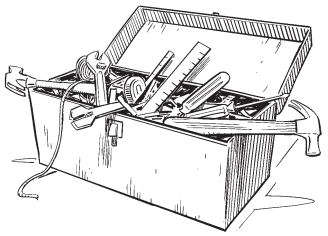
Когда вы выбираете разные виды пива и щелкаете на кнопке Find Beer, приложение выводит подборку подходящих сортов пива. При выборе варианта Light выводится один набор рекомендаций, а при выборе варианта Amber — другой.



Поздравляем! Вы только что дописали свое первое интерактивное приложение для Android и узнали, как заставить представления макета реагировать на действия пользователя.

ГЛАВА 2

Ваш инструментарий Android



Глава 2 осталась позади, а ваш инструментарий пополнился средствами построения интерактивных приложений Android.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Элемент `<Button>` используется для добавления кнопки.
- Элемент `<Spinner>` используется для добавления раскрывающегося списка.
- Все компоненты графического интерфейса наследуют от класса Android `View`.
- *strings.xml* — файл строковых ресурсов. Он используется для отделения текстовых значений от макетов и активностей, а также для упрощения локализации.
- Синтаксис добавления строк в *strings.xml*:

```
<string name="name">
  Value
</string>
```

- Для обращения к строкам из макета используется синтаксис:

```
"@string/name"
```

- Массив строковых значений создается в *strings.xml* конструкцией следующего вида:

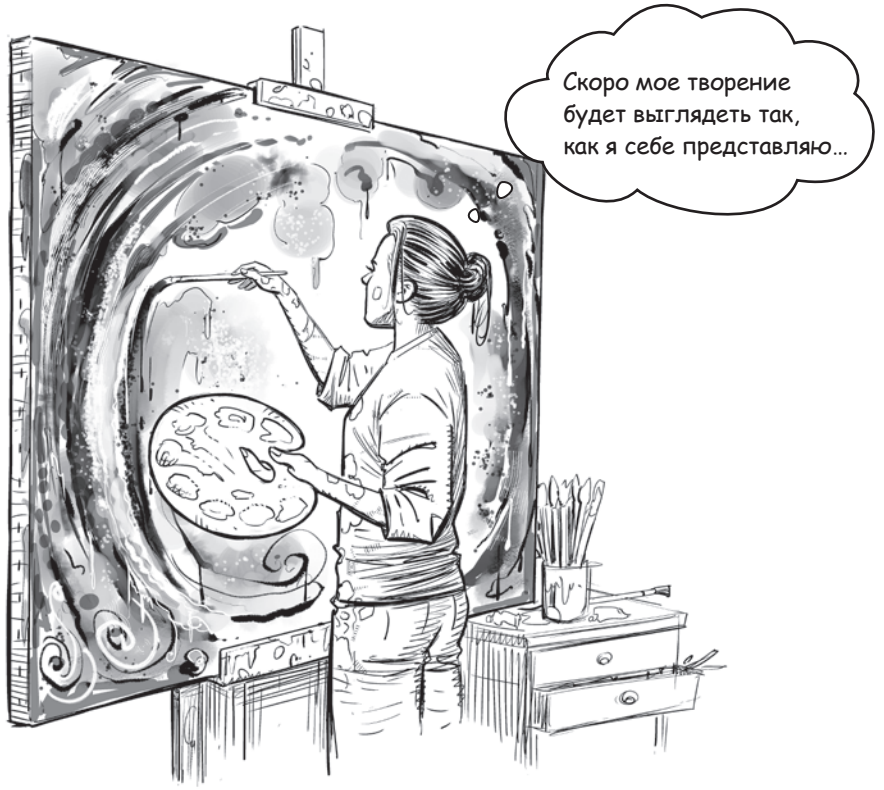
```
<string-array name="array_name">
    <item>string1</item>
    ...
</string-array>
```

- Для обращения к массивам `string-array` из макета используется синтаксис:

```
"@array/array_name"
```

- Метод `findViewById` используется для получения ссылки на представление по его идентификатору.
- Чтобы кнопка реагировала на щелчки, вызовите ее метод `setOnClickListener`.
- Метод `setOnClickListener` получает один параметр — лямбда-выражение, которое описывает, что должно происходить при щелчке на кнопке.
- Чтобы изменить текст, выводимый в представлении, измените его свойство `text`.
- Для получения выбранного элемента в раскрывающемся списке используется свойство `selectedItem`.
- Свойство `selectedItem` может содержать значение любого типа, и перед использованием его следует преобразовать к нужному типу.

Как работают макеты



Мы едва затронули тему использования макетов. К настоящему моменту вы видели, как разместить представления в простых линейных макетах, но это лишь малая часть того, на что способны макеты. В этой главе мы **рассмотрим вопрос глубже** и покажем, как на самом деле работают макеты. Вы узнаете, как **настроить линейные макеты**, и научитесь использовать **фреймовые макеты** и **представления с прокруткой**. А к концу главы вы узнаете, что, несмотря на внешние различия, все макеты — и добавленные к ним представления — имеют **больше общего**, чем кажется на первый взгляд.

Все начинается с макета

Как вы уже знаете, файлы макетов пишутся на языке разметки XML, и они определяют, как будет выглядеть приложение.

Каждый раз, когда вы создаете макет, необходимо сделать три вещи:

1. **Задать разновидность макета.**
Чтобы сообщить Android, как представления (например, кнопки и надписи) должны быть выстроены на экране, необходимо выбрать разновидность макета. Например, в линейном макете `LinearLayout` представления выстраиваются в столбец или в строку.

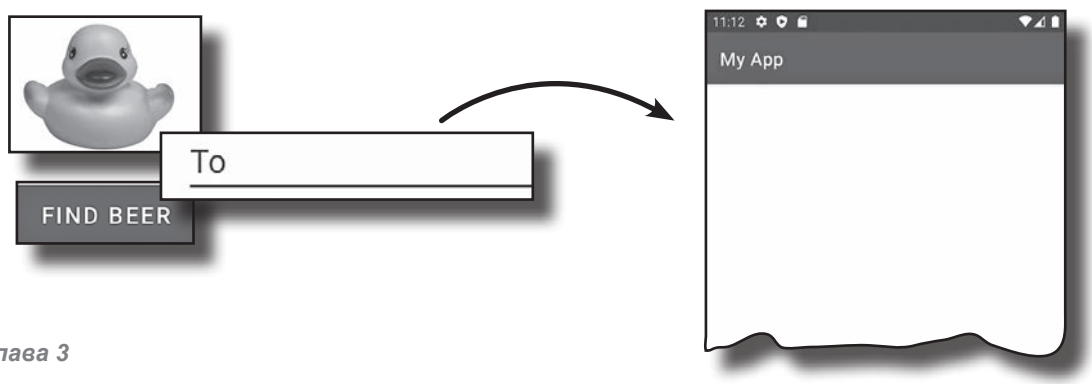
```
<LinearLayout ...>


</LinearLayout>
```

2. **Задать представления.**
Каждый макет содержит одно или несколько представлений, которые используются приложением для вывода информации или взаимодействия с пользователем.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me" />
```

3. **Приказать активности использовать макет.**
Чтобы сообщить Android, какая активность должна использовать только что определенный макет, следует добавить в активность код Kotlin следующего вида:

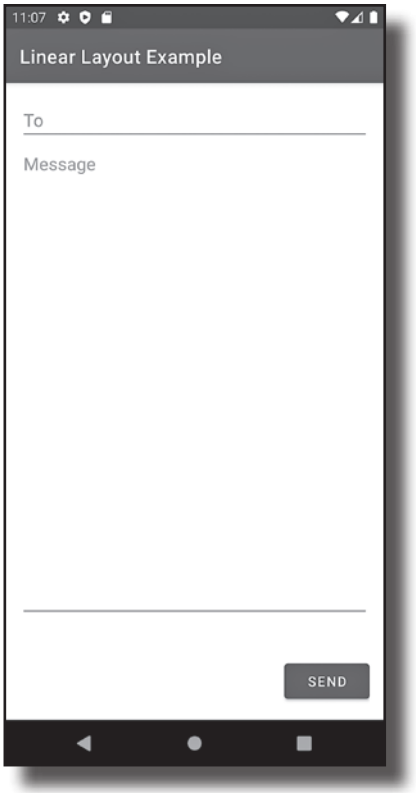
```
setContentView(R.layout.activity_main)
```



В Android существуют разные виды макетов

В Android существуют разные виды макетов, и каждый использует собственные правила размещения представлений. Например, линейный макет всегда размещает представления в строку или в столбец, тогда как во фреймовом макете представления накладываются друг на друга. Выбор типа макета зависит от того, как представления должны размещаться на экране устройства.

Линейный макет размещает свои представления в один столбец или строку.

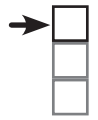


Фреймовый макет накладывает свои представления друг на друга.

Используйте макет, наиболее подходящий для вашего экрана

Все приложения, которые встречались вам до настоящего момента, использовали линейные макеты для размещения представлений в один столбец. В этой главе мы подробнее изучим линейные макеты, а также представим два других вида макетов: фреймовый макет и представление с прокруткой.

Начнем с линейных макетов.



Линейный макет
Фреймовый макет
Представление с прокруткой

Построение линейного макета

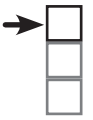
Мы воспользуемся линейным макетом для построения интерфейса, изображенного ниже. В данном случае линейный макет подходит особенно хорошо, потому что представления выстроены в один столбец. Как вы уже знаете, в линейном макете представления выстраиваются одно за другим по вертикали (столбец) или по горизонтали (строка).

Макет состоит из двух редактируемых текстовых полей (надписи, в которых можно вводить текст) и кнопки. Результат должен выглядеть так:

Создание нового проекта

Для приложения с линейным макетом будет создан новый проект Android Studio.

Создайте новый проект по той же схеме, которая использовалась в предыдущих главах. Выберите вариант Empty Activity, введите имя «Linear Layout Example» с именем пакета «com.hfad.linearlayoutexample» и подтвердите место сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android.



Линейный макет
Фреймовый макет
Представление с прокруткой

Как определить линейный макет

Как вы уже знаете, линейный макет определяется элементом <LinearLayout>. Код выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    ...>
</LinearLayout>
```

Эта строка задает формат XML.

<LinearLayout> используется для определения линейного макета.

Атрибуты layout_width и layout_height определяют, какие размеры должен иметь макет.

Атрибут orientation определяет, как должны размещаться представления — по вертикали или по горизонтали.

Также могут присутствовать и другие атрибуты.

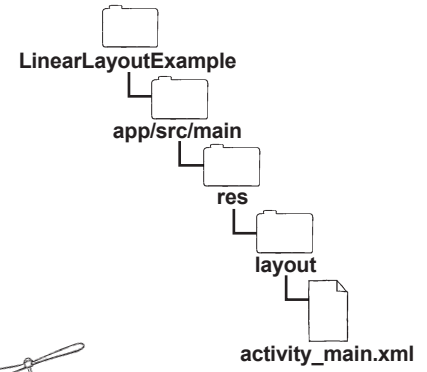
Элемент <LinearLayout> содержит различные атрибуты, необходимые для определения его внешнего вида и поведения.

Первый атрибут — **xmlns:android** — определяет пространство имен с именем android. Ему должно быть присвоено значение "http://schemas.android.com/apk/res/android", как показано выше. Определение этого пространства имен предоставляет вашему макету доступ к элементам и атрибутам, необходимым вашему макету, и оно должно присутствовать во всех файлах макетов, которые вы создаете.

Следующие два атрибута, **android:layout_width** и **android:layout_height**, указывают, какую ширину и высоту должен иметь макет. Эти атрибуты обязательны для *всех* типов макетов и представлений.

Атрибутам android:layout_width и android:layout_height присваиваются строковые значения "wrap_content" и "match_parent" или конкретные размеры, например 8dp (8 аппаратно-независимых пикселей). "wrap_content" означает, что размер макета должен быть минимально достаточным для размещения всех его внутренних представлений, а "match_parent" — что размеры макета подгоняются по размерам его родителя, в нашем случае экрана устройства за вычетом отступов (они будут подробно рассмотрены через несколько страниц). Обычно макетам назначается ширина и высота "match_parent".

Следующий атрибут задает ориентацию линейного макета. Далее мы рассмотрим возможные значения этого атрибута.

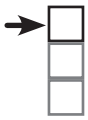


Серьезное программирование

Некоторые устройства способны выводить очень четкие изображения за счет использования мелких пикселей. Другие устройства дешевле в производстве, потому что их экраны состоят из меньшего количества больших пикселей.

Аппаратно-независимые пиксели (dp) используются для создания интерфейсов, которые кажутся очень мелкими на одних устройствах и слишком крупными на других. Размеры в аппаратно-независимых пикселях остаются приблизительно одинаковыми для всех устройств.

Размеры текста задаются в **масштабируемых пикселях (sp)**. Масштабируемые пиксели похожи на аппаратно-независимые пиксели, не считая того, что они используются для шрифтов.



Линейный макет
Фреймовый макет
Представление с прокруткой

Вертикальная и горизонтальная ориентация

Направление, в котором выстраиваются представления, определяется атрибутом **`android:orientation`**.

Чтобы представления выстраивались по вертикали в один столбец, используется следующее значение:

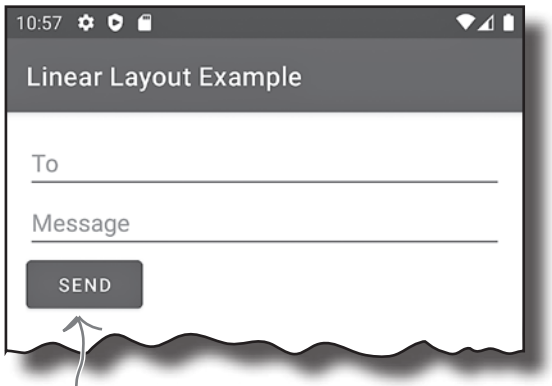
```
android:orientation="vertical"
```

а для размещения представлений по горизонтали в одну строку используется атрибут

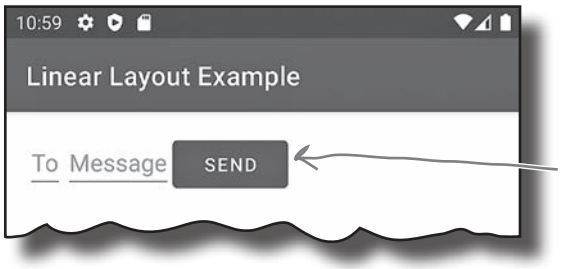
```
android:orientation="horizontal"
```

При горизонтальной ориентации порядок размещения представлений зависит от языковых настроек устройства.

Если на устройстве выбран язык, который читается слева направо (например, английский), представления размещаются по горизонтали слева направо:



Если выбрана вертикальная ориентация, представления выстраиваются по вертикали в один столбец.



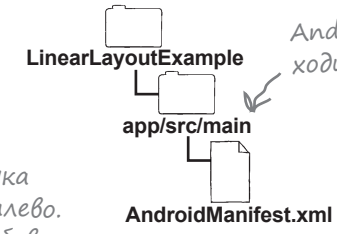
Если выбрана горизонтальная ориентация, представления выстраиваются по горизонтали в строку. В данном случае на устройстве выбран английский язык, поэтому представления размещаются слева направо.

Если на устройстве выбран язык с письмом справа налево (например, арабский), вы можете выбрать вывод представлений справа налево, чтобы первое представление отображалось у правого края макета. Чтобы включить эту возможность, добавьте свойство с именем **`supportsRtl`** в файл *AndroidManifest.xml* и присвойте ему значение `"true"`:

supportsRtl означает «поддержка направления справа налево».

```
<manifest ...>
    <application
        ...
        android:supportsRtl="true">
        ...
    </application>
</manifest>
```

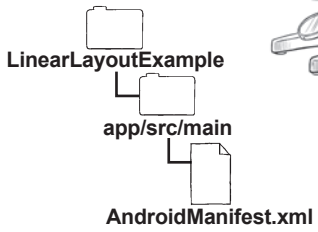
Включает поддержку языка с направлением справа налево. Обычно Android Studio добавляет эту строку за вас.



AndroidManifest.xml находится в этой папке.

Прежде чем рассматривать другие атрибуты, которые могут использоваться с линейными макетами, давайте ненадолго отвлечемся и поближе познакомимся с файлом *AndroidManifest.xml*.

Анатомия AndroidManifest.xml



Каждое приложение Android должно включать файл с именем *AndroidManifest.xml*, который находится в папке *app/src/main* вашего проекта. Файл *AndroidManifest.xml* содержит важнейшую информацию о приложении, необходимую для операционной системы Android, средств построения приложений Android и Google Play Store. Эта информация включает имя пакета приложения, описания его активностей и разрешения, необходимые для работы приложения. Android Studio создает файл *AndroidManifest.xml* при создании проекта.

Типичный файл *AndroidManifest.xml* выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.linearlayoutexample">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.LinearLayoutExample">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Имя пакета приложения.

Android Studio предоставляет стандартные значки приложения.

Приложение поддерживает языки с написанием справа налево.

Тема влияет на внешний вид приложения. Эта тема более подробно рассматривается в главе 8.

Приложение содержит одну активность с именем MainActivity.

MainActivity — главная активность приложения, которая начинает работу при запуске приложения.

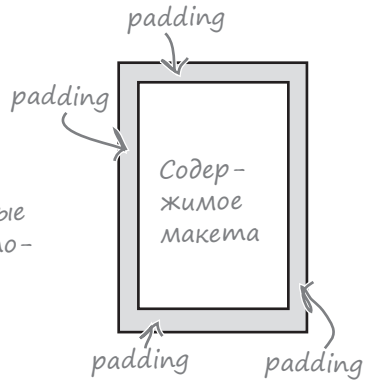
Android Studio обычно включает свойство `supportsRtl` в *AndroidManifest.xml* автоматически, чтобы ваше приложение по умолчанию поддерживало языки с написанием справа налево.

Использование padding для добавления отступов

При выборе разновидности макета, которую вы хотите использовать, можно задать один или несколько атрибутов **padding** для добавления дополнительного пространства между сторонами макета и его содержимым. Так, в следующем коде атрибут `android:padding` используется для добавления отступов 16dp ко всем сторонам:

```
<LinearLayout ...
    android:padding="16dp" >

</LinearLayout>
```

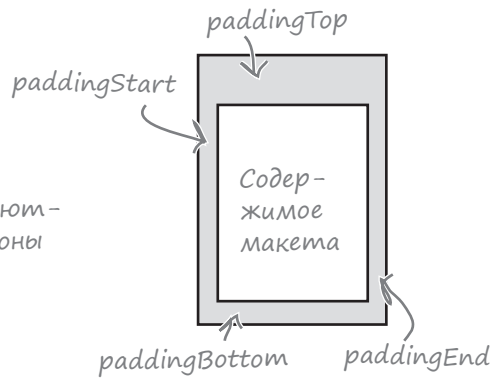
Добавляет одинаковые отступы со всех сторон макета.



Если вы хотите назначить разные отступы для разных сторон, их можно задавать для каждой стороны по отдельности. Например, следующий код добавляет отступ 32dp к верхней стороне макета и 16dp к другим сторонам:

```
<LinearLayout ...
    android:paddingBottom="16dp"
    android:paddingStart="16dp"
    android:paddingEnd="16dp"
    android:paddingTop="32dp" >

</LinearLayout>
```

Отступы определяются для каждой стороны по отдельности.

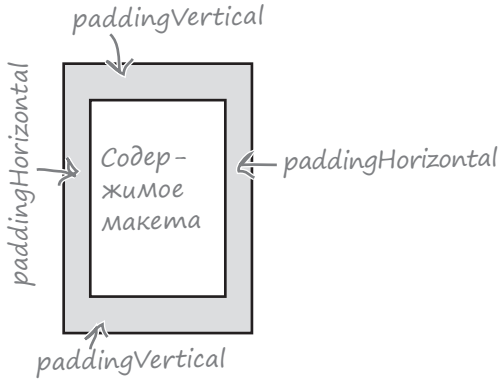


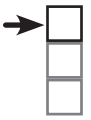
Атрибут `android:paddingStart` добавляет отступы к начальной стороне макета. Для языков с написанием слева направо (например, английского) начальной является левая сторона. Если же на устройстве выбран язык с написанием справа налево — и приложение поддерживает такие языки, — начальной является правая сторона.

Атрибут `android:paddingEnd` добавляет отступ к конечной стороне макета — правой для языков с написанием слева направо и левой для языков с написанием справа налево (если они поддерживаются приложением).

Если вы хотите применить одинаковые отступы к горизонтальным или вертикальным сторонам, также можно использовать значения `android:paddingHorizontal` и `android:paddingVertical`. Эти атрибуты добавляют отступы к горизонтальным и вертикальным сторонам соответственно.

Итак, теперь вы знаете, как добавить отступы к линейному макету. Давайте добавим отступы к макету, который мы строим.





Линейный макет
Фреймовый макет
Представление с прокруткой

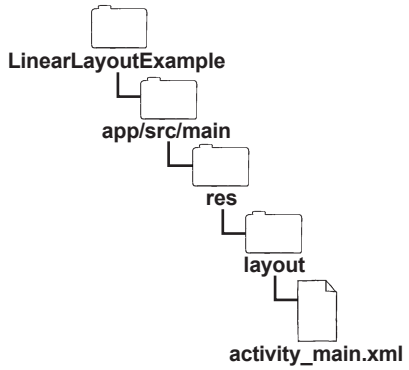
Текущий код макета

В этом приложении мы воспользуемся линейным макетом с вертикальной ориентацией и добавим отступы 16dp к каждой стороне, чтобы стороны макета были отделены от его содержимого небольшими интервалами.

Откройте файл *activity_main.xml* и приведите его к следующему виду:

Замените текущий код активности, чтобы в нем использовался линейный макет.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

</LinearLayout>
```



На текущий момент мы определили пустой линейный макет; давайте сделаем следующий шаг и добавим в него несколько представлений.

Часто Задаваемые Вопросы

В: Я вижу, что в приведенном выше коде макета указано дополнительное пространство имен `tools`. Для чего это нужно?

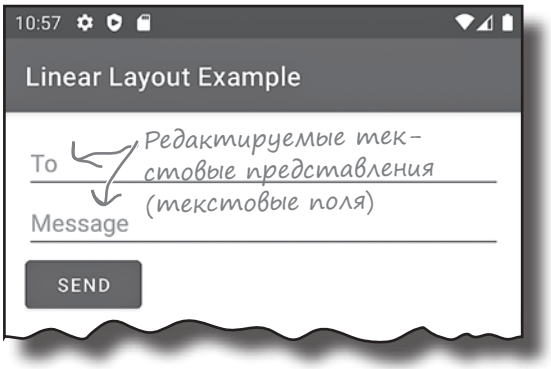
О: Пространство имен `tools` включает набор атрибутов, которые используются некоторыми средствами, включенными в Android Studio. Оно не изменяет внешний вид вашего приложения на экране устройства, а только немного упрощает программирование.

Отступы создают дополнительные интервалы между сторонами макета и его содержимым.

Отступы также могут использоваться с представлениями. Они создают дополнительные интервалы между сторонами представления и его содержимым.

Текстовое поле предназначено для ввода текста

Линейный макет будет содержать кнопку и два текстовых поля (предназначенных для ввода текста). Вы уже умеете работать с кнопками, так что перед тем, как переходить к обновлению макета, нужно понять, как добавить текстовое поле.



Текстовое поле — разновидность текстового представления с возможностью ввода текста. Для добавления его в макет используется элемент **<EditText>**, а код выглядит примерно так:

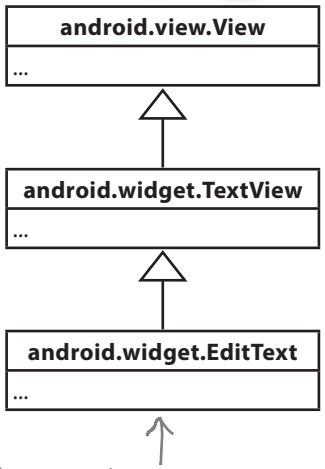
```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="To"
    android:inputType="text" />
```

Приведенный выше код создает редактируемое текстовое поле, ширина которого соответствует ширине его родителя, а высота выбирается минимально достаточной для хранения его содержимого.

Атрибут **android:hint** используется для определения текста подсказки. Этот текст выводится в том случае, если в текстовом поле еще не вводился текст; он подсказывает пользователю, какую информацию нужно ввести. В приведенном примере используется жестко фиксированный текст подсказки, но в реальном приложении вместо него следовало бы использовать строковый ресурс.

Атрибут `android:inputType` сообщает тип данных, которые должны вводиться пользователем, чтобы система Android могла предоставить клавиатуру правильного типа. В приведенном примере используется разметка:

```
android:inputType="text"
```



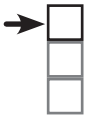
Класс EditText является субклассом TextView. Он расширяет функциональность TextView возможностью ввода данных.

Такое текстовое поле позволяет ввести одну строку текста. Приведем несколько примеров полезных текстовых полей, которые пригодятся вам при создании собственных приложений:

Значение	Что делает
text	Текстовое поле для ввода одной строки текста.
textMultiLine	Текстовое поле для ввода нескольких строк текста.
phone	Клавиатура для ввода телефонных номеров.
textPassword	Клавиатура для ввода текста, введенные символы скрываются.
textCapSentences	Первое слово каждого предложения начинается с буквы верхнего регистра.

За дополнительной информацией обращайтесь к электронной документации Android по адресу https://developer.android.com/training/keyboard-input/style.

Теперь вы знаете, как включать текстовые поля в приложения, и мы можем заняться добавлением представлений в код макета создаваемого приложения.



Линейный макет
Фреймовый макет
Представление с прокруткой

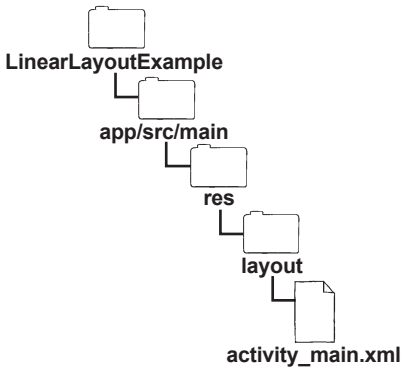
Добавление представлений в XML макета

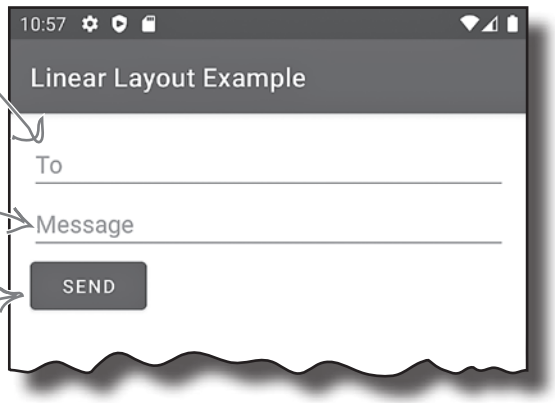
При определении линейного макета вы перечисляете представления макета в том порядке, в каком они должны выводиться на экран.

В приложении, которое мы строим, должны выводиться два текстовых поля, под которыми располагается кнопка. Код макета выглядит так, как показано ниже. Обновите файл *activity_main.xml* и включите в него изменения (выделенные жирным шрифтом):

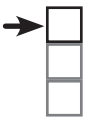
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />

</LinearLayout>
```

В реальном приложении для атрибутов hint и text использовались бы строковые ресурсы.





И это все представления, которые необходимы для нашего макета. Что же дальше?



Линейный макет
Фреймовый макет
Представление с прокруткой

Растяжение представлений

Все представления в базовом макете занимают столько вертикального пространства, сколько необходимо для их содержимого. Но мы хотим, чтобы текстовое поле Message растягивалось по вертикали и занимало в макете все вертикальное пространство, не используемое другими представлениями.

Для этого нужно назначить текстовому полю Message **весовой коэффициент**, или **вес**. Назначение весов — способ отдать команду представлению занять дополнительное пространство в макете.

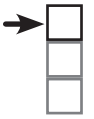
Для назначения веса представлению используется атрибут

```
android:layout_weight="число"
```

где `число` — некоторое положительное значение.

Если представлению назначен вес, макет прежде всего выделяет каждому представлению место, достаточное для вывода его содержимого: каждой кнопке хватает места для вывода ее текста, каждому текстовому полю — для вывода подсказок, и т. д. После этого все оставшееся пространство пропорционально распределяется между представлениями с весом 1 и более.

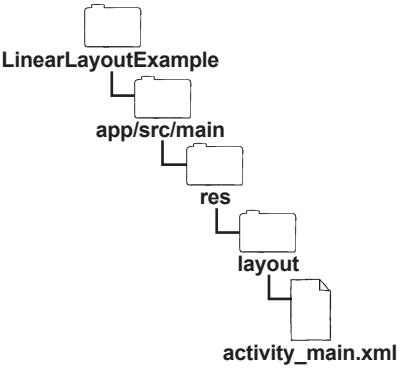
Давайте посмотрим, как сделать это в том макете, который мы строим.



Линейный макет
Фреймовый макет
Представление с прокруткой

Назначение веса одному представлению

Текстовое поле Message должно занимать все свободное место в макете, не используемое другими двумя представлениями. Для этого мы присвоим его атрибуту android:layout_weight значение 1. Так как это единственное представление в макете, которому назначен вес, текстовое поле растягивается по вертикали, занимая всю оставшуюся часть экрана. Разметка приведена ниже; обновите файл *activity_main.xml* (изменения выделены жирным шрифтом):



У элементов <EditText> и <Button> атрибут layout_weight не задан. Они занимают столько места, сколько необходимо для их содержимого, но не более.

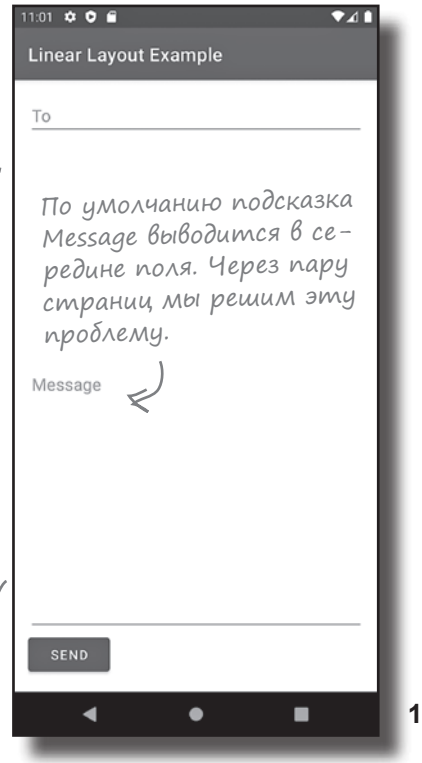
Единственное представление в макете, которому назначен вес. Представление заполняет все пространство, не занятое другими представлениями.

```
<LinearLayout ... >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

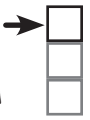
Высота представления в линейном макете определяется по значению layout_weight. Присваивание layout_height значения 0dp более эффективно, чем присваивание значения «wrap_content», так как Android не приходится выполнять вычисления для «wrap_content».

Присваивание текстовому полю веса 1 означает, что оно займет все свободное пространство, не занятое другими представлениями в макете. Причина заключается в том, что двум другим представлениям веса в разметке XML макета не назначены.

Представлению Message присвоен вес 1. Так как это единственное представление с заданным атрибутом веса, представление растягивается и занимает всё вертикальное пространство в макете.



В нашем примере вес необходимо назначить только одному представлению. Но прежде чем обновлять макет, посмотрим, что произойдет, если назначить веса нескольким представлениям.



Линейный макет
Фреймовый макет
Представление с прокруткой

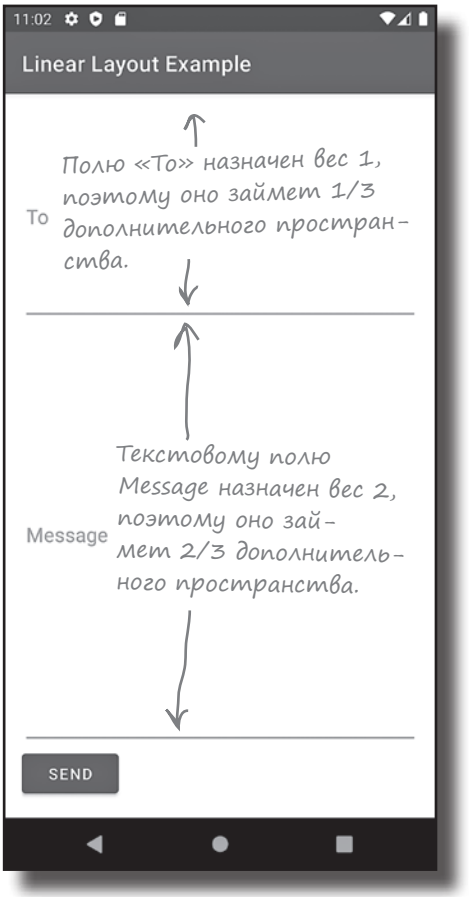
Назначение весов нескольким представлениям

Если веса назначены сразу нескольким представлениям, то линейный макет распределяет место между представлениями на основании весов, назначенных каждому представлению.

Допустим, текстовому полю «To» назначен вес 1, а текстовому полю Message назначен вес 2:

```
<LinearLayout ... >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="2"
        android:hint="Message"
        android:inputType="textMultiLine" />

       <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

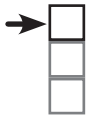
Это всего лишь пример; мы не будем изменять макет, чтобы он выглядел так.



Линейный макет замечает, что текстовым полям To и Message назначены веса, и на основании этих весов вычисляет, какую долю свободного места нужно выделить каждому из них.

Все начинается со сложения атрибутов layout_weight всех представлений. В нашем примере получится 1+2=3. Доля свободного пространства, занятого каждым представлением, будет равна весу представления, разделенному на сумму весов. Текстовому полю «To» назначен вес 1; следовательно, оно будет занимать 1/3 свободного пространства в макете. Текстовому полю Message назначен вес 2, поэтому оно займет 2/3 свободного пространства.

Теперь вы знаете, как использовать веса, и мы можем продолжить обновление макета.



Атрибут gravity и управление выравниванием текста в представлении

Следующая задача — переместить текст подсказки в текстовом поле Message. На данный момент он выводится в середине поля по вертикали. Нужно изменить разметку так, чтобы текст отображался у верхнего края. Эта задача решается с помощью атрибута **`android:gravity`**.

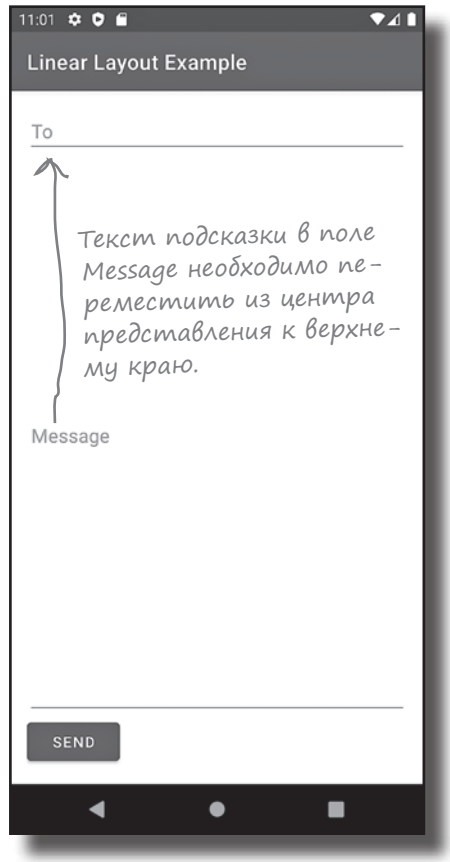
Атрибут `android:gravity` позволяет указать, как содержимое должно размещаться внутри представления, например, как текст должен позиционироваться в текстовом поле. Если вам нужно, чтобы текст выводился у верхнего края, можно присвоить атрибуту `android:gravity` значение «`top`»:

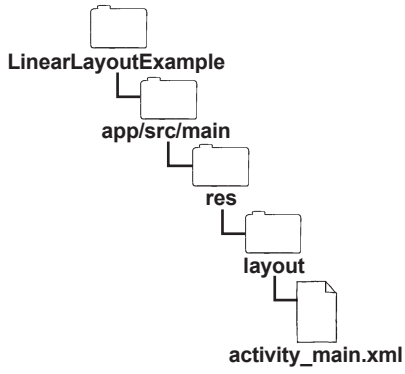
```
android:gravity="top"
```

Чтобы текст подсказки сместился в верхнюю часть текстового поля, следует включить в разметку поля атрибут `android:gravity`. Код приведен ниже; обновите файл *activity_main.xml* (изменения выделены жирным шрифтом):

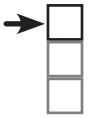
```
<LinearLayout ... >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

Внутренний текст должен выводиться у верхнего края текстового поля.





Список других возможных значений атрибута `android:gravity` приведен на следующей странице.



Линейный макет
Фреймовый макет
Представление с прокруткой

Список значений атрибута android:gravity

Ниже перечислены другие значения, которые могут использоваться с атрибутом `android:gravity`. Добавьте атрибут в представление и присвойте ему значение из следующего списка:

```
android:gravity="value"
```

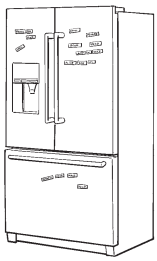
Значение	Что делает
top	Содержимое размещается у верхней стороны представления.
bottom	Содержимое размещается у нижней стороны представления.
start	Содержимое размещается у начальной стороны представления.
end	Содержимое размещается у конечной стороны представления.
center_vertical	Содержимое выравнивается по центру представления (по вертикали)
center_horizontal	Содержимое выравнивается по центру представления (по горизонтали)
center	Содержимое выравнивается по центру представления (по вертикали и по горизонтали)
fill_vertical	Содержимое заполняет представление по вертикали
fill_horizontal	Содержимое заполняет представление по горизонтали
fill	Содержимое заполняет представление по вертикали и по горизонтали.

Также можно назначить представлению несколько значений `android:gravity`, разделяя их символами «|». Например, чтобы сместить содержимое представления к правому нижнему углу, используйте следующее значение атрибута:

```
android:gravity="bottom|end"
```

Разобравшись с позиционированием содержимого представления, можно переходить к упражнению на следующей странице.

Атрибут android:gravity управляет размещением содержимого внутри представления.

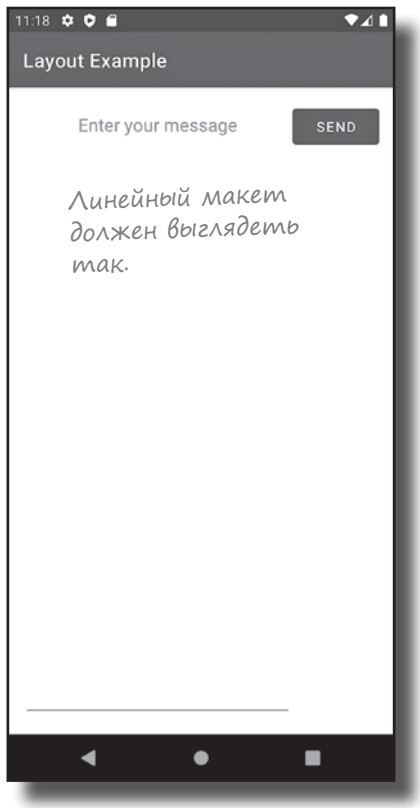


Развлечения с магнитами

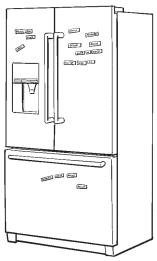
Кто-то выложил на холодильнике линейный макет, который создает приведенный ниже интерфейс. К сожалению, от сквозняка магниты упали на пол. Сможете ли вы снова собрать код метода?

Подсказка: использовать все фрагменты необязательно.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width=  .....

    android:layout_height=  .....

     .....
    android:padding="16dp"
    tools:context=".MainActivity">

    <EditText
        android:layout_width=  .....

        android:layout_height=  .....

        .....

        .....

        android:hint="Enter your message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width=  .....

        android:layout_height=  .....

        android:text="Send" />
</LinearLayout>
```



"vertical"
"horizontal"
"0dp"
"top|start"
"wrap_content"
"match_parent"
android:orientation
android:gravity
android:layout_weight
"top|center_horizontal"
"1"
"wrap_content"
=
"match_parent"
=
=
android:weight
"match_parent"



Развлечения с магнитами. Решение

Кто-то выложил на холодильнике линейный макет, который создает приведенный ниже интерфейс. К сожалению, от сквозняка магниты упали на пол. Сможете ли вы снова собрать код метода?

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width=  "match_parent"
    android:layout_height=  "match_parent"
    android:orientation  =  "horizontal"
    android:padding="16dp"
    tools:context=".MainActivity">

    <EditText
        android:layout_width=  "0dp"
        android:layout_height=  "match_parent"
        android:layout_weight  =  "1"
        android:gravity  =  "top|center_horizontal"
        android:hint="Enter your message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width=  "wrap_content"
        android:layout_height=  "wrap_content"
        android:text="Send" />
</LinearLayout>
```

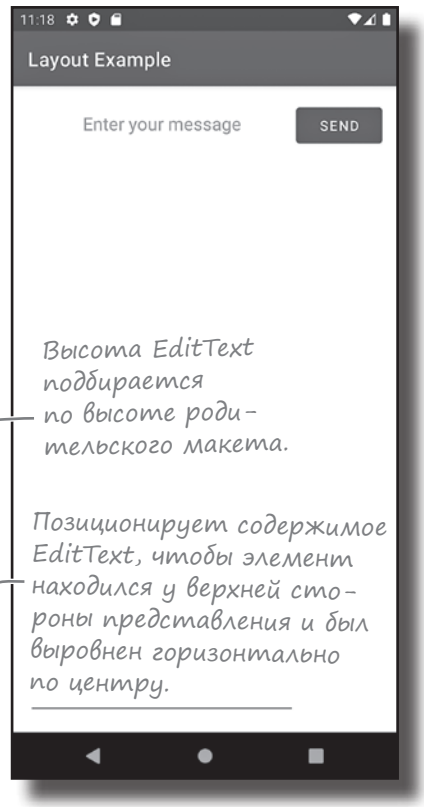
Представления выводятся по горизонтали в строку.

Использовать все горизонтальное пространство, не используемое Button.

Высота EditText подбирается по высоте родительского макета.

Позиционирует содержимое EditText, чтобы элемент находился у верхней стороны представления и был выровнен горизонтально по центру.

Размеры кнопки Send должны быть достаточными для ее содержимого.

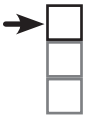


Эти магниты не понадобились.

"top|start"

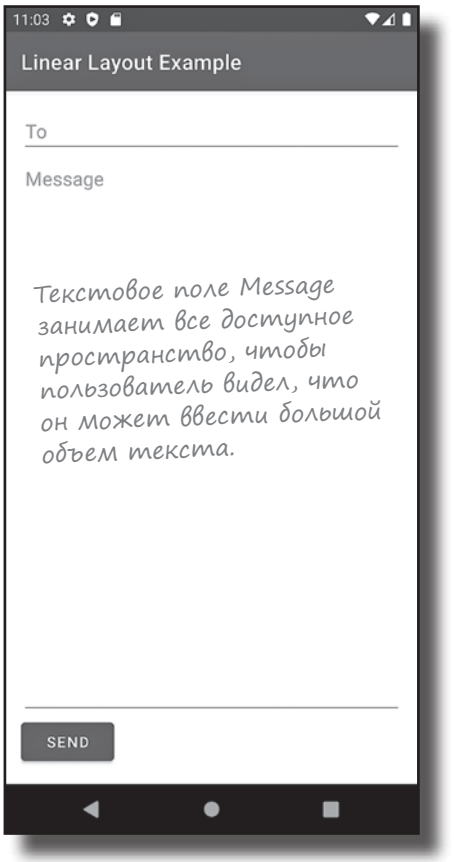
android:weight

"vertical"



К настоящему моменту

К настоящему моменту мы добавили в линейный макет три представления и отрегулировали их позиции, добавив атрибуты `layout_weight` и `gravity` в текстовое поле Message. Эти атрибуты означают, что текстовое поле захватывает все дополнительное пространство, не используемое другими представлениями, а его текст подсказки выводится в верхней части представления:

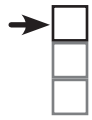


Линейный макет почти завершен, осталось внести пару последних изменений.

1. **Кнопка Send перемещается к конечной стороне.**
   Для языков с написанием слева направо кнопка перемещается вправо.

2. **Между текстовым полем Message и верхней стороной кнопки создается дополнительный промежуток.**

Давайте посмотрим, как это делается, начиная со смещения кнопки к конечной стороне.



Линейный макет
Фреймовый макет
Представление с прокруткой

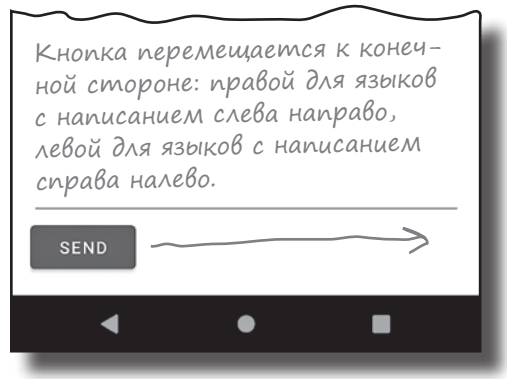
layout_gravity управляет положением представления в макете

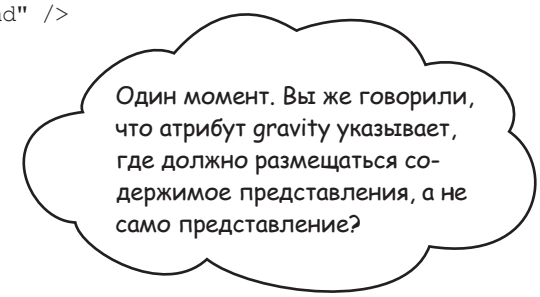
Чтобы переместить кнопку к конечной стороне макета, мы добавим к ней атрибут **`android:layout_gravity`**.

Атрибут `android:layout_gravity` позволяет указать, в какой части внешнего пространства должно находиться представление в линейном макете. Например, атрибут может использоваться для смещения представления вправо или для горизонтального выравнивания по центру.

Для смещения кнопки к конечной стороне макета атрибуту `android:layout_gravity` присваивается значение `"end"`:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end"
    android:text="Send" />
```





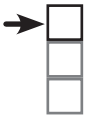
У линейных макетов есть два атрибута с похожими именами, gravity и layout_gravity.

Ранее атрибут `android:gravity` использовался для размещения текста подсказки Message в текстовом поле. Это объяснялось тем, что атрибут `android:gravity` указывает, где должно выводиться **содержимое** представления.

Атрибут `android:layout_gravity` относится к **размещению самого представления**. Он управляет тем, где представления выводятся во внешнем пространстве. В нашем случае представление должно сместиться к концу доступного пространства:

```
android:layout_gravity="end"
```

На следующей странице приведены некоторые допустимые значения атрибута `android:layout_gravity`.



Другие допустимые значения атрибута android:layout_gravity

Ниже приведена сводка некоторых возможных значений атрибута `android:layout_gravity`. Добавьте атрибут в свое представление и задайте ему одно из значений, перечисленных ниже:

```
android:layout_gravity="value"
```

Значение	Что делает
top, bottom, start, end	Размещает представление у верхней, нижней, начальной или конечной стороны контейнера.
center_vertical, center_horizontal	Выравнивает представление по вертикали или по горизонтали внутри контейнера.
center	Выравнивает представление по вертикали и по горизонтали внутри контейнера.
fill_vertical, fill_horizontal	Масштабирует представление так, чтобы оно заполняло доступное пространство в вертикальном или горизонтальном направлении.
fill	Масштабирует представление так, чтобы оно заполняло доступное пространство по вертикали и по горизонтали.

Также можно назначить представлению несколько значений `android:layout_gravity`, разделяя их символами «|». Например, чтобы сместить содержимое представления к правому нижнему углу доступного пространства, используйте следующее значение атрибута:

```
android:layout_gravity="bottom|end"
```

Итак, мы разобрались с изменением позиции представления с помощью атрибута `android:layout_gravity`. Теперь посмотрим, как добавить дополнительные интервалы между представлениями.

Атрибут android:layout_gravity позволяет указать, в какой части доступного пространства должно выводиться представление.

Атрибут android:layout_gravity определяет размещение самого представления, тогда как атрибут android:gravity определяет размещение содержимого представления.

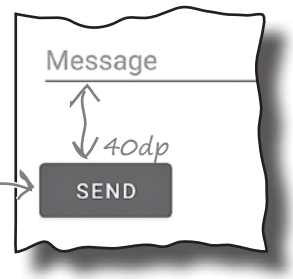
Разделение представлений свободными промежутками

Когда вы размещаете представления в линейном макете, между ними почти не остается свободного места. Вы можете увеличить размер свободного пространства вокруг представления, добавив к нему один или несколько **интервалов**.

Допустим, в линейном макете размещаются два представления — текстовое поле и находящаяся под ним кнопка. Если вы захотите увеличить пространство между двумя представлениями, добавьте интервал размером 40dp к верхней стороне кнопки при помощи атрибута `android:layout_marginTop`:

```
<LinearLayout...>
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="40dp"
        android:text="Send" />
</LinearLayout>
```

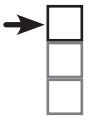
Добавление интервала к верхней стороне кнопки добавляет свободное пространство между двумя представлениями.



Ниже приведен список типов полей, которые вы можете использовать, чтобы придать вашим представлениям дополнительное пространство. Добавьте атрибут в представление и установите для его значения желаемый размер поля:

```
android:attribute="8dp"
```

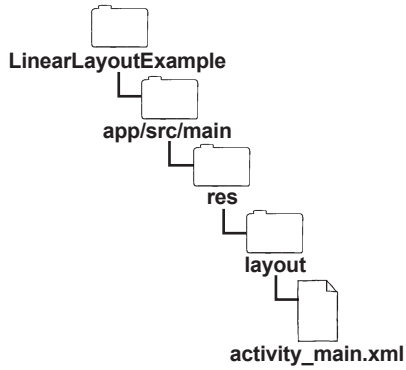
Атрибут	Что делает
layout_marginTop	Добавляет дополнительный интервал к верхней стороне представления.
layout_marginBottom	Добавляет дополнительный интервал к нижней стороне представления.
layout_marginStart	Добавляет дополнительный интервал к начальной стороне представления.
layout_marginEnd	Добавляет дополнительный интервал к конечной стороне представления.
layout_margin	Добавляет равные интервалы ко всем сторонам представления.
layout_marginVertical, layout_marginHorizontal	Добавляет равные интервалы к вертикальным (верхней и нижней) или горизонтальным (начальной и конечной) сторонам представления.

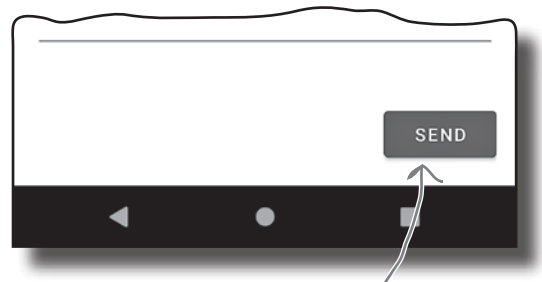


Полный код линейного макета

Теперь вы знаете, как использовать атрибуты представлений `layout_gravity` и `margin`. Воспользуемся ими в коде линейного макета, чтобы изменить позицию кнопки и добавить дополнительное пространство к ее верхней стороне. Ниже приведен полный код файла *activity_main.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"
        android:inputType="text" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:layout_marginTop="40dp"
        android:text="Send" />
</LinearLayout>
```





Перемещает кнопку в конец доступного пространства и создает над ней интервал 40dp.

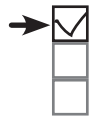
Код линейного макета готов, опробуем его на практике.

Тест-драйв

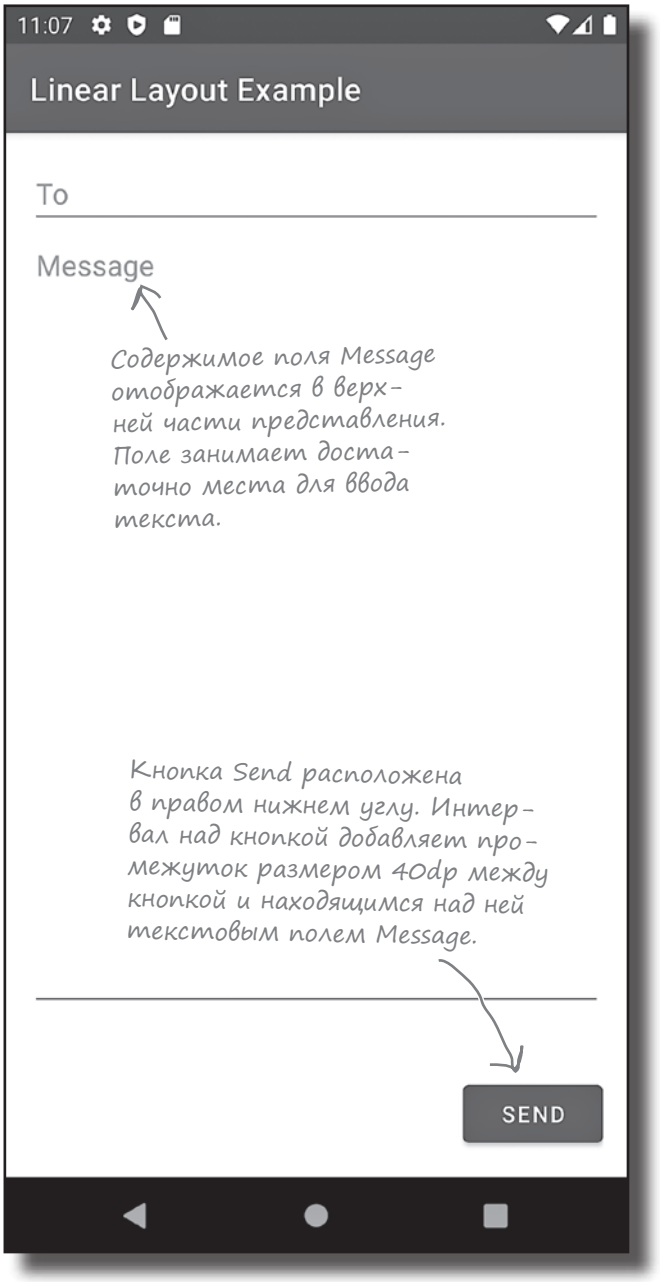
После внесения всех изменений запустите свое приложение.

Оно выводит линейный макет с тремя представлениями: двумя текстовыми полями и кнопкой. Представления выстроены в вертикальный столбец. Поле To находится сверху, кнопка Send — в правом нижнем углу, а представление Message занимает все свободное пространство, оставляя интервал 40dp у верхней стороны кнопки.

Теперь вы знаете, как создать линейный макет и управлять выводом его представлений. Проверьте свои силы, выполнив упражнение на следующей странице.



Линейный макет
Фреймовый макет
Представление с прокруткой



Часто Задаваемые Вопросы

В: Нельзя ли использовать атрибут `paddingTop` с кнопкой вместо `layout_marginTop`? Разве это не приведет к тому же эффекту?

О: Атрибуты отступов (такие, как `paddingTop`) могут использоваться с представлениями, но они приводят к другому эффекту. Как вы уже видели, `layout_marginTop` увеличивает величину промежутка *над* представлением, тогда как `paddingTop` увеличивает высоту представления, добавляя дополнительное пространство *внутри него*:

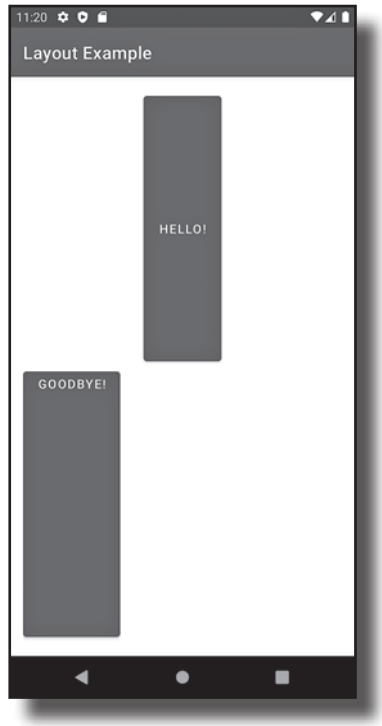
padding добавляет дополнительное пространство внутри представления.

Стань макетом

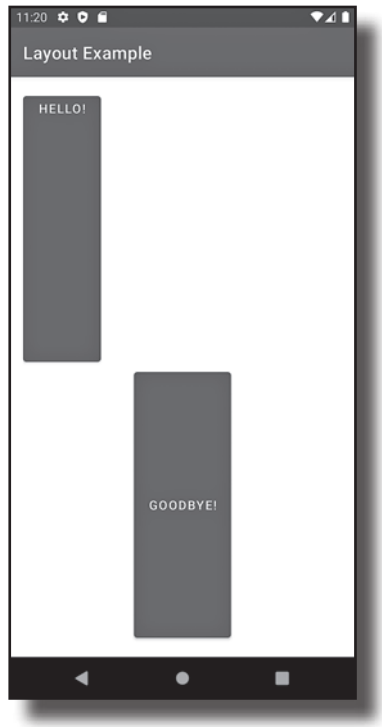
Приведенный ниже код описывает завершенный линейный макет. Представьте себя на месте макета и определите, какой экран (A или B) будет получен при выводе этого макета.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_gravity="center_horizontal"
        android:text="Hello!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="center_horizontal"
        android:text="Goodbye!" />

</LinearLayout>
```

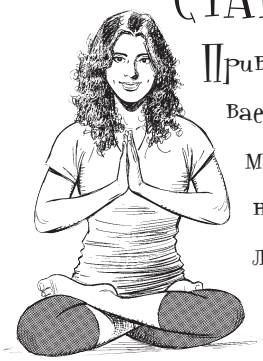
A



B



СТАНЬ макетом. Решение

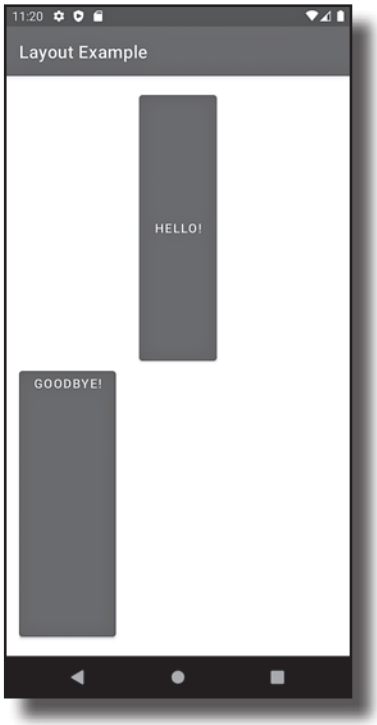


Приведенный ниже код описывает завершенный линейный макет. Представьте себя на месте макета и определите, какой экран (A или B) будет получен при выводе этого макета.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_gravity="center_horizontal"
        android:text="Hello!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="center_horizontal"
        android:text="Goodbye!" />

</LinearLayout>
```

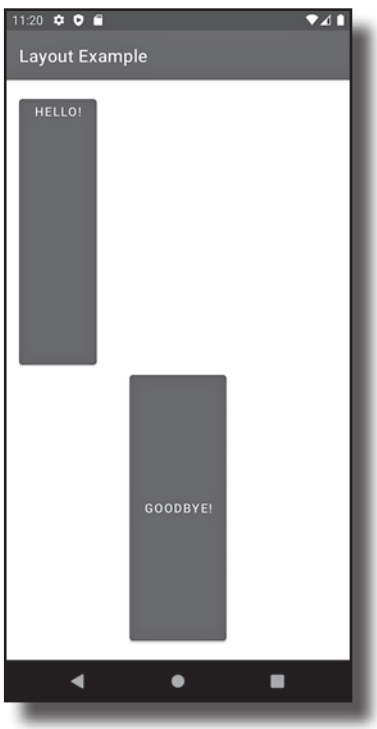
A ✓

Код макета создает экран A. Кнопка «Hello!» выравнивается горизонтально по центру, а текст кнопки «Goodbye!» выравнивается горизонтально по центру в представлении.



B ✗

Чтобы получить экран B, нужно поменять местами свойства gravity и layout_gravity.



Код активности сообщает Android, какой макет он использует

В этой главе вы научились использовать линейный макет и настраивать параметры вывода его представлений. Прежде чем знакомить вас со следующей разновидностью макетов, давайте заглянем под капот и посмотрим, что происходит с макетом при запуске приложения.

Как вы уже знаете, при запуске приложения Android запускает главную активность приложения. В текущей версии это активность с именем `MainActivity`.

При запуске активности выполняется метод `onCreate()`. Этот метод включает следующий код, который указывает, какой макет должен использоваться активностью:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
}
```

← *Дает команду активности использовать файл макета activity_main.xml.*

Чтобы сообщить Android, какой макет следует использовать, этот пример передает имя макета методу с именем `setContentView()`. Затем метод выводит макет на экран.

Представления макета преобразуются в объекты

Кроме вывода макета на экране устройства, метод `setContentView()` преобразует элементы XML макета в объекты. Этот процесс называется **заполнением макета**.

Заполнение макета играет важную роль, потому что оно позволяет вашему коду макета оперировать с представлениями в макете. Во внутренней реализации каждое представление преобразуется в объект, с которым вы можете взаимодействовать из кода активности.

Посмотрим, как заполнение макета работает с кодом линейного макета, который мы построили.

Когда вы запускаете свое приложение, Android обрабатывает разметку XML макета и преобразует каждый элемент макета в объект. Этот процесс называется заполнением макета.

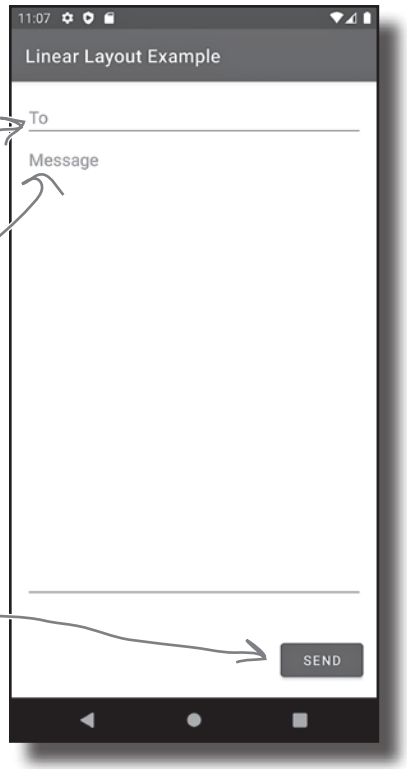
- [x] **Линейный макет**
- [ ] **Фреймовый макет**
- [ ] **Представление с прокруткой**

Заполнение макета: пример

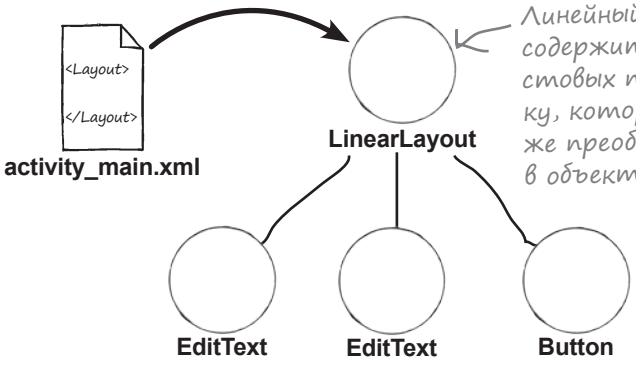
Как вы уже знаете, в нашем примере выводятся два текстовых поля и кнопка внутри линейного макета:

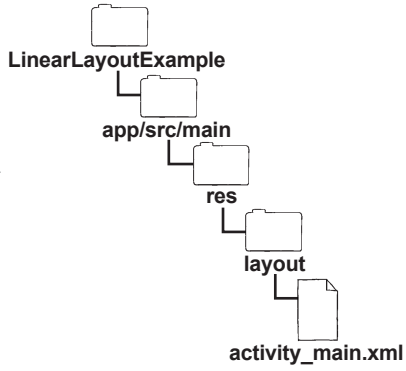
Некоторые подробности были опущены, но основная структура кода выглядит так.

```
<LinearLayout ... >

    <EditText
        ...
        android:hint="To"
        android:inputType="text" />

    <EditText
        ...
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        ...
        android:text="Send" />

</LinearLayout>
```

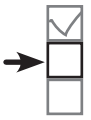


При запуске приложения представления в макете заполняются и преобразуются в объекты. Линейный макет преобразуется в объект `LinearLayout`, текстовые поля преобразуются в объекты `EditText`, а кнопка преобразуется в объект `Button`:





А теперь посмотрим, как использовать новую разновидность макетов: **фреймовый макет**.



Фреймовый макет накладывает предоставления друг на друга

Как вы уже знаете, линейный макет выстраивает свои представления в одну строку или столбец. Каждому представлению выделяется собственный участок экрана, и они не перекрывают друг друга.

Но иногда представления *должны* перекрываться. Например, представьте, что вы хотите вывести изображение, наложив поверх него текст. Сделать это только с линейным макетом не удастся.

Если вам нужен макет, в котором представления могут перекрываться, проще всего воспользоваться фреймовым макетом. Вместо вывода представлений в одну строку или в столбец они накладываются друг на друга. Часто в этом макете выводится только одно представление.

Для примера создадим полезное приложение, которое выводит текст поверх изображения утки. Начнем с создания нового проекта.

Мы воспользуемся фреймовым макетом для наложения текстового поля на изображение утки.

Создание нового проекта

Создайте новый проект Android Studio так же, как это делалось ранее. Выберите вариант Empty Activity, введите имя «Frame Layout Example» с именем пакета «com.hfad.framelayoutexample» и подтвердите место сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android.

После создания проекта можно переходить к определению фреймового макета.

Как определить фреймовый макет

Фреймовый макет определяется элементом `<FrameLayout>`:

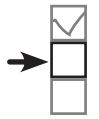
Элемент <FrameLayout> используется для определения фреймового макета.

```
<FrameLayout "xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</FrameLayout>
```

Те же атрибуты, которые использовались для линейного макета. Они обязательны для всех макетов и представлений.

Как и у линейны макетов и других типов представлений и групп представлений, атрибуты `android:layout_width` и `android:layout_height` задают ширину и высоту макета; они являются обязательными. Также при желании можно добавить атрибуты для определения отступов, хотя здесь мы этого не делали.

Приведенный выше код создает пустой фреймовый макет. Сделаем следующий шаг и добавим в него изображение утки.



Линейный макет
Фреймовый макет
Представление с прокруткой

Добавление изображений в проект

Во фреймовом макете будет выводиться изображение *duck.webp*, но сначала нужно добавить файл в проект.

Для этого создайте папку для ресурсов с именем *drawable* (если среда Android Studio еще не создала ее за вас). Эта папка используется по умолчанию для хранения графических ресурсов вашего приложения. Переключитесь в представление Project на панели Explorer среды Android Studio, выберите папку *app/src/main/res*, откройте меню File, выберите команду New... и щелкните на варианте создания нового каталога ресурсов Android. Затем выберите тип ресурса «drawable», введите имя папки «drawable» и щелкните на кнопке OK.

Загрузите файл *duck.webp* по адресу *tinyurl.com/hfad3*, добавьте его в папку *app/src/main/res/drawable*.

Файл *duck.webp* будет выводиться в графическом представлении (представление для вывода графических изображений), которое мы добавим во фреймовый макет. Графическое представление определяется элементом `<ImageView>`:

Серьезное программирование

В Android для работы с графикой обычно используется формат **WebP**, сочетающий небольшой размер файла с минимальной потерей качества.

Изображения можно преобразовать в формат WebP в Android Studio. Для этого щелкните правой кнопкой мыши на изображении и выберите команду «Convert to WebP».

Добавляет представление для вывода изображения.

Использовать изображение с именем «duck» из папки drawable.

Края изображения обрезаются по границам доступного пространства.

Добавляет описание содержимого для улучшения доступности приложения. В реальном приложении этот текст был бы добавлен в виде строкового ресурса.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout ...>

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/duck"
        android:contentDescription="Duck image" />

</FrameLayout>
```

Элемент `<ImageView>` включает уже знакомые вам атрибуты `android:layout_width` и `android:layout_height`, а также три новых атрибута.

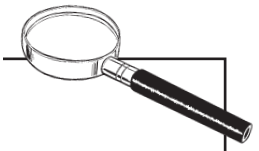
Атрибут `android:src` задает изображение, которое должно отображаться в графическом представлении. Мы присвоили этому атрибуту значение `"@drawable/duck"`, чтобы выводился файл *duck.webp* из папки *drawable*.

Атрибут `android:contentDescription` предоставляет текстовое описание изображения для улучшения доступности.

Наконец, атрибут `android:scaleType` описывает способ масштабирования изображения. В нашем случае используется значение `"centerCrop"`, которое обрезает края изображения.

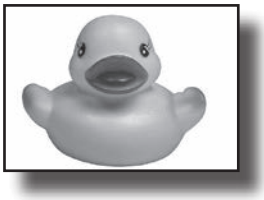
И это все, что необходимо знать для вывода изображения утки во фреймовом макете. Прежде чем вы узнаете, как вывести текст поверх изображения, поближе познакомимся с использованием *графических ресурсов*.

Графические ресурсы под увеличительным стеклом



Как вы уже узнали, чтобы добавить изображение в проект, следует поместить его в папку *app/src/main/res/drawable*. Тем самым вы включаете изображение в свой проект в виде графического ресурса, для отображения которого в макете используется код следующего вида:

```
<ImageView
    android:id="@+id/image"
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:src="@drawable/duck"
    android:contentDescription="Duck image" />
```



Добавляя файл duck.webp в папку drawable, вы включаете его в свой проект как графический ресурс.

При желании можно использовать разные версии изображения для разной плотности пикселей экрана. Это позволит выводить изображения с большим разрешением на устройствах с большей плотностью пикселей и изображения с меньшим разрешением на экранах меньшей плотности.

Для этого следует сначала создать разные папки *drawable* в *app/src/main/res* для разных вариантов плотности экрана. Каждая папка имеет суффикс, определяющий конкретную плотность экрана:

`drawable-ldpi`	Экраны низкой плотности, около 120 dpi.
`drawable-mdpi`	Экраны средней плотности, около 160 dpi.
`drawable-hdpi`	Экраны высокой плотности, около 240 dpi.
`drawable-xhdpi`	Экраны сверхвысокой плотности, около 320 dpi.
`drawable-xxhdpi`	Экраны сверхсверхвысокой плотности, около 480 dpi.
`drawable-xxxhdpi`	Экраны сверхсверхсверхвысокой плотности, около 640 dpi.
`drawable-nodpi`	Изображения из этой папки могут использоваться на экранах любой плотности, но изображение не масштабируется.

В зависимости от используемой версии Android Studio IDE может создать некоторые из этих папок автоматически.

Затем изображения с разными разрешениями добавляются в каждую из папок *drawable* — проследите за тем, чтобы все файлы имели одинаковые имена. Android выбирает используемую версию изображения на стадии выполнения в зависимости от плотности экрана устройства, на котором выполняется приложение. Например, если устройство оснащено экраном сверхвысокой плотности, будет использоваться изображение из папки *drawable-xhdpi*.

Если изображение добавлено только в одну из этих папок, то Android использует один и тот же файл изображения для всех устройств. Если вы хотите, чтобы одно изображение использовалось независимо от плотности экрана, обычно оно помещается в папку *drawable*.

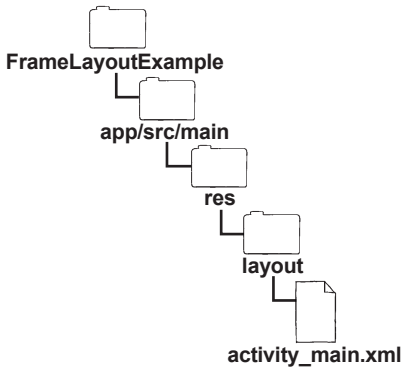
Линейный макет
Фреймовый макет
Представление с прокруткой

Фреймовый макет накладывает изображения в порядке их перечисления в разметке XML макета

При определении фреймового макета представления перечисляются в том порядке, в каком они должны накладываться друг на друга. Сначала выводится первое представление, затем на него накладывается второе, и т. д.

В данном примере мы собираемся наложить надпись на графическое представление, так что в разметке XML она должна следовать после графического представления. Обновите файл *activity_main.xml*, чтобы он выглядел так, как показано ниже:

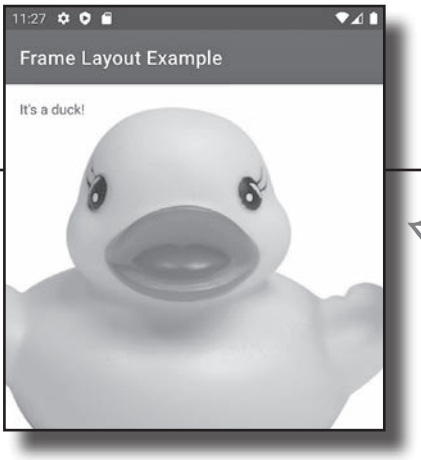
```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/duck"
        android:contentDescription="Duck image" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="It's a duck!" />
</FrameLayout>
```



Добавляет надпись.

Тест-драйв

При запуске приложения на устройстве появляется изображение утки с текстом «It's a duck!» в левом верхнем углу.

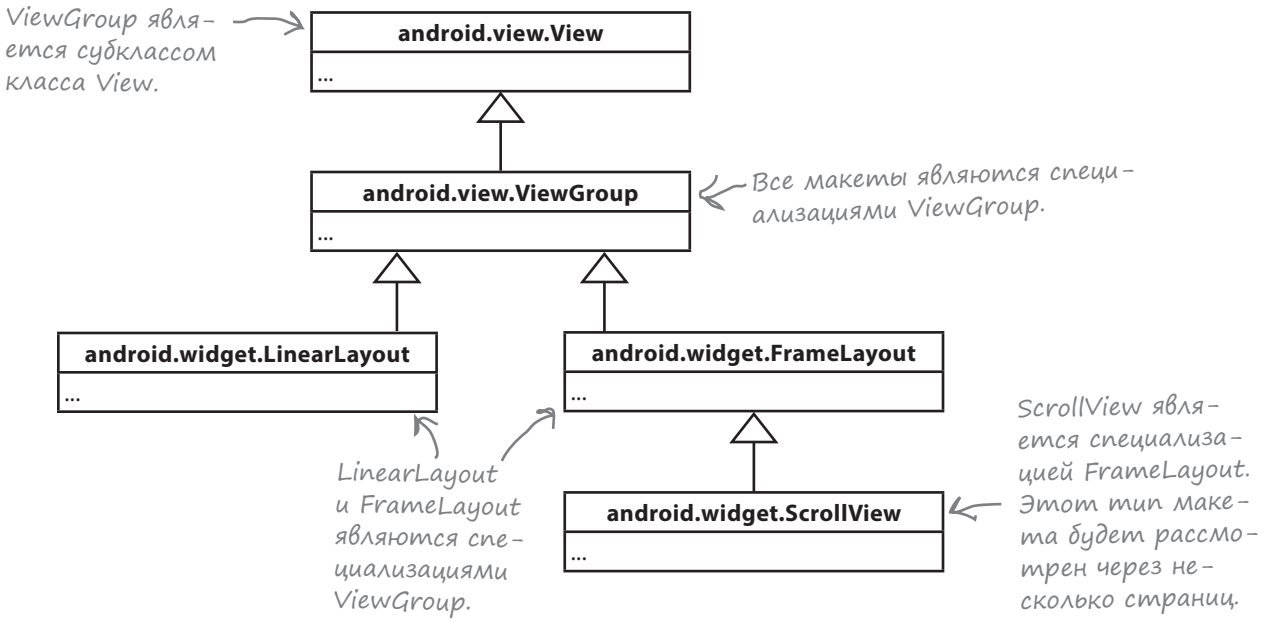


Обратите внимание: текст «It's a duck!» наложен на изображение.

Все макеты являются специализациями ViewGroup...

И хотя все они выводят свои представления по-разному, вероятно, вы заметили, что у линейных и фреймовых макетов много общего. Например, оба могут содержать представления, и каждый макет имеет свою политику, которая определяет способ вывода представлений.

И у этого сходства есть веская причина. Во внутренней реализации все макеты, включая линейные и фреймовые, являются субклассами для класса `android.view.ViewGroup`:



...а ViewGroup является специализацией View

Класс `ViewGroup` является субклассом класса `View`, который может содержать другие представления. Во внутренней реализации каждый макет является субклассом `ViewGroup`, это означает, что **каждый макет также является специализацией `View`**.

Такая иерархия классов означает, что все представления и макеты обладают общими атрибутами и поведением. Например, все они могут отображаться на экране, и вы можете определить их высоту или ширину. Именно поэтому вы должны задавать значения атрибутов `android:layout_height` и `android:layout_width` всех макетов. Эти атрибуты обязательны для всех представлений, а поскольку макет является разновидностью представления, эти атрибуты обязательны и для всех макетов.

Каждый компонент графического интерфейса, добавляемый в макет, является специализацией View: объекта, занимающего пространство на экране.

Каждый макет является специализацией ViewGroup: разновидности View, которая может содержать другие представления View.

Линейный макет
Фреймовый макет
Представление с прокруткой

Все верно.

Как вы уже знаете, макет является специализацией представления, которая может содержать другие представления. Так как каждый макет является специализацией представления, это означает, что **макеты могут содержать другие макеты**.

Возможность вложения макетов в другие макеты полезна, потому что она позволяет строить более сложные графические интерфейсы. Например, можно добавить горизонтальные строки в вертикальный линейный макет, вкладывая горизонтальные линейные макеты в корневой линейный макет. А если вы хотите расположить текст по вертикали над изображением, можно вложить вертикальный линейный макет во фреймовый макет. Давайте поближе присмотримся к тому, как это делается, на примере новой разновидности макетов: представления с прокруткой.

Часто Задаваемые Вопросы

В: Ранее вы говорили, что я могу вывести изображение в макете при помощи графического представления. А если понадобится добавить изображение к кнопке? Это вообще возможно?

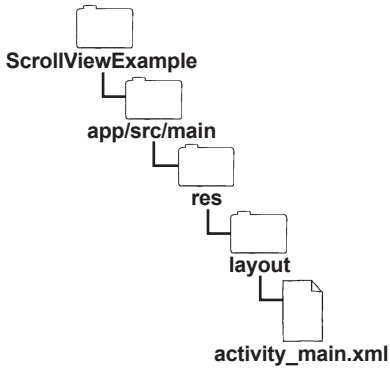
О: Да. Кнопки могут включать такие атрибуты, как `android:drawableStart` и `android:drawableEnd`, что позволяет разместить изображение в начале или в конце кнопки. Например, следующий код добавляет изображение утки в начале кнопки:

```
android:drawableStart="@drawable/duck"
```

Android также включает представление графической кнопки: кнопки с изображением, но без текста. В этой книге мы не будем рассматривать графические кнопки, но если они вас заинтересуют, дополнительную информацию можно найти по адресу *https://developer.android.com/guide/topics/ui/controls/button*

Представление с прокруткой добавляет вертикальную полосу прокрутки

Представление с прокруткой — разновидность фреймового макета с вертикальной полосой прокрутки. Оно пригодится для макетов, слишком больших для устройств, на которых они выполняются, так как с помощью полосы прокрутки можно перейти к любым представлениям, не помещающимся на экране.

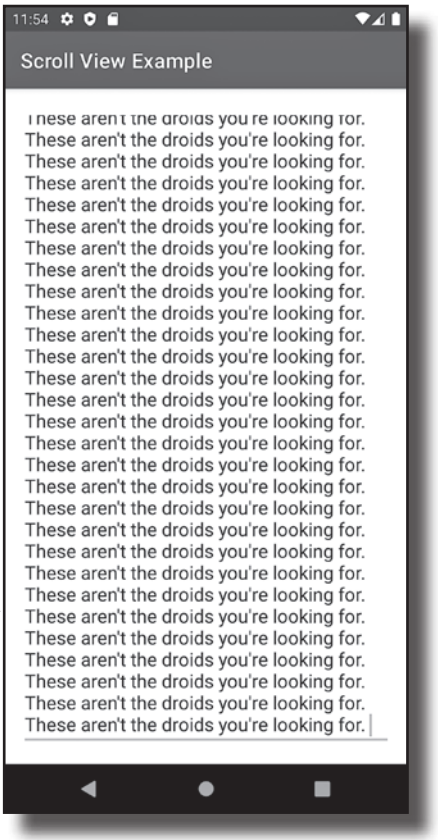


Мы создадим приложение, использующее представление с прокруткой. Создайте новый проект с именем «Scroll View Example» и выберите вариант Empty Activity. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android. Затем замените код в файле *activity_main.xml* следующим кодом:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity" >

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send" />
</LinearLayout>
```

Проследите за тем, чтобы код activity_main.xml совпадал с приведенным ниже.

Присвойте значение «textMultiLine», чтобы можно было ввести несколько строк текста.



Текстовое поле содержит столько текста, что кнопка вышла за край экрана.

Этот интерфейс состоит из простого линейного макета, который содержит текстовое поле Message и кнопку Send.

Попробуйте запустить приложение и ввести побольше текста в текстовом поле. Представление расширяется по размерам содержимого, и через какое-то время кнопка Send выйдет за край экрана. Полосы прокрутки нет, поэтому кнопка становится недоступной.

Чтобы решить эту проблему, мы добавим представление с прокруткой к макету. Оно создает полосу прокрутки, при помощи которой вы сможете обратиться к кнопке Send, если она не видна на экране.

Как добавить представление с прокруткой

Линейный макет
Фреймовый макет
Представление с прокруткой

Чтобы добавить представление с прокруткой, включите в макет элемент **<ScrollView>**. Элемент <ScrollView> используется так же, как и <FrameLayout>, не считая того, что он включает дополнительный атрибут **fillViewport**, который указывает, должно ли представление с прокруткой заполнять экран устройства.

ScrollView является субклассом FrameLayout. Это означает, что он может делать все, что может делать FrameLayout, и что-то еще.

В приведенном ниже коде мы добавили представление с прокруткой в код макета и включили в него исходный линейный макет. Обновите файл *activity_main.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<~~LinearLayout~~ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ~~android:orientation="vertical"~~
    ~~android:padding="16dp"~~
    android:fillViewport="true"
    tools:context=".MainActivity" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp" >

        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Message"
            android:inputType="textMultiLine" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Send" />
    </LinearLayout>
</ScrollView>
```

Замените элемент LinearLayout в корне макета элементом ScrollView.

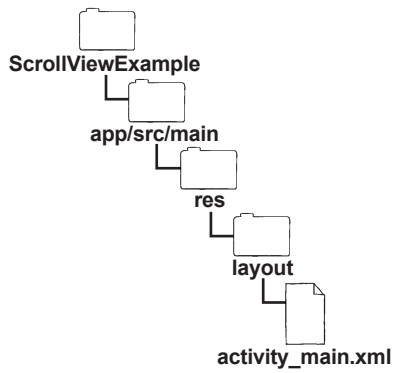
Удалите эти строки. Атрибут orientation не применяется к представлениям с прокруткой, и отступы для этого представления не нужны, так как они приведут к смещению полосы прокрутки.

Атрибуту fillViewport присваивается значение «true», чтобы он заполнял экран устройства.

Переместите исходный линейный макет, чтобы он был вложен в представление с прокруткой.

Проследите за тем, чтобы в вашем коде присутствовала эта строка.

Добавьте закрывающий тег для элемента <ScrollView>.



Вскоре мы посмотрим, как работает этот код, но сначала проверьте свои силы на следующем упражнении.

Возьмите в руку карандаш

Как вы уже знаете, макеты можно вкладывать друг в друга для формирования более сложных макетов. Руководствуясь этими знаниями, попробуйте рассказать или нарисовать, как бы вы построили изображенный ниже макет.

Мы не ожидаем, что вы напишете весь код макета. Просто расскажите, как бы вы его построили.

Макет состоит из текстового поля и двух кнопок под ним.

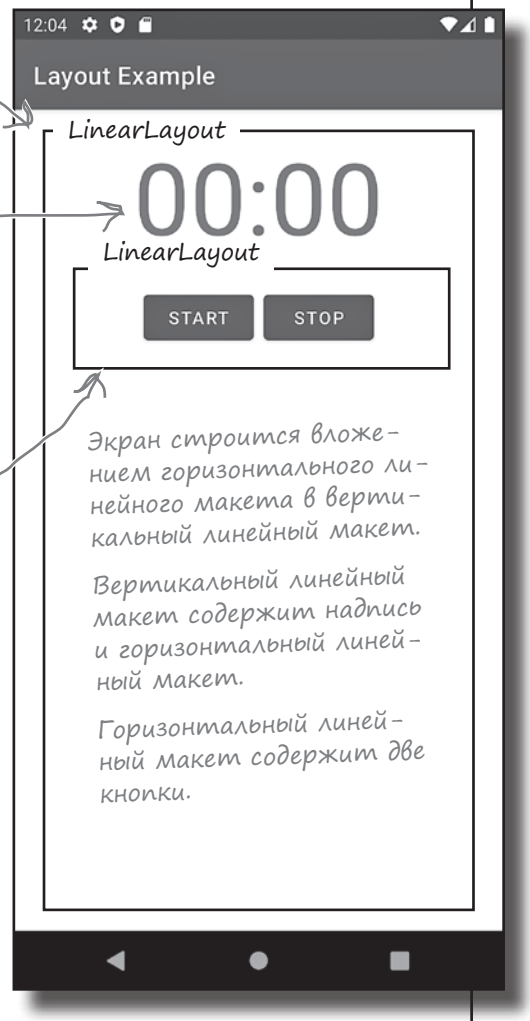
Возьмите в руку карандаш
Решение

Как вы уже знаете, макеты можно вкладывать друг в друга для формирования более сложных макетов. Руководствуясь этими знаниями, попробуйте рассказать или нарисовать, как бы вы построили изображенный ниже макет.

Мы не ожидаем, что вы напишете весь код макета. Просто расскажите, как бы вы его построили.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textSize="80sp" />
        android:text="00:00" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_marginTop="20dp"
        android:layout_gravity="center">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginHorizontal="4dp"
            android:text="Start" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginHorizontal="4dp"
            android:text="Stop" />
    </LinearLayout>
</LinearLayout>
```

Атрибут textSize используется для увеличения размера текста.



Экран строится вложением горизонтального линейного макета в вертикальный линейный макет.

Вертикальный линейный макет содержит надпись и горизонтальный линейный макет.

Горизонтальный линейный макет содержит две кнопки.

Тест-драйв

Линейный макет
Фреймовый макет
Представление с прокруткой

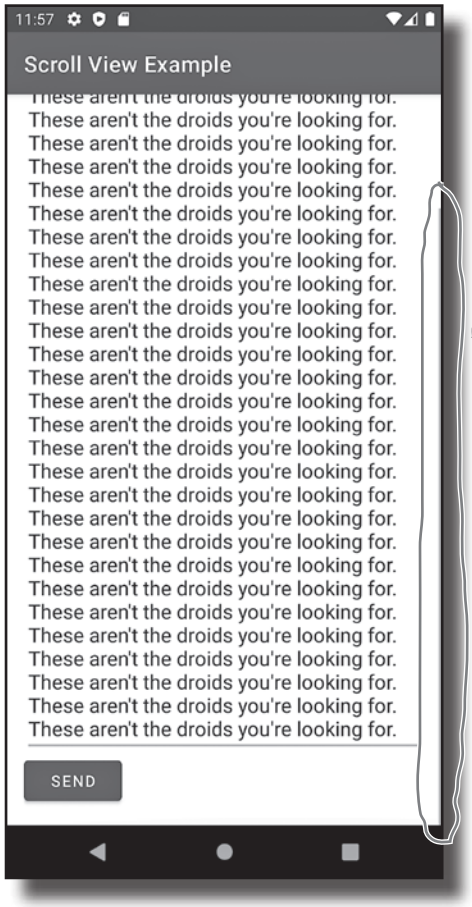
После того как представление с прокруткой будет включено в код разметки, запустите приложение.

Когда в текстовое поле Message будет введен достаточный объем текста, кнопка Send выходит за край экрана, как и прежде. Однако на этот раз вы можете прокрутить экран устройства, чтобы получить доступ к кнопке.

Поздравляем! Вы научились использовать разные типы макетов для управления внешним видом графического интерфейса вашего приложения. Эти знания станут отправной точкой для следующей главы.

Все макеты (включая линейные макеты, фреймовые макеты и представления с прокруткой) могут содержать другие макеты.

Вы можете строить сложные макеты, вкладывая макет одного типа внутрь другого.



При прокрутке экрана появляется вертикальная полоса прокрутки.

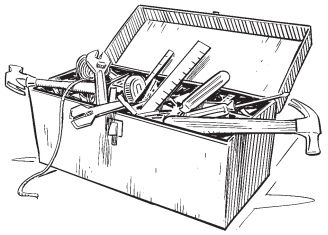
Часто Задаваемые Вопросы

В: Представление с прокруткой добавляет только вертикальную полосу прокрутки? А если мне понадобится горизонтальная полоса прокрутки?

О: Да, `<ScrollView>` добавляет только вертикальную полосу прокрутки, но не горизонтальную. Если вы захотите добавить в макет горизонтальную полосу прокрутки, для этого можно воспользоваться элементом `<HorizontalScrollView>`. Горизонтальное представление с прокруткой работает так же, как и вертикальное, но добавляет горизонтальную полосу прокрутки вместо вертикальной.

ГЛАВА 3

Ваш инструментарий Android



Глава 3 осталась позади, а ваш инструментарий пополнился средствами построения макетов.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Элемент `<EditText>` определяет текстовое поле с возможностью редактирования.
- Элемент `<ImageView>` используется для вывода графики.
- Линейный макет размещает представления по горизонтали или по вертикали. Направление задается атрибутом `android:orientation`.
- В линейном макете атрибут `android:layout_weight` используется для распределения свободного места в макете.
- Атрибуты `android:padding` используются для создания отступов между сторонами представления и его содержимым.
- Атрибут `android:layout_gravity` указывает, где представления должны отображаться в пределах свободного пространства.
- Атрибут `android:gravity` указывает, где содержимое должно выводиться внутри приложения.
- Атрибут `android:layout_margin` создает дополнительные интервалы вокруг представления.
- Фреймовый макет накладывает представления друг на друга.
- Для добавления полос прокрутки используется представление с прокруткой или горизонтальное представление с прокруткой. Оба являются разновидностями фреймового макета.
- Все макеты являются субклассами класса `android.view.ViewGroup`. Группа представлений является специализацией представления, которая может содержать несколько других представлений.
- Макеты могут включать другие макеты.

Построение эскиза

Дом невозможно построить без плана. А некоторые макеты строятся по **эскизам,** чтобы они выглядели **в точности так, как вам нужно**. В этой главе представлены **ограничения в макетах** Android: **гибкий механизм проектирования более сложных графических интерфейсов.** Вы узнаете, как **ограничения** и **смещения** используются для позиционирования и определения размеров ваших представлений **независимо от размера и ориентации экрана**. Вы узнаете, как закрепить представления в положенных местах при помощи **направляющих** и **барьеров**. Наконец, вы научитесь группировать или распределять представления с использованием **цепочек** и **потоков**. За дело!

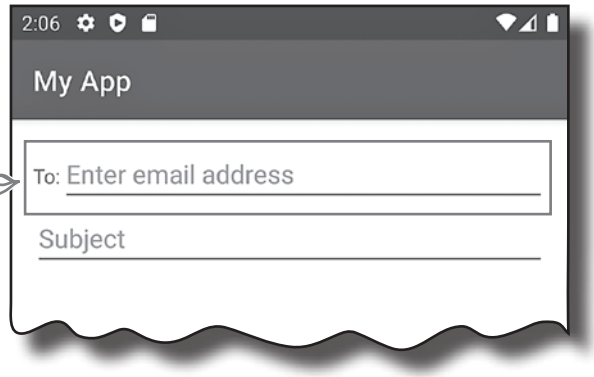
Снова о вложенных макетах

В предыдущей главе вы узнали, что вложение макетов может применяться для создания более сложных экранов. Например, следующий код использует вложенные макеты, чтобы вывести надпись и текстовое поле рядом по горизонтали и текстовое поле под ними:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="To:" />

        <EditText
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:hint="Enter email address"
            android:inputType="text" />
    </LinearLayout>

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Subject"
        android:inputType="text" />
</LinearLayout>
```

Используется вертикально ориентированный линейный макет.

Вложенный линейный макет размещает надпись и текстовое поле по горизонтали.



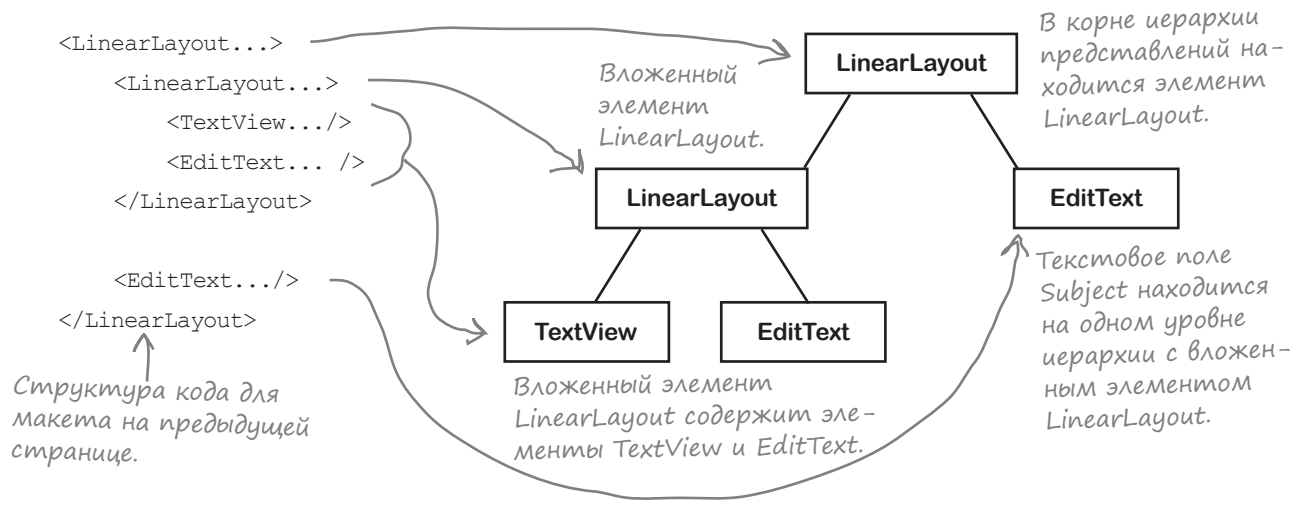
Ширина надписи выбирается минимально достаточной для его содержимого. Текстовое поле использует весовой коэффициент для захвата всего оставшегося горизонтального пространства.

Текстовое поле Subject выводится под вложенным линейным макетом.

Вложение макетов не дается бесплатно

У вложения макетов есть и обратная сторона: сложные макеты, построенные этим способом, могут быть неэффективными, возникает больше сложностей с чтением и сопровождением кода, а приложение может медленнее работать.

Когда Android выводит макет на экране устройства, система сначала проверяет структуру файла макета и использует ее для построения иерархии представлений. Например, для вложенного макета, приведенного на предыдущей странице, строится иерархия представлений с двумя линейными макетами, двумя текстовыми полями и надписью:



Android использует иерархию представлений для определения того, в каком месте экрана устройства должно размещаться каждое представление. Каждое представление необходимо измерить, разместить и вывести на экран, а система Android должна позаботиться о том, чтобы каждому представлению хватало места для его содержимого, и при размещении должны быть учтены все веса.

Если макет содержит вложенные макеты, иерархия представлений усложняется. Android может потребоваться выполнить несколько проходов для определения того, как должны размещаться представления. Глубокая иерархия макетов может привести к появлению узких мест в вашем коде, а у вас возникнут трудности с чтением и сопровождением кода.

Если в вашем приложении используется сложный графический интерфейс такого рода, вместо вложенных макетов можно воспользоваться **макетом с ограничениями**.

Каждое представление в макете нужно инициализировать, измерить, разместить и вывести на экран. В макетах с большим уровнем вложенности это может замедлить работу приложения.

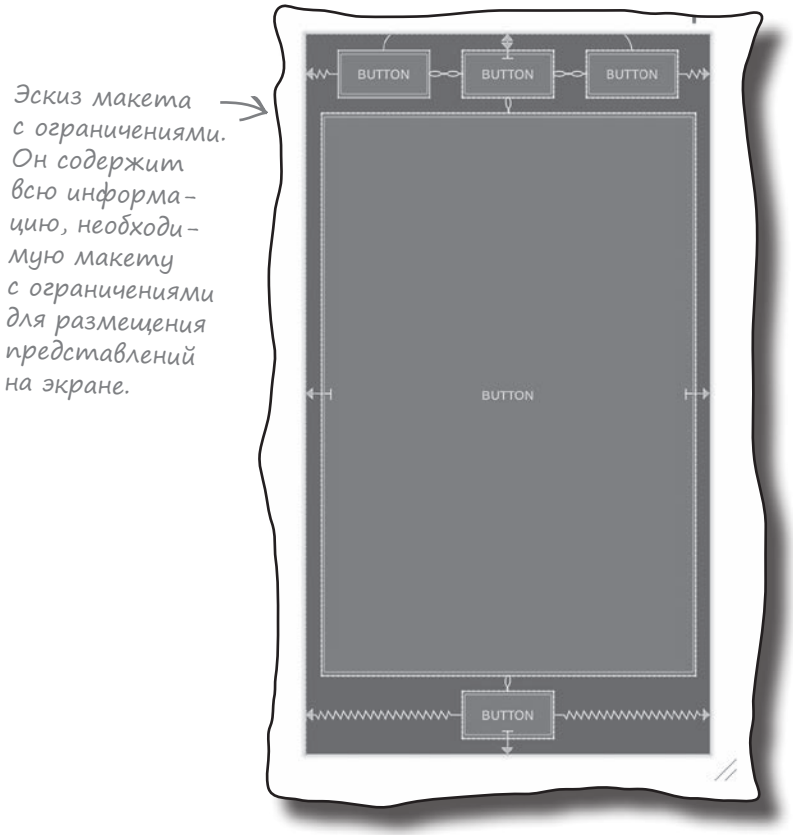
Макеты с ограничениями: первое знакомство

Макеты с ограничениями используются для построения гибких графических интерфейсов без вложения макетов.

Макеты с ограничениями сложнее линейных или фреймовых макетов, но они обладают значительно большей гибкостью. Также они намного эффективнее работают при построении сложных интерфейсов, так как формируют иерархию представлений с меньшей глубиной, а это означает, что Android придется выполнять меньше вычислений во время выполнения.

Макеты с ограничениями строятся ВИЗУАЛЬНО

Другое преимущество макетов с ограничениями заключается в том, что они спроектированы специально для работы с визуальным редактором Android Studio. В отличие от линейных и фреймовых макетов, которые обычно пишутся напрямую на уровне XML, макеты с ограничениями строятся *визуально*. Вы перетаскиваете представления на эскиз макета в визуальном редакторе и вводите инструкции, которые определяют, как должно выводиться каждое представление:



В отличие от линейных и фреймовых макетов, макеты с ограничениями входят в семейство библиотек **Android Jetpack**. Возможно, это название вам уже встречалось, но что же такое Jetpack?

Макеты с ограничениями являются частью Android Jetpack

Android Jetpack — семейство библиотек, которые помогают применять передовые практики разработки, снижают объем рутинного кода и вообще упрощают программирование. Они включают макеты с ограничениями, навигацию, библиотеку хранения данных Room (которая помогает создавать базы данных) и многое, многое другое.

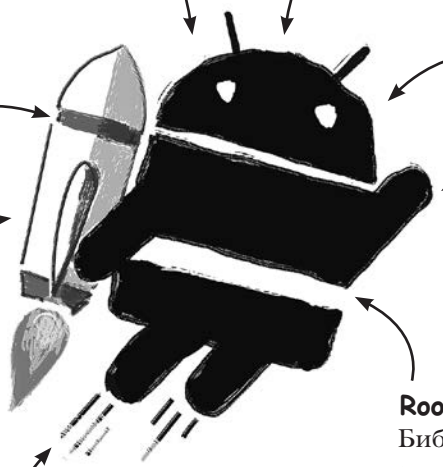
Ниже представлены наши любимые компоненты Jetpack; в последующих главах вы научитесь пользоваться ими.

Фрагменты
Фрагмент — своего рода субактивность, которая используется для управления частью экрана.

Навигация
Включает средства для переходов от экрана к экрану с безопасной передачей аргументов между ними.

LiveData
Позволяет строить приложения, реагирующие на изменения данных сразу же при их обнаружении.

Связывание данных
Позволяет строить макеты, которые могут обращаться к методам и свойствам Kotlin.



Compose
Инструментарий для построения платформенных приложений Android без использования файлов макетов.

Модель представления
Позволяет выделить бизнес-логику экрана в отдельный класс.

Макеты с ограничениями
Используются для построения сложных, гибких макетов без лишних затрат, связанных с вложенностью.

Представление Recycler
Эффективный список, который может использоваться для вывода данных и переходов между экранами.

Room
Библиотека хранения данных, которая позволяет создавать базы данных и взаимодействовать с ними.

Другая замечательная особенность Jetpack заключается в том, что вы можете писать код, последовательно работающий в новых и старых версиях Android. И это очень удобно для ваших пользователей, так как вы можете включать в приложения новую интересную функциональность Android, которая будет работать на старых устройствах.

Примером служит класс `AppCompatActivity`, который уже использовался для написания кода активности. Ранее мы об этом не упоминали, но `AppCompatActivity` является частью Android Jetpack. Класс дополняет новой функциональностью активности в новых и старых версиях Android, и вам не придется беспокоиться о проблемах обратной совместимости.

Да! Вы уже использовали часть Android Jetpack, хотя и не подозревали об этом. Вы узнаете больше об использовании Jetpack в оставшейся части книги.

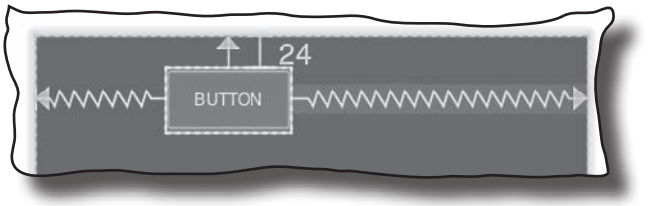
В этой главе вы научитесь пользоваться макетами с ограничениями. Давайте посмотрим, что же нам предстоит сделать.

Что нам предстоит сделать

Наше изучение макетов с ограничениями будет разделено на два больших этапа:

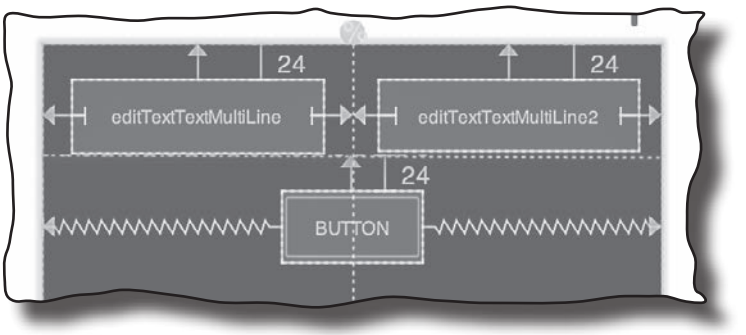
1. **Как позиционировать и задавать размер одного представления.** Вы научитесь пользоваться ограничениями и смещением для управления тем, где отдельное представление должно отображаться в его макете.

На первом этапе вы научитесь пользоваться ограничениями.



2. **Как позиционировать и задавать размеры нескольких представлений.** Затем вы примените новые знания к нескольким представлениям и освоите более мощные инструменты: направляющие, барьеры, цепочки и потоки.

На втором этапе вы научитесь строить более сложные макеты с ограничениями.



Создание нового проекта

Для приложения, которое мы построим, понадобится новый проект. Создайте его по той же схеме, которая использовалась в предыдущих главах. Выберите вариант Empty Activity, введите имя «My Constraint Layout» с именем пакета «com.hfad.myconstraintlayout» и подтвердите место сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android.

После того как проект будет создан, нужно принять меры к тому, чтобы в нем использовались макеты с ограничениями.

Отдельное представление
Несколько представлений

Использование Gradle для включения библиотек Jetpack

Под капотом каждый проект Android Studio использует Gradle для сборки приложения.

Чтобы все библиотеки Jetpack, в том числе макеты с ограничениями, стабильно работали во всех версиях Android, они не включаются в основной пакет Android SDK. Вместо этого любые необходимые библиотеки добавляются при помощи **Gradle**. Это система сборки, которая используется для компиляции кода, настройки приложений и загрузки всех дополнительных библиотек, необходимых вашему проекту.

Каждый раз, когда вы создаете новый проект, Android Studio создает два файла Gradle с именем ***build.gradle***.

Первая версия *build.gradle* находится в папке *project* и содержит основные настройки приложения, например используемую версию плагина Gradle.

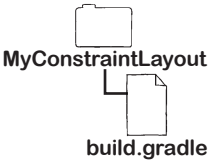
Вторая версия *build.gradle* находится в папке *app*. В ней задается большинство свойств приложения, например уровень API.

Файл build.gradle проекта должен содержать ссылку на репозиторий Google

Каждый проект должен знать, где найти необходимые ему дополнительные библиотеки Jetpack. Для этого в файл *build.gradle* проекта включается ссылка на репозиторий Google. Обычно Android Studio делает это за вас, но вы можете убедиться в том, что эта ссылка присутствует в файле: откройте файл *MyConstraintLayout/build.gradle* и найдите следующую строку (выделена ниже жирным шрифтом) в группе `repositories` раздела `allprojects`:

```
allprojects {
    repositories {
        google()
        ...
    }
}
```

Все библиотеки Jetpack берутся из этого репозитория. Среда Android Studio должна добавить эту строку за вас.

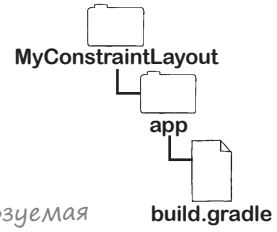


Файл build.gradle из папки app включает библиотеку макетов с ограничениями

Чтобы использовать макеты с ограничениями, необходимо включить ссылку на соответствующую библиотеку в файл *build.gradle* из папки **app**. Среда Android Studio должна добавить ее за вас, но вы можете лишний раз проверить: откройте файл *MyConstraintLayout/app/build.gradle* и найдите следующую строку (выделенную жирным шрифтом) в разделе `dependencies`:

```
dependencies {
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    ...
}
```

Используемая версия библиотеки макетов с ограничениями.

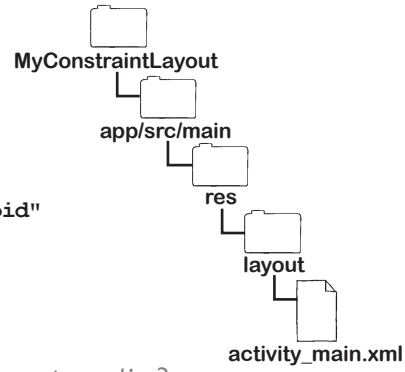


Если в файле нет этой строки, добавьте ее и щелкните на кнопке Sync Now, появляющейся в редакторе кода. Эта кнопка синхронизирует внесенные изменения с остальным кодом проекта и добавляет библиотеку.

Добавим макет с ограничениями в файл activity_main.xml

Итак, наш проект настроен для использования макетов с ограничениями. Можно переходить к работе с таким макетом.

Чтобы добавить макет с ограничениями в файл макета, воспользуйтесь элементом **<androidx.constraintlayout.widget.ConstraintLayout>**. Он будет использоваться в файле макета *activity_main.xml*, поэтому откройте этот файл в папке *app/src/main/res/layout* и убедитесь в том, что его код выглядит так:



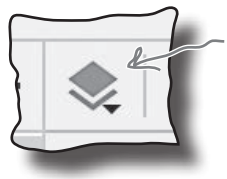

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

</androidx.constraintlayout.widget.ConstraintLayout>
```

Так определяется макет с ограничениями.

Если среда Android Studio добавила в макет дополнительные представления, удалите их.

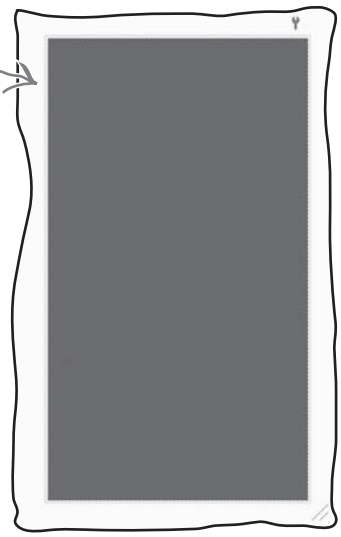
Вывод макета в эскизе

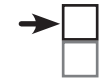
Представления будут добавляться в макет в эскизном режиме визуального редактора. Переключитесь в визуальный редактор при помощи кнопки Design, щелкните на кнопке Select Design Surface на панели инструментов редактора и выберите вариант Blueprint. На экране появляется эскиз макета:



Кнопка Select Design Surface. Используйте ее для выбора режима отображения макета: эскиза, визуального режима или их комбинации.

Эскиз макета. Пока он пуст, потому что в макет с ограничениями еще не добавлено ни одного представления.

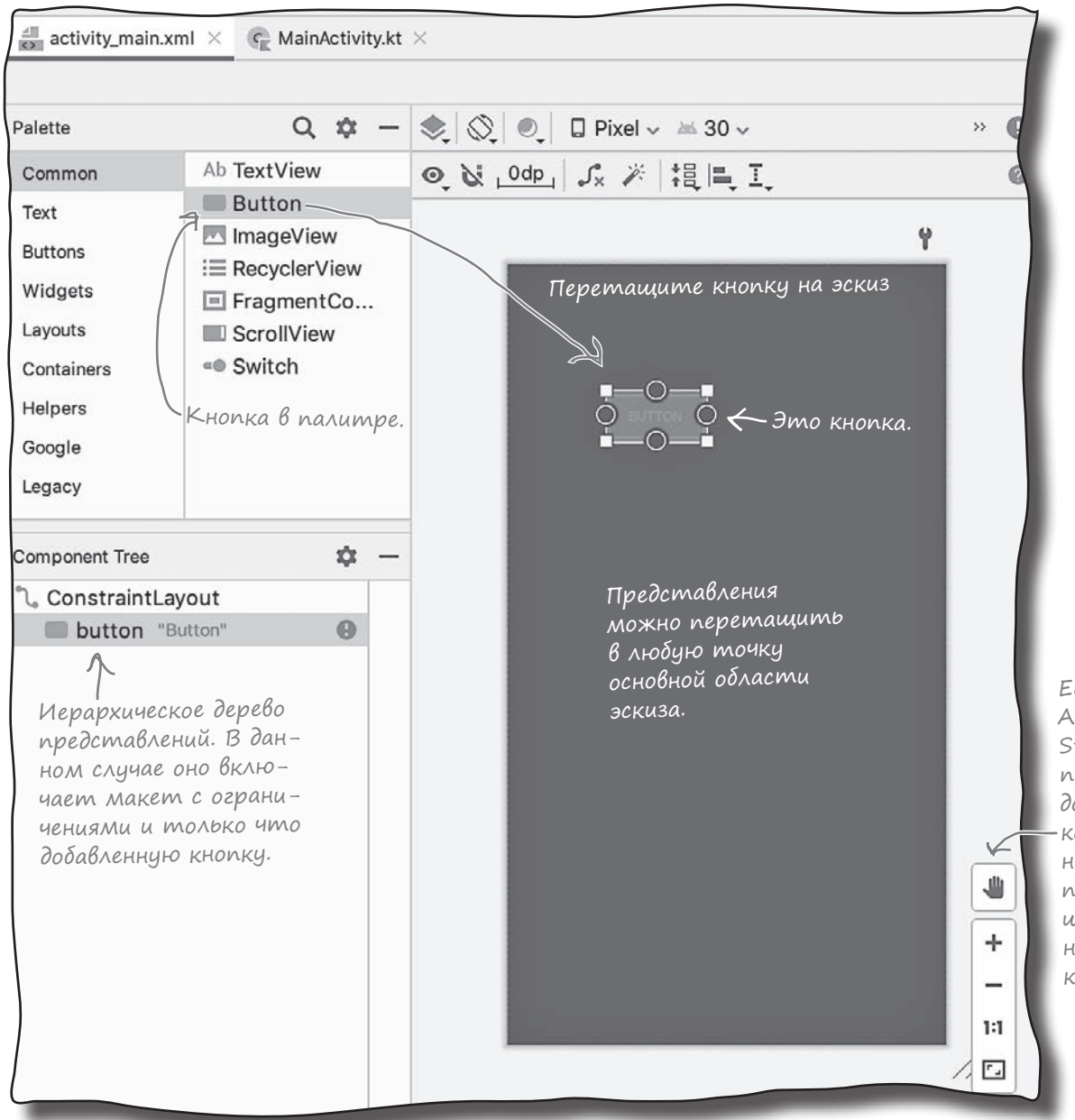




Отдельное представление
Несколько представлений

Добавление кнопки в макет

Сейчас мы добавим кнопку в макет. Для этого откройте палитру визуального редактора, найдите компонент Button (обычно он находится в разделе Common) и перетащите его на эскиз. Кнопку можно расположить в любом месте; главное, чтобы она располагалась в основной области:



Отдельное представление
Несколько представлений

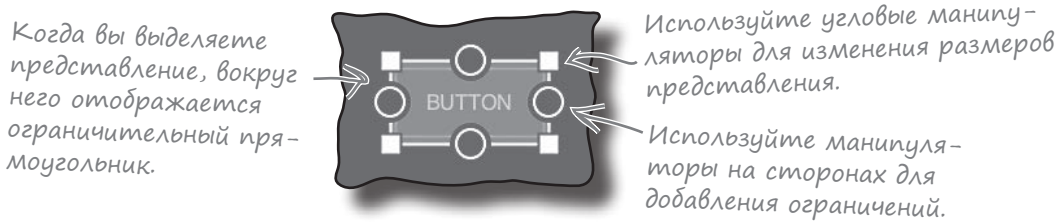
Размещение представлений с ограничениями

Когда вы перетаскиваете мышью представление в некоторую точку эскиза при использовании макета с ограничениями, вы не указываете, где это представление должно размещаться. Вместо этого размещение задается определением **ограничений**. **Ограничение** представляет собой связь или отношение, которое сообщает макету, где должно позиционироваться представление. Например, ограничение можно использовать для связывания представления с начальной стороной макета или под нижней стороной другого представления.

Добавим к кнопке горизонтальное ограничение.

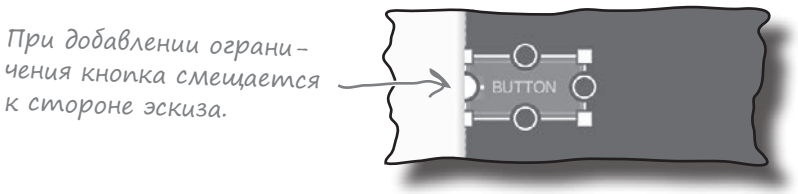
Чтобы вы поняли, как это работает, добавим ограничение, связывающее кнопку с левой стороной макета.

Сначала убедитесь в том, что кнопка выделена (для этого щелкните на ней). При выделении представления вокруг него отображается ограничивающий прямоугольник, на углах и сторонах которого располагаются манипуляторы. Угловые манипуляторы изменяют размеры представления, а манипуляторы на сторонах позволяют создать ограничения:



Чтобы добавить ограничение, щелкните на одном из манипуляторов ограничений и перетащите его к тому объекту, с которым должно быть связано представление. В данном случае левая сторона кнопки должна быть связана с левой стороной макета, поэтому щелкните на левом манипуляторе ограничения и перетащите его к левой стороне эскиза:

Тем самым вы добавляете ограничение, а кнопка смещается влево:



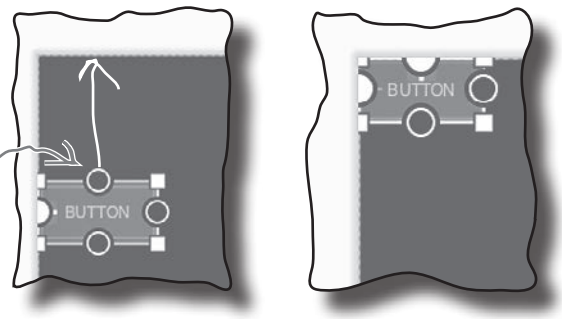
Так добавляется горизонтальное ограничение. Посмотрим, что происходит при добавлении вертикальных ограничений.

Отдельное представление
Несколько представлений

Добавление вертикального ограничения

Мы воспользуемся вертикальным ограничением, чтобы связать кнопку с верхней стороной макета. Щелкните на манипуляторе ограничения на верхней стороне кнопки и перетащите его к верхней стороне эскиза. Тем самым вы добавите вертикальное ограничение, из-за которого кнопка смещается наверх.

Щелкните на манипуляторе на верхней стороне кнопки и перетащите ее на верхнюю сторону эскиза. Кнопка смещается к верхней стороне эскиза.



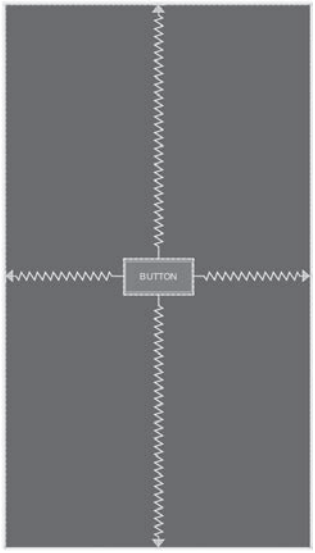
Упражнение

Вы уже видели, что происходит при добавлении ограничений к левой и верхней стороне. Как вы думаете, что произойдет при добавлении обратных ограничений к правой и нижней стороне кнопки? Попробуйте сделать это и нарисуйте свой ответ внизу.

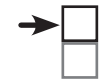
Упражнение Решение

Вы уже видели, что происходит при добавлении ограничений к левой и верхней стороне. Как вы думаете, что произойдет при добавлении обратных ограничений к правой и нижней стороне кнопки? Попробуйте сделать это и нарисуйте свой ответ внизу.

Когда вы добавляете ограничение, соединяющее правую сторону кнопки с левой, левое и правое ограничения растягивают кнопку в противоположных направлениях. Кнопка выравнивается горизонтально по центру.



Когда вы добавляете ограничение, связывающее нижние стороны кнопки и эскиза, верхнее и нижнее ограничения растягивают кнопку и она выравнивается по центру макета.

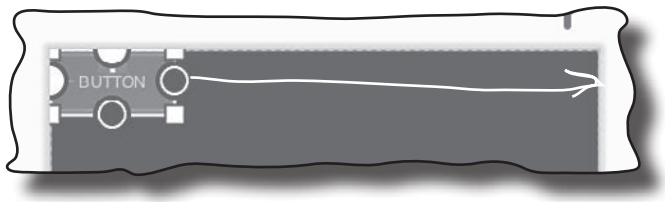


Отдельное представление
Несколько представлений

Использование противоположных ограничений для выравнивания по центру

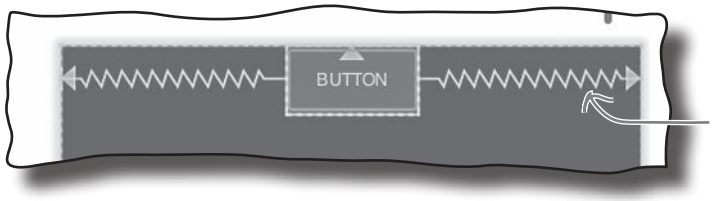
Как вы узнали, ограничения могут использоваться для связывания представлений со сторонами эскиза. Каждое ограничение работает как пружина, связывающая представление со стороной эскиза.

Если вы хотите позиционировать представления в центре эскиза, для этого можно добавить ограничения к противоположным сторонам представления. Например, чтобы выровнять кнопку горизонтально по центру, добавьте одно ограничение, которое смещает кнопку влево, и другое ограничение, которое смещает ее вправо:



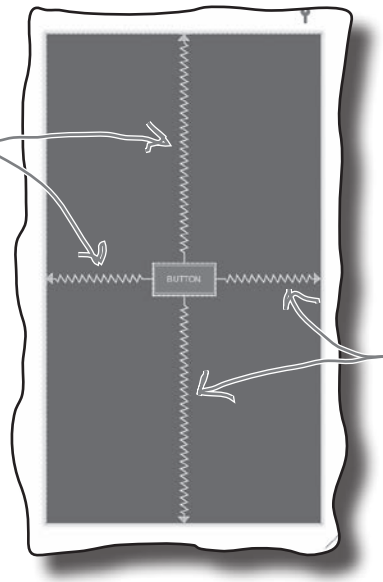
Чтобы добавить ограничение, смещающее кнопку вправо, щелкните на манипуляторе ограничения на правой стороне кнопки и перетащите его к правой стороне эскиза.

Два ограничения тянут кнопку в разных направлениях, и в результате она выравнивается по центру:



Противоположные ограничения смещают кнопку в центр.

Представление также можно выровнять по центру по вертикали; для этого следует добавить ограничения к верхней и нижней стороне. А чтобы кнопка была выровнена по центру и по горизонтали, и по вертикали, добавьте ограничения ко всем сторонам:



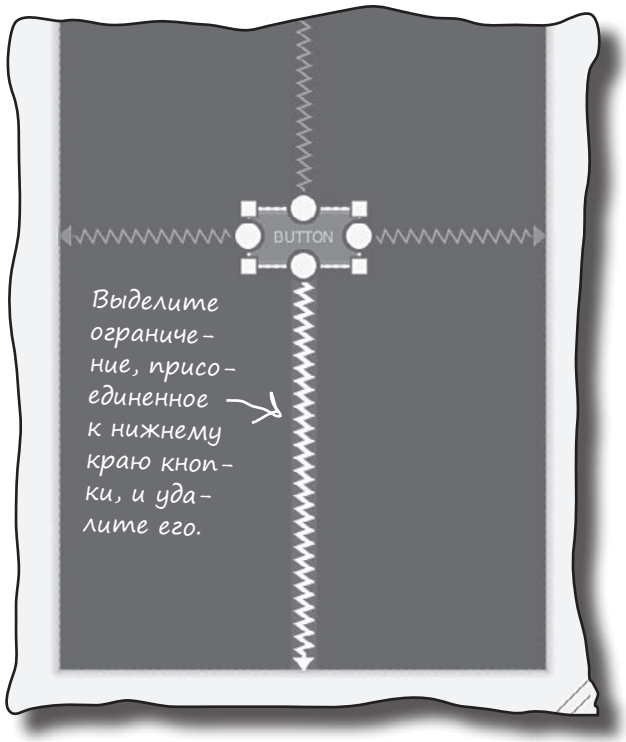
Эти ограничения подтягивают кнопку к верхней и нижней стороне эскиза.

Если на экране видны не все ограничения, щелкните на кнопке View и выберите команду Select All Constraints.

Эти ограничения подтягивают кнопку к нижней и правой стороне эскиза.

Ограничения, которые стали ненужными, можно удалить

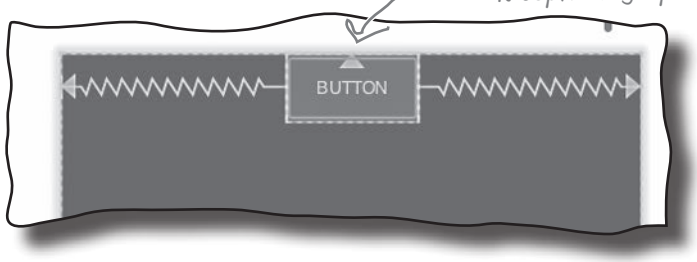
Вы можете убрать любые ограничения, которые вам более не нужны. Для этого выделите их в эскизе и удалите. Например, если где-то в центре эскиза расположена кнопка, можно удалить ограничение, присоединенное к ее нижнему краю:



Следите за тем, чтобы у каждого представления было хотя бы одно вертикальное и горизонтальное ограничение.

Если у представления нет вертикальных ограничений, то при запуске приложения оно будет отображаться у верхнего края макета. Если у представления нет горизонтальных ограничений, оно будет отображаться у стартового края.

Удаление этого ограничения означает, что кнопка уже не будет смещаться к нижнему краю эскиза. Верхнее ограничение смещает кнопку к верхней стороне, чтобы она выравнивалась по центру по горизонтали, но не по вертикали:



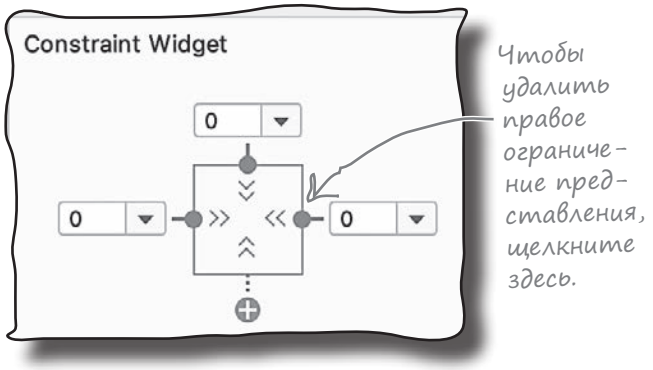
Для удаления ненужных ограничений также можно воспользоваться виджетом ограничений. Давайте посмотрим, как это делается.

Удаление ограничений с использованием виджета

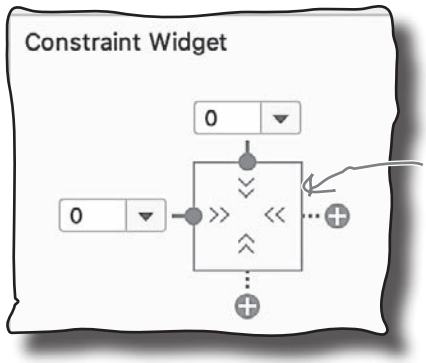
Виджет ограничений отображается на панели Attributes в боковой части конструктора. Он появляется, когда вы выделяете представление, и отображает диаграмму с ограничениями представления и размерами полей.

Через несколько страниц вы больше узнаете о панели Attributes.

Чтобы удалить ограничение в виджете, выделите в эскизе представление, из которого вы хотите его удалить, затем щелкните на манипуляторе ограничения в виджете. Ограничение удаляется, а позиция представления в эскизе изменяется.



Чтобы удалить правое ограничение представления, щелкните здесь.

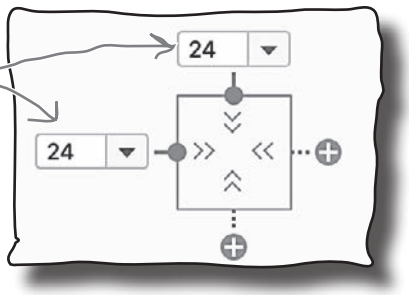


Ограничение удаляется.

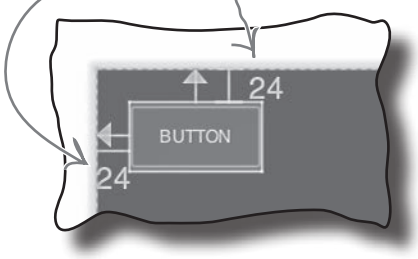
Использование виджета для добавления полей

Возможно, вы также заметили, что рядом с каждым ограничением в виджете выводится число. Оно определяет размер полей у края представления, чтобы между краем представления и краем макета оставалось свободное пространство. Например, чтобы назначить левому и верхнему полю значение 24dp, введите соответствующие значения на диаграмме:

Полям у этих сторон представления назначен размер 24dp.



Поля отображаются в эскизе.



При помощи кнопки Default Margins на панели инструментов визуального конструктора можно назначить размер по умолчанию для всех новых полей. Например, если ввести здесь значение 24dp, то всем новым ограничениям будут автоматически назначаться поля 24dp.

Параметр Default Margins. Здесь всем новым полям назначается размер по умолчанию 24dp.

Отдельное представление
Несколько представлений

Изменения в эскизе отражаются в XML

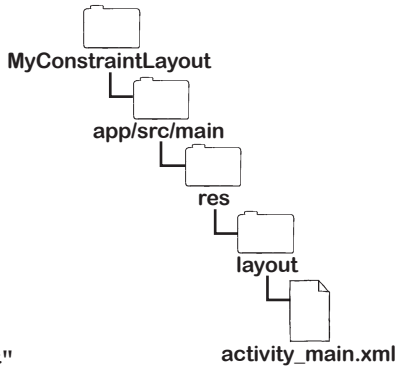
Когда вы добавляете представления в эскиз, задаете ограничения и поля, они добавляются в базовую разметку XML макета. Чтобы убедиться в этом, переключитесь в режим вывода кода макета. Код должен выглядеть примерно так (если он несколько отличается от приведенного, не беспокойтесь):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ...>
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:text="Button"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Добавленная кнопка.

Все эти атрибуты вам уже знакомы.

Эти строки применимы только к макетам с ограничениями.



Как видите, разметка XML теперь содержит кнопку. Код кажется вам знакомым? Если так, мы похвалим вас за внимательность — он включает атрибуты, о которых вы узнали в главе 3.

Атрибуты width, height и margins определяются точно так же, как и прежде. При желании вы можете изменять их значения в XML вместо визуального редактора.

Весь незнакомый код ограничивается двумя строками, которые определяют ограничения для начальной и верхней стороны:

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent" />
```

Эти строки описывают ограничения для левой (начальной) и верхней стороны.

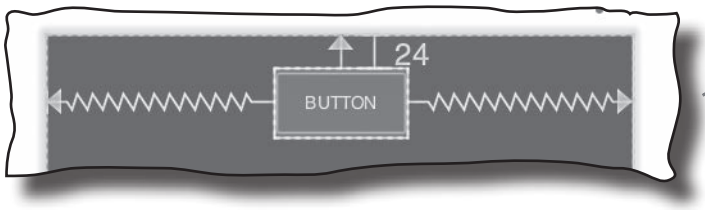
Похожий код генерируется при добавлении ограничений к остальным сторонам кнопки.

Теперь, когда вы примерно представляете, как выглядит разметка XML макетов с ограничениями, вернитесь в визуальный редактор. Мы рассмотрим некоторые приемы, используемые для позиционирования представлений.

Для представлений может определяться смещение

Как было показано ранее, ограничения можно применять к противоположным сторонам представления. При этом представление по умолчанию выравнивается по центру, но вы также можете управлять его положением относительно каждой стороны, изменяя величину **смещения**. Тем самым вы сообщаете Android пропорциональные длины каждого ограничения на его обеих сторонах.

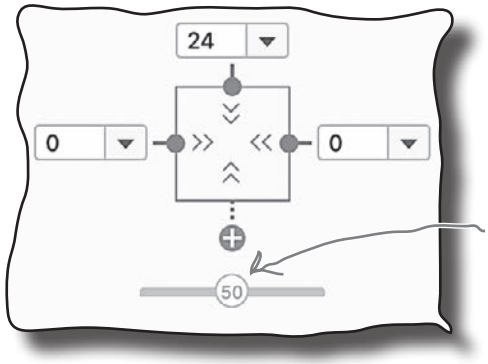
Чтобы увидеть, как работает смещение, изменим величину горизонтального смещения кнопки, чтобы она не располагалась в центре. Сначала убедитесь в том, что кнопке назначены ограничения с левой и правой стороны:



Убедитесь в том, что кнопка имеет ограничения слева и справа, чтобы она была выровнена горизонтально по центру.

Затем выделите кнопку, чтобы появился виджет ограничения.

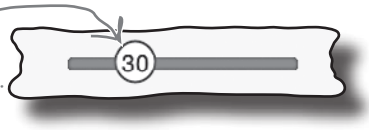
Под диаграммой представления находится ползунок с числом. Это число определяет горизонтальное смещение представления в процентах.



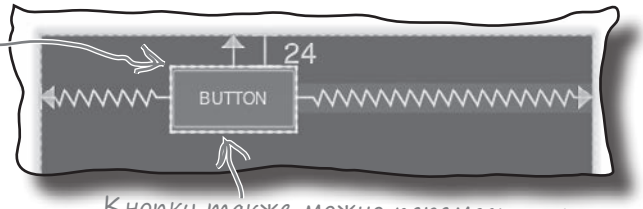
Ползунок предназначен для определения горизонтального смещения представления. В настоящее время выводится значение 50, чтобы представление размещалось на середине между его горизонтальными ограничениями.

Чтобы изменить смещение, достаточно передвинуть ползунок. Если, допустим, вы сместите ползунок влево, чтобы появилось число 30, кнопка в эскизе также сместится влево:

Перемещение ползунка...



...приводит к перемещению кнопки.



Кнопку также можно переместить, щелкая на ней и перетаскивая ее мышью, но этот способ менее точен.

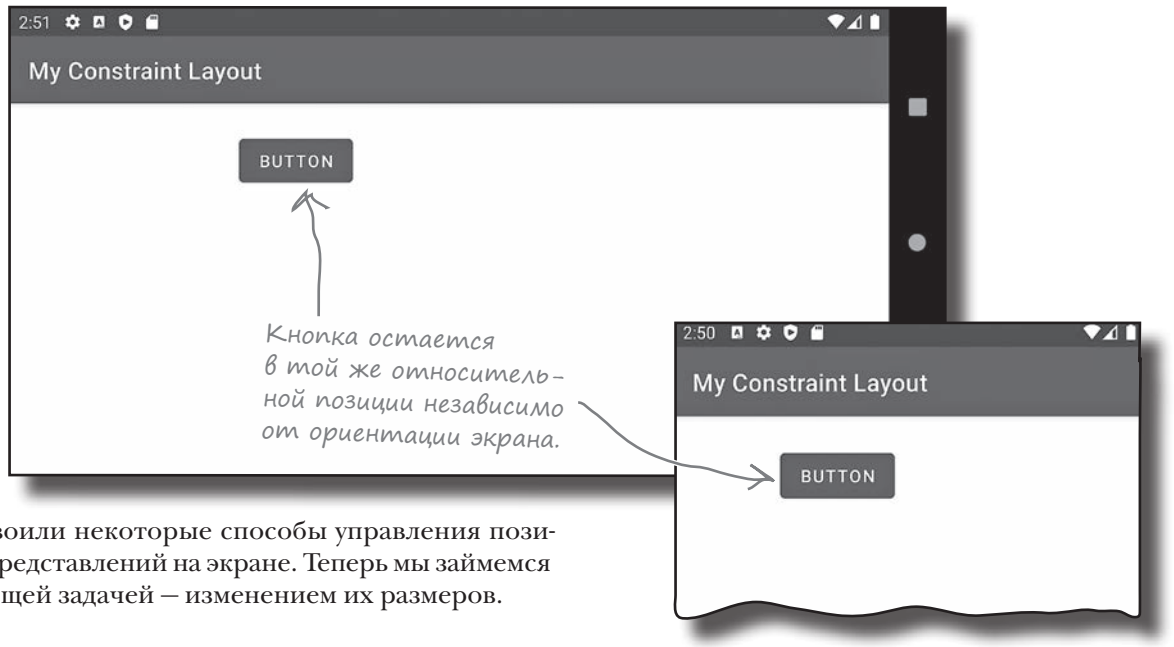
Представление сохраняет эту относительную позицию независимо от размера и ориентации экрана. Давайте проверим на практике, как работает этот макет.

Тест-драйв

При запуске приложения кнопка выводится смещенной от центра к верхнему краю экрана. При повороте устройства относительная позиция сохраняется.

Отдельное представление
Несколько представлений

Если ваш экран не поворачивается при повороте устройства, проверьте, включен ли у вас режим автоповорота. Этот режим включается в разделе Settings → Display.



Вы освоили некоторые способы управления позицией представлений на экране. Теперь мы займемся следующей задачей — изменением их размеров.

Часто Задаваемые Вопросы

В: Вы сказали что все компоненты Jetpack (включая макеты с ограничениями) содержатся в отдельных библиотеках, которые не входят в основную версию Android. Почему?

О: Чтобы вы могли использовать новые версии и возможности этих компонентов в старых версиях Android.

В: При добавлении библиотеки макетов с ограничениями *build.gradle* вы сказали, что используете конкретную версию. А можно использовать другую?

О: Все возможности и весь код, приведенный в этой главе, работает с этой версией библиотеки макетов с ограничениями. Постоянно выходят обновленные версии библиотек, но помните, что переход на другую версию может нарушить работоспособность кода.

В: А как мне узнать, существует ли новая версия?

О: Файлы *build.gradle* обычно обнаруживают появление новой версии и сообщают вам об этом.

В: Может ли визуальный редактор добавить ограничения автоматически?

О: На панели инструментов визуального редактора находится кнопка «Enable Autoconnection to Parent», которая позволяет автоматически создавать ограничения между каждым новым представлением и стороной эскиза. Также существует кнопка Infer Constraints, которая пытается определить необходимые ограничения на основании положения представлений в макете. Оба варианта не всегда дают идеальный результат, но вам стоит хотя бы опробовать их.

Размеры представлений можно изменять

Отдельное представление
Несколько представлений

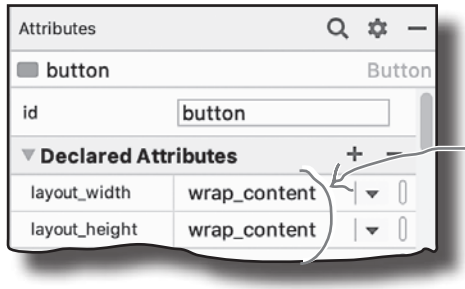
Как нетрудно предположить, для изменения размеров представлений в макетах с ограничениями можно воспользоваться атрибутами `layout_width` и `layout_height`. Это можно сделать в разметке XML макета или на панели Attributes визуального редактора.

Панель Attributes выводится со стороны эскиза. При выборе представления на ней выводятся все атрибуты, которые были объявлены ранее (например, `layout_width` и `layout_height`), а вы можете создать новые атрибуты, которые еще не объявлялись.

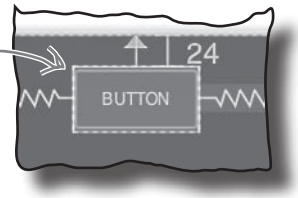
Подгонка размера по содержимому

Как и с линейными и фреймовыми макетами, представлению можно назначить минимальные размеры, достаточные для вывода его содержимого, — для этого его атрибутам `layout_width` и `layout_height` присваивается значение `wrap_content`. Например, если представление является кнопкой, то кнопке можно назначить минимальный размер, необходимый для вывода текста:

Панель Attributes.



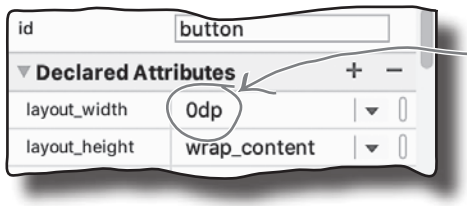
Если присвоить ширине и высоте кнопки значение «wrap_content», кнопка будет иметь минимальный размер, необходимый для вывода ее содержимого (как и с другими макетами).



Подгонка размеров по ограничениям

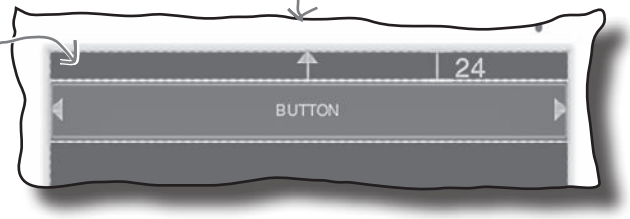
Если вы добавили ограничения к противоположным сторонам представления, можно подогнать размер представления по ограничениям. Для этого его атрибутам `layout_width` и/или `layout_height` присваивается значение 0dp: если присвоить `layout_width` значение 0dp, размер представления выбирается по горизонтальным ограничениям, а если присвоить `layout_height` значение 0dp, размер представления будет выбираться по вертикальным ограничениям.

В следующем примере атрибуту `layout_width` кнопки присваивается значение 0dp, чтобы размер кнопки соответствовал ее горизонтальным ограничениям:



Если ширина задана равной 0dp, размер кнопки выбирается в соответствии с ее горизонтальными ограничениями.

По умолчанию при подгонке размеров по ограничениям представление расширяется настолько, насколько это возможно. Существуют и другие варианты, о которых можно узнать по адресу https://developer.android.com/training/constraint-layout



Итак, теперь вы знаете, как изменить размеры представления. Поэкспериментируйте с разными приемами, а затем попробуйте выполнить упражнение на следующей странице.

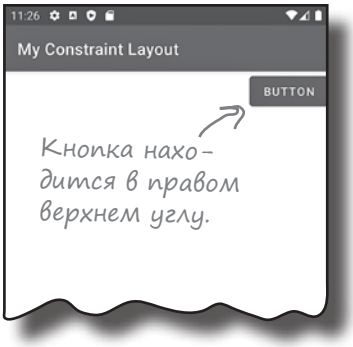
СТАНЬ ограничением

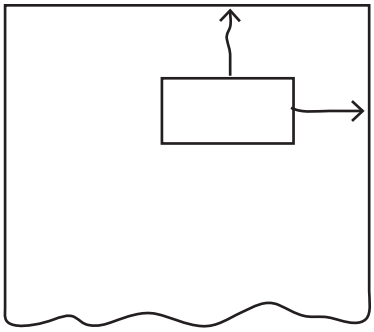
Представьте, что вы — макет с ограничениями. Нарисуйте ограничения, необходимые для построения всех приведенных ниже макетов. Также задайте значения layout_width, layout_height и смещение (там, где это необходимо) для каждого представления. Первый пример мы решили за вас.

К каждому эскизу необходимо добавить представления и ограничения.

A

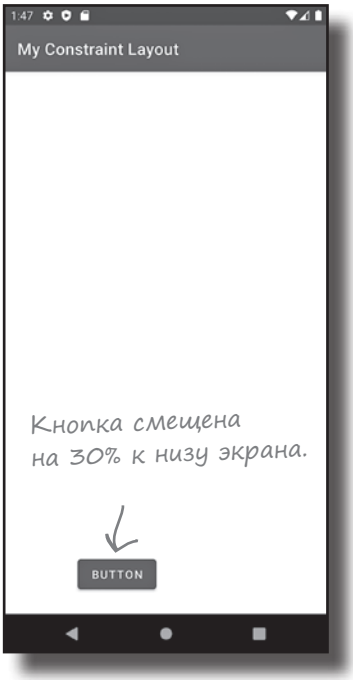
Так должен выглядеть экран.

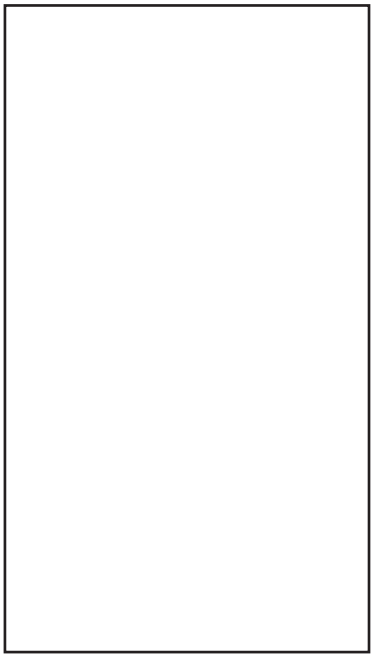




layout_width: wrap_content
layout_height: wrap_content

B

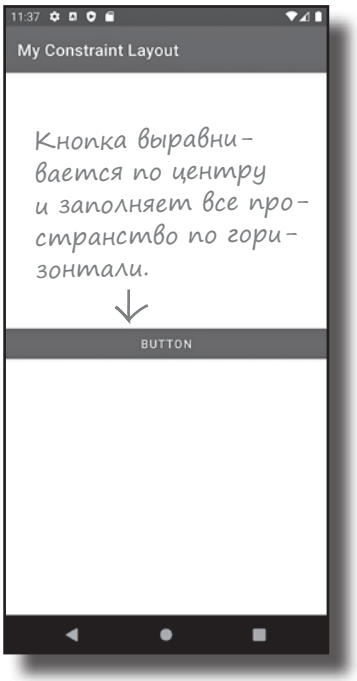




Кнопка заполняет все доступное пространство.

C

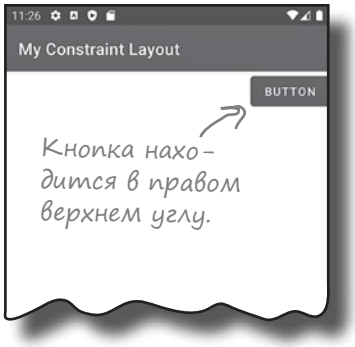
D

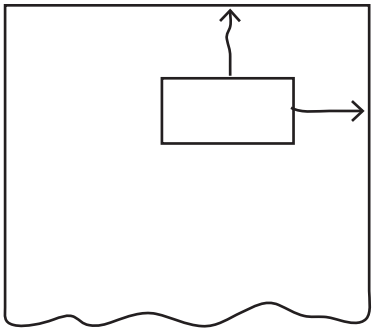


СТАНЬ ограничением. Решение

Представьте, что Вы — макет с ограничениями. Нарисуйте ограничения, необходимые для построения всех приведенных ниже макетов. Также задайте значения layout_width, layout_height и смещение (там, где это необходимо) для каждого представления. Первый пример мы решили за Вас.

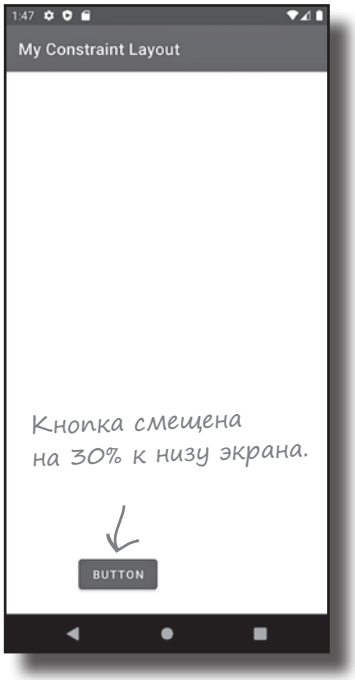
A

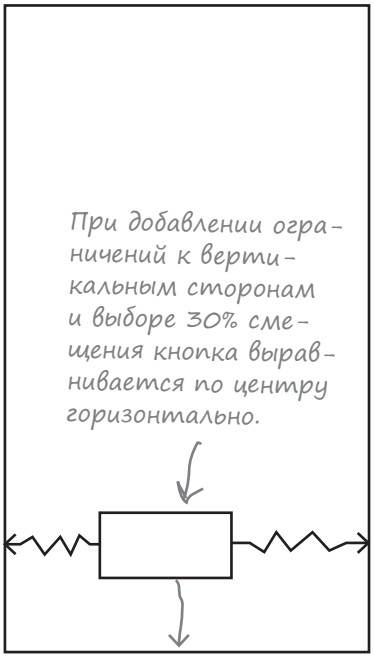




layout_width: wrap_content
layout_height: wrap_content

B

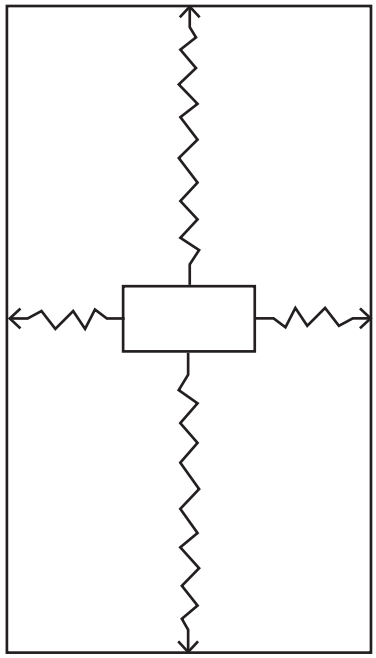




layout_width: wrap_content
layout_height: wrap_content
смещение: 30%

Кнопка заполняет все доступное пространство.

C

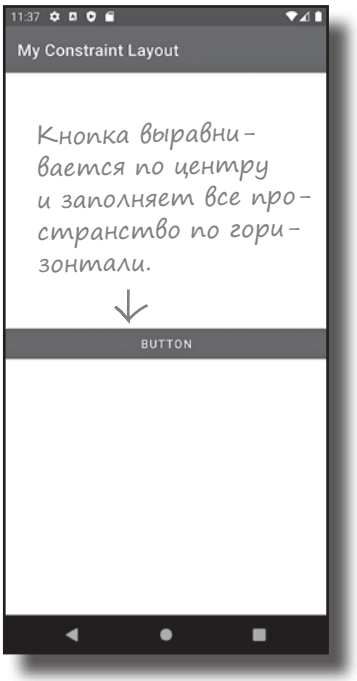


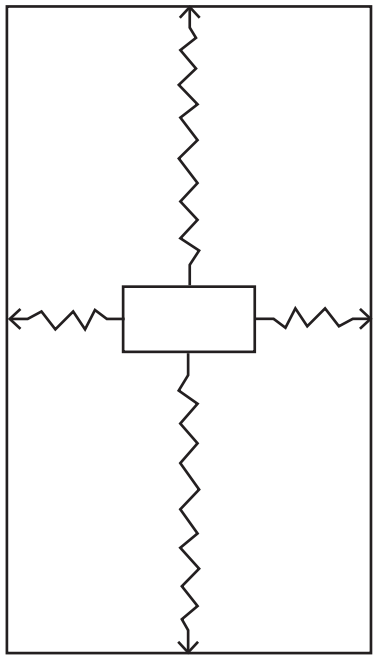
Кнопка должна растягиваться во всех направлениях, поэтому ограничения должны быть применены ко всем сторонам, а ширина и высота должны быть равны 0dp.

layout_width: 0dp

layout_height: 0dp

D





layout_width: 0dp

layout_height: wrap_content

Кнопка должна быть растянута по горизонтали и выровнена по центру по вертикали. Следовательно, для всех сторон должны быть определены ограничения, ширине необходимо присвоить 0dp, а высоте — значение wrap_content.

Отдельное представление
Несколько представлений

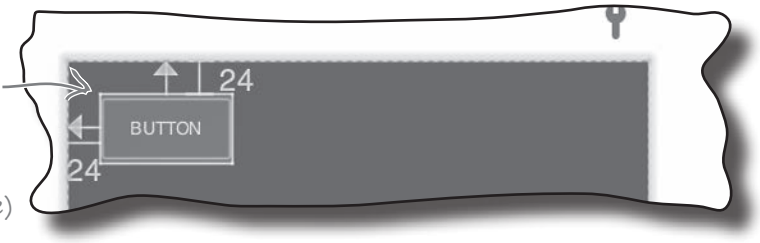
В макетах обычно используется несколько представлений

К настоящему моменту вы узнали, как позиционировать и задать размеры отдельного представления в макете с ограничениями. Впрочем, в большинстве случаев макет должен содержать несколько представлений, которые размещаются относительно друг друга.

Чтобы увидеть, как это делается, сначала убедитесь с том, что ваш макет с ограничениями содержит одну кнопку с двумя ограничениями: одно связывает верхнюю сторону кнопки с верхней стороной эскиза, а другое связывает левую сторону с левой стороной эскиза. Его свойствам `layout_width` и `layout_height` присваивается значение `wrap_content`, и этим сторонам назначаются интервалы 24dp.

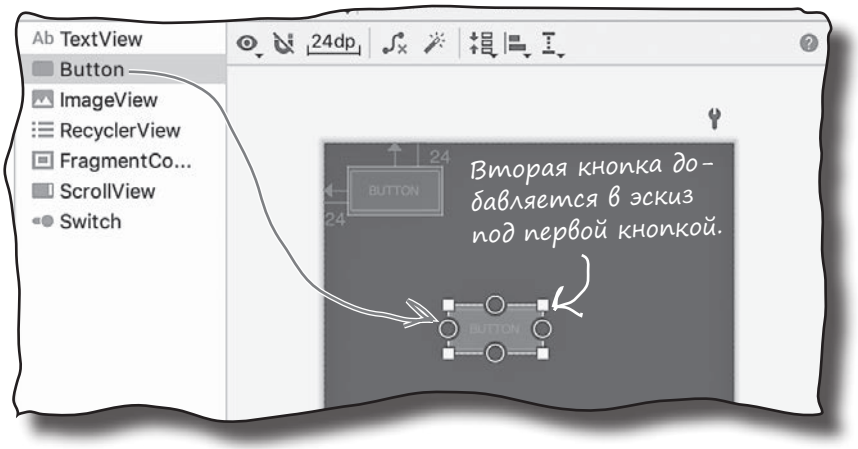
После внесения этих изменений кнопка размещается в левом верхнем углу эскиза:

Кнопка размещается здесь, потому что ей назначены два ограничения (одно горизонтальное и одно вертикальное) и интервалы 24dp.



Добавление второй кнопки в эскиз

Затем добавьте вторую кнопку в эскиз. Перетащите кнопку с палитры и разместите ее ниже первой кнопки:



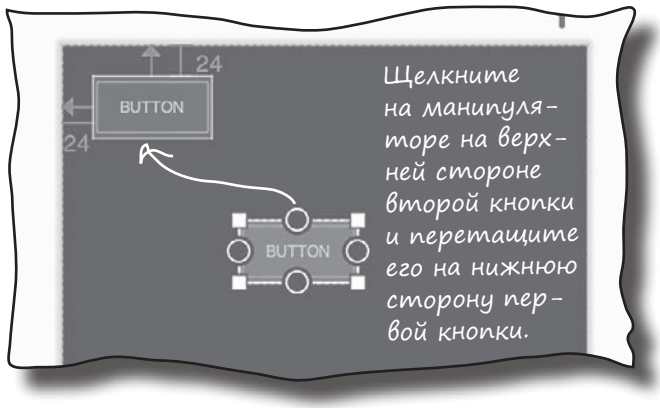
Теперь эскиз содержит две кнопки. Разберемся, как разместить эти кнопки относительно друг друга.

Представления можно связать с другими представлениями

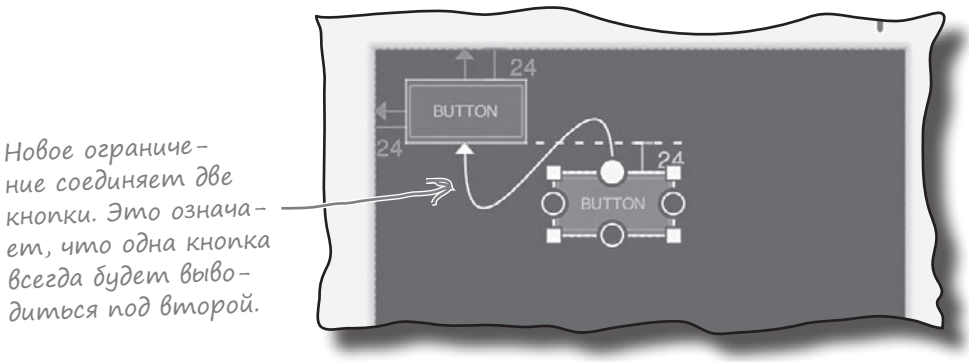
Отдельное представление
Несколько представлений

Как вы уже знаете, ограничения позволяют связать представление со стороной эскиза. Однако ограничения также можно использовать для связывания двух представлений между собой; тем самым вы указываете, как эти представления должны отображаться относительно друг друга.

Чтобы понять, как работает этот вариант связывания, выделите вторую кнопку в эскизе и создайте ограничение от верхней стороны второй кнопки к нижней стороне первой кнопки:



После добавления ограничения кнопка смещается вверх, соединяется с первой кнопкой и выводится под ней:



Ограничение означает, что вторая кнопка всегда будет расположена под первой независимо от позиции первой кнопки на экране устройства.

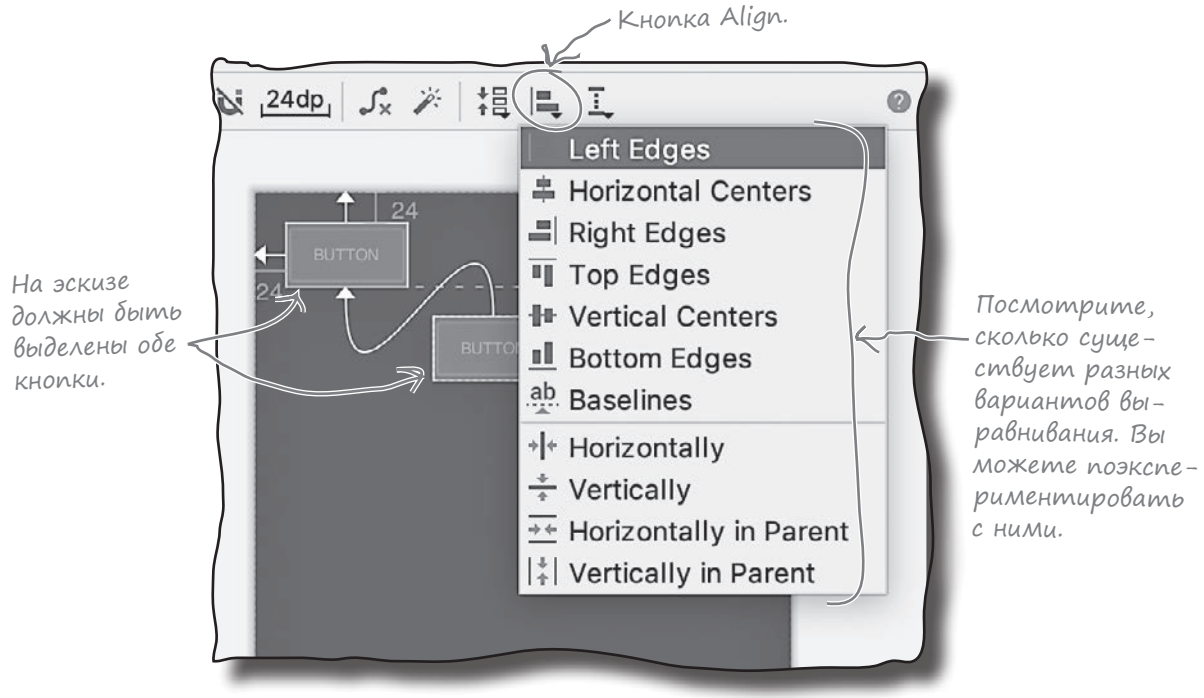
Когда вы разместите два представления подобным образом, на следующем шаге нужно позаботиться о том, чтобы они всегда были выровнены. Посмотрим, как это делается.

Представления также можно выравнивать

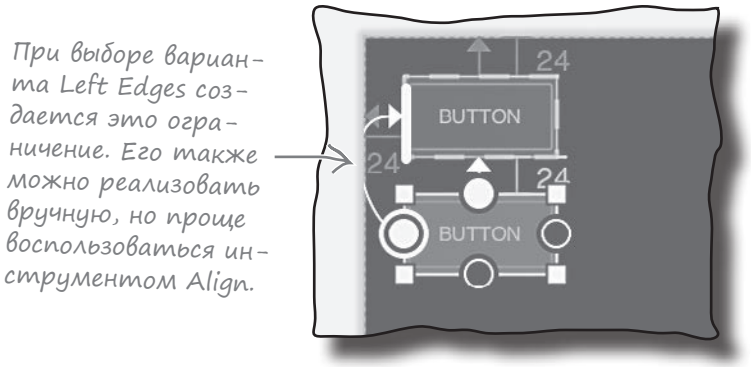
Отдельное представление
Несколько представлений

Самый простой способ выравнивания двух представлений основан на использовании кнопки Align на панели инструментов визуального редактора.

Чтобы понять, как он работает, выровняем две кнопки в эскизе по левому краю. Сначала выделите обе кнопки — щелкните на них, удерживая нажатой клавишу Shift. Затем щелкните на кнопке Align, чтобы открыть набор вариантов выравнивания:



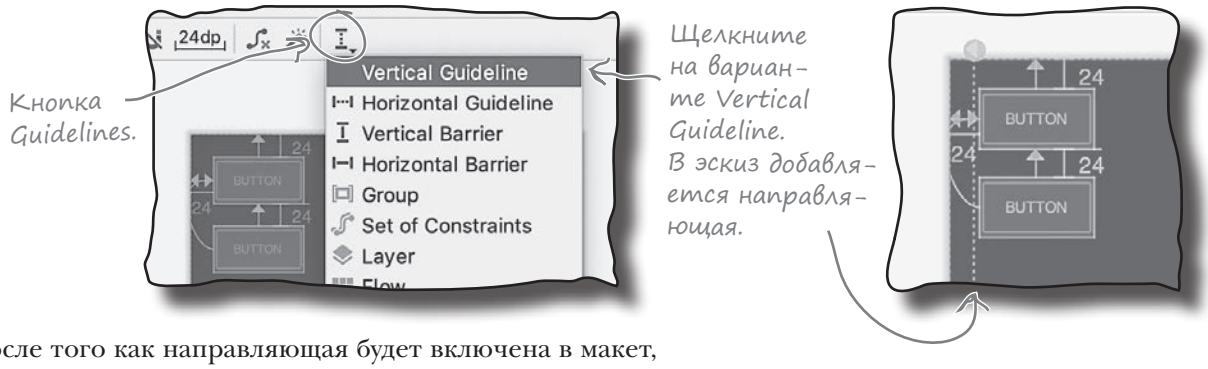
Щелкните на варианте Left Edges, чтобы выровнять две кнопки по левому краю. При этом в эскиз добавляется ограничение, которое соединяет левые стороны двух представлений:



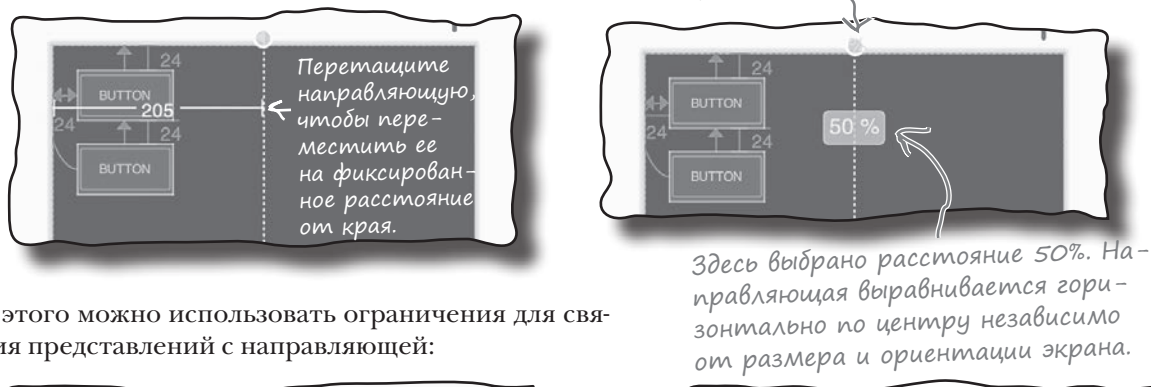
Выравнивание представлений по направляющим

Еще один способ выравнивания представлений основан на использовании **направляющих** — фиксированных линий, которые включаются в макет для ограничения представлений. Направляющие видны только в визуальном редакторе, а во время работы приложения они остаются невидимыми.

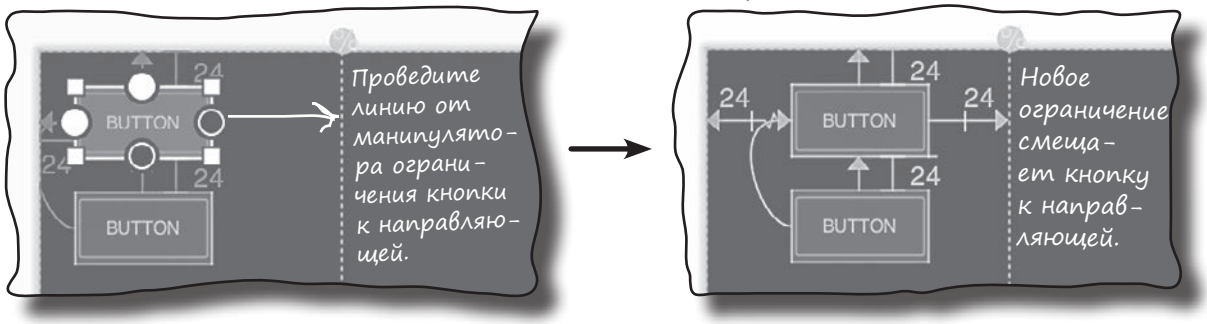
Чтобы понять, как работают направляющие, добавим направляющую в эскиз. Щелкните на кнопке Guidelines на панели инструментов визуального редактора и выберите вариант Vertical guideline. В эскиз включается вертикальная направляющая:



После того как направляющая будет включена в макет, ее можно переместить в любое место, просто перетащив мышью. Также можно переключить режим измерения и выбрать между фиксированным расстоянием от края эскиза и фиксированным процентом:



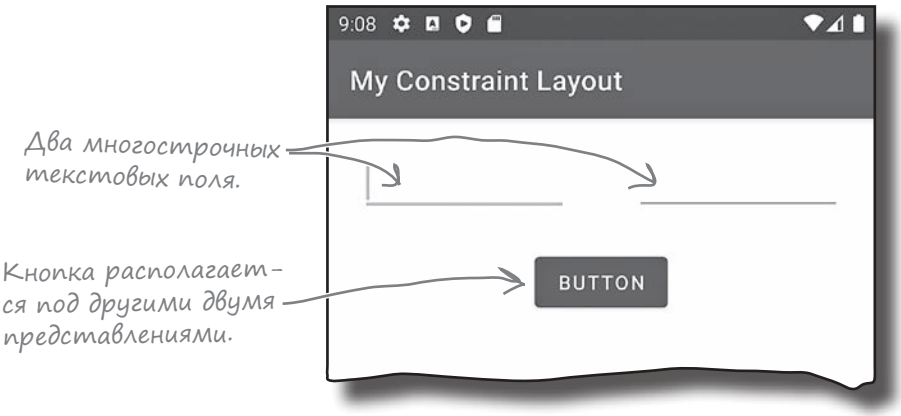
После этого можно использовать ограничения для связывания представлений с направляющей:



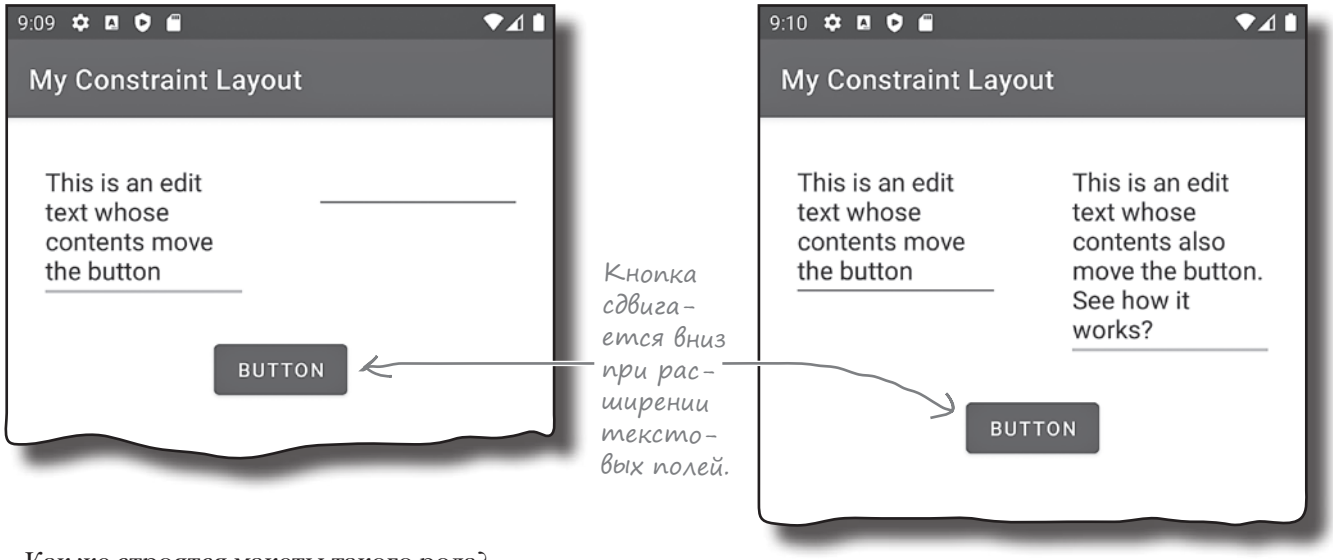
Направляющие имеют фиксированную позицию

Направляющие устанавливаются либо на фиксированном расстоянии от края эскиза, либо на фиксированном процентном соотношении между ними. Они остаются в этой позиции в ходе выполнения приложения, таким образом предоставляя полезные средства выравнивания представлений.

В некоторых ситуациях требуется что-то более гибкое. Предположим, у вас имеется макет с двумя многострочными текстовыми полями, расположенными рядом друг с другом, а под ними выводится кнопка:



Текстовые поля расширяются по вертикали в процессе ввода текста пользователем. Кнопка должна смещаться при изменении размеров представлений, чтобы она всегда располагалась под текстовыми полями:



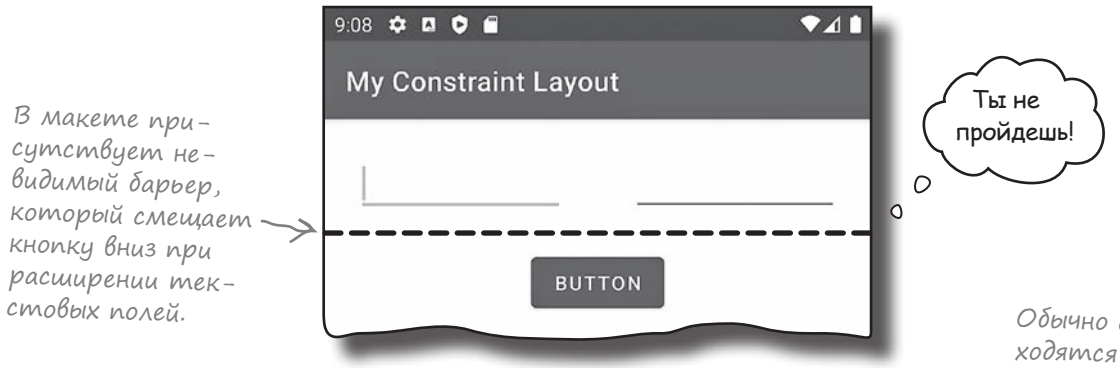
Как же строятся макеты такого рода?

Отдельное представление
Несколько представлений

Создание подвижного барьера

Для создания подобных макетов можно воспользоваться **барьером**. Барьер похож на направляющую, но он не имеет фиксированной позиции. Вместо этого он отслеживает размеры представлений и смещается при их изменении. При этом изменяется позиция любых представлений, связанных ограничениями с барьером.

В примере на предыдущей странице два текстовых поля размещаются над горизонтальным барьером, а кнопка связывается ограничением для размещения под ними. При расширении текстового поля барьер смещается вниз и изменяет позицию кнопки:



В макете присутствует невидимый барьер, который смещает кнопку вниз при расширении текстовых полей.

Обычно они находятся в разделе «Text» палитры и называются «Multiline Text».

Построение макета с барьером

Следующий пример показывает, как работают барьеры.

Сначала удалите все представления и убедитесь в том, что эскиз включает вертикальную направляющую, расположенную в позиции 50%. Затем перетащите два многострочных текстовых поля из палитры и расположите их по разные стороны направляющей.

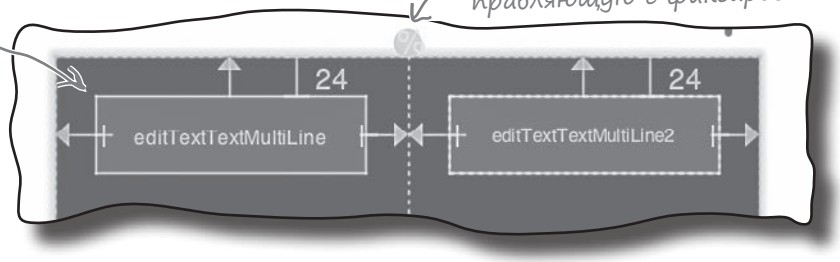
Добавьте два вертикальных ограничения, которые связывают каждое представление с верхним краем эскиза, и горизонтальные ограничения для размещения представлений между краем эскиза и направляющей.

Наконец, измените атрибут `layout_width` каждого текстового поля и присвойте ему значение 0dp, чтобы его размеры подгонялись под горизонтальные ограничения. Присвойте `layout_height` значение «wrap_content», чтобы представления могли расширяться.

Когда все будет сделано, эскиз должен выглядеть примерно так:

Чтобы упростить размещение двух представлений, мы используем вертикальную направляющую с фиксированной позицией 50%.

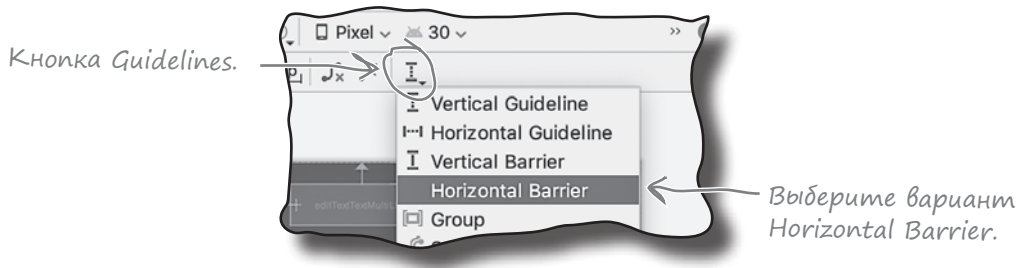
Каждое текстовое поле размещается между краем эскиза и направляющей.



Отдельное представление
Несколько представлений

Добавление горизонтального барьера

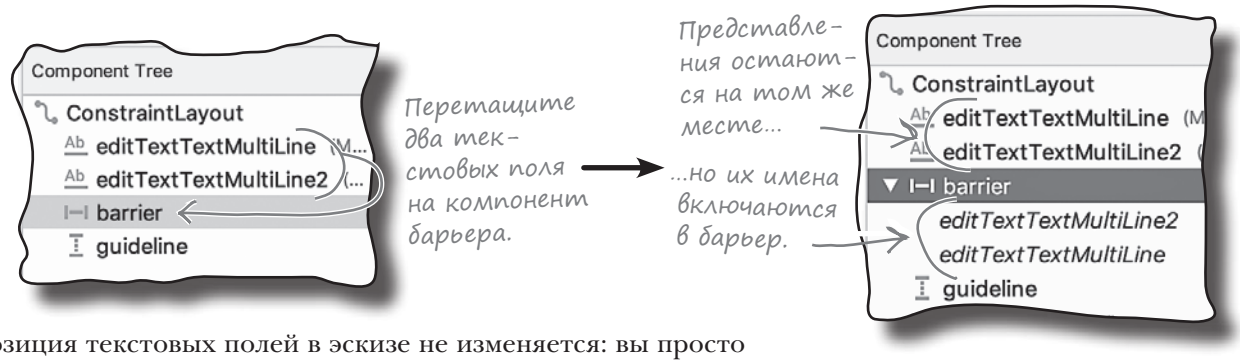
В эскиз необходимо включить горизонтальный барьер. Для этого щелкните на кнопке Guidelines на панели инструментов визуального редактора и выберите вариант Horizontal Barrier:



В результате будет создан горизонтальный барьер.

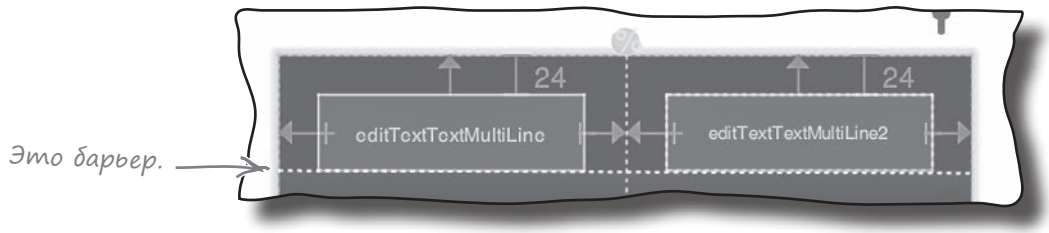
Размещение барьера под представлениями

Барьер должен перемещаться вниз при расширении двух текстовых полей. Перейдите на панель с деревом компонентов макета и перетащите два компонента текстового поля на барьер:



Позиция текстовых полей в эскизе не изменяется: вы просто сообщаете барьеру, что он должен смещаться вместе с представлениями.

Затем необходимо разместить барьер так, чтобы он находился у нижней стороны двух представлений. Выделите барьер в дереве компонентов и на панели Attributes задайте атрибуту `barrierDirection` значение «bottom». Барьер размещается под двумя текстовыми полями, в результате эскиз выглядит так:



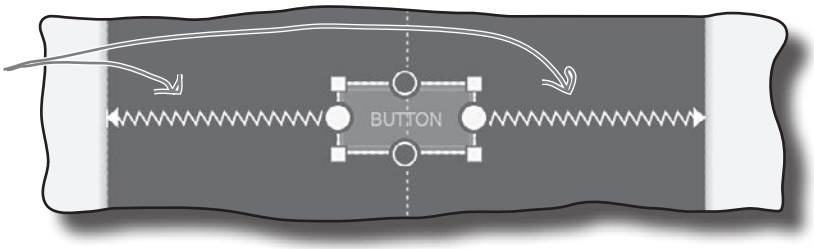
Отдельное представление
Несколько представлений

Размещение кнопки под барьером

Итак, барьер макета находится на своем месте. Добавим кнопку и свяжем ее с барьером, чтобы она смещалась вниз при расширении текстовых полей.

Сначала перетащите кнопку с палитры на эскиз и разместите ее под барьером. Затем выровняйте ее горизонтально по центру, добавив два горизонтальных ограничения, связывающих стороны кнопки с краями эскиза:

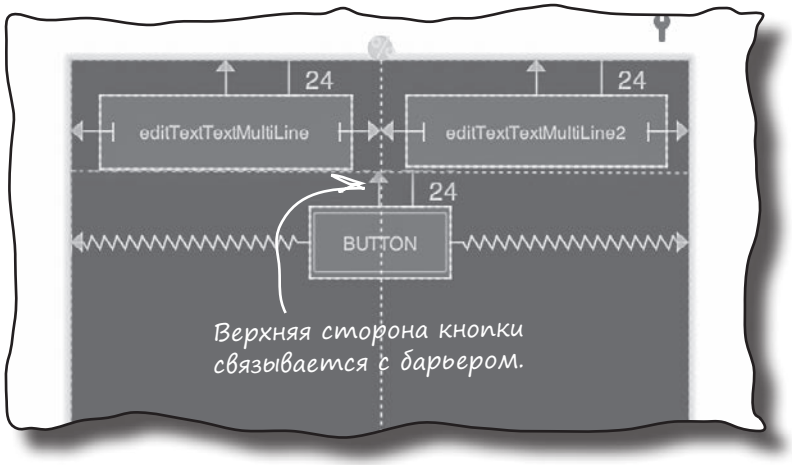
Горизонтальные ограничения перемещают кнопку в центр эскиза.



Затем верхнюю сторону кнопки необходимо связать с барьером. Для этого можно просто провести линию ограничения прямо на эскизе. Если вам (как и нам) этот способ кажется слишком хлопотным, найдите атрибут `layout_constraintTop_toBottomOf` на панели Attributes и присвойте ему идентификатор барьера (в нашем примере `id/barrier`).

Для нахождения нужных атрибутов можно воспользоваться функцией поиска в верхней части панели Attributes.

После внесения этого изменения эскиз должен выглядеть примерно так:



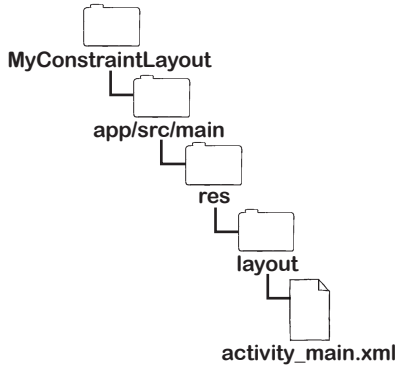
Поначалу работа с барьерами может создавать определенные трудности, поэтому на паре ближайших страниц приводится полная разметка XML, а затем мы опробуем приложение на практике.

Отдельное представление
Несколько представлений

Полный код activity_main.xml

Ниже приведен полный код *activity_main.xml*. Если вы хотите, чтобы ваши макеты не отличались от наших, замените содержимое этого файла приведенным здесь:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/editTextTextMultiLine"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:layout_marginEnd="24dp"
        android:ems="10"
        android:gravity="start|top"
        android:inputType="textMultiLine"
        app:layout_constraintEnd_toStartOf="@+id/guideline"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/editTextTextMultiLine2"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:layout_marginEnd="24dp"
        android:ems="10"
        android:gravity="start|top"
        android:inputType="textMultiLine"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="@+id/guideline"
        app:layout_constraintTop_toTopOf="parent" />
```



Два многострочных текстовых поля. Они ограничиваются по верхнему краю макета, его стороне и направляющей.

Продолжение на следующей странице.

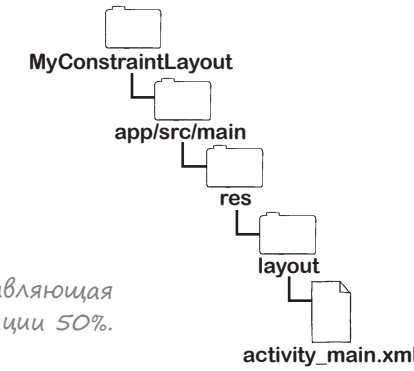
Отдельное представление
Несколько представлений

Полный код activity_main.xml (продолжение)

```
<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:barrierDirection="bottom"
    app:constraint_referenced_ids="editTextTextMultiLine2,editTextTextMultiLine"
    tools:layout_editor_absoluteY="731dp" />

<androidx.constraintlayout.widget.Guideline
    android:id="@+id/guideline"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintGuide_percent="0.5" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:text="Button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/barrier" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Барьер перемещается с двумя текстовыми полями и связывается с их нижними сторонами.

Направляющая в позиции 50%.



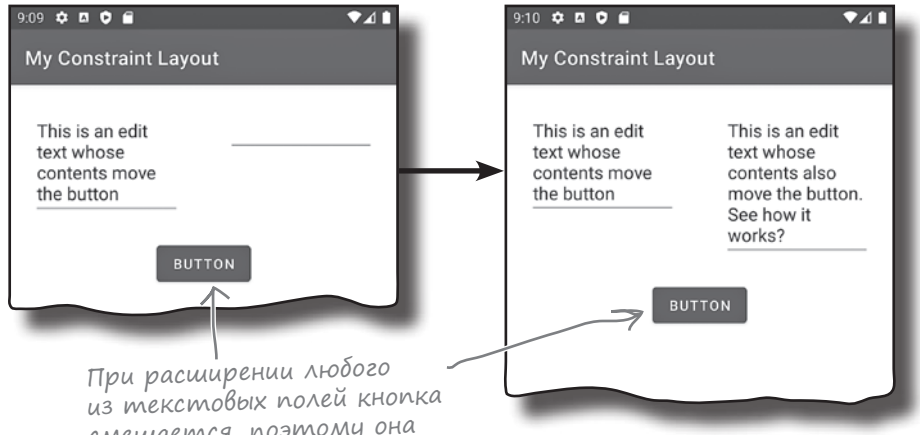
Кнопка связывается ограничениями с каждой стороной макета и с барьером.

Тест-драйв

При запуске приложения кнопка выводится под двумя текстовыми полями. Кнопка смещается вниз при вводе текста в любом из текстовых полей и их расширении.

Как видите, добавить барьер несколько сложнее, чем проводить ограничения и выравнивать представления, но, на наш взгляд, результат того стоит.

Что же дальше?



При расширении любого из текстовых полей кнопка смещается, поэтому она всегда остается под ними.

Использование цепочек для управления линейными группами представлений

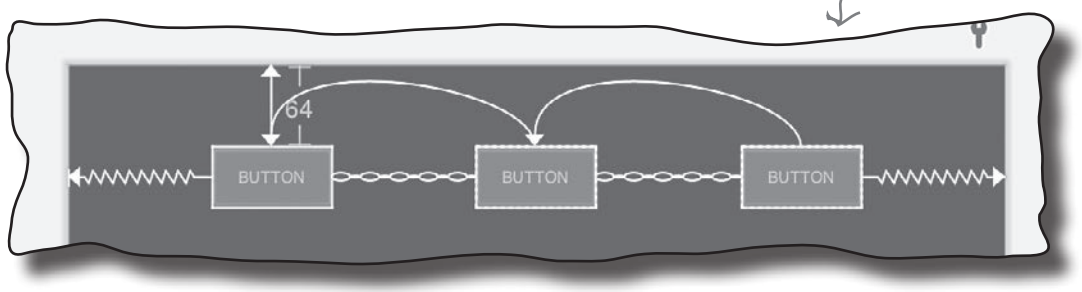
Вы научились соединять и выравнивать представления, а также пользоваться направляющими и ограничениями. Но что, если вы хотите создать строку или столбец представлений, разделенных равномерными интервалами?

В подобных ситуациях можно воспользоваться **цепочкой** — линейной группой представлений, связанных вместе двусторонними ограничениями. Цепочка управляет позицией каждого представления и может использоваться для равномерного распределения представлений или размещения их в центре эскиза.

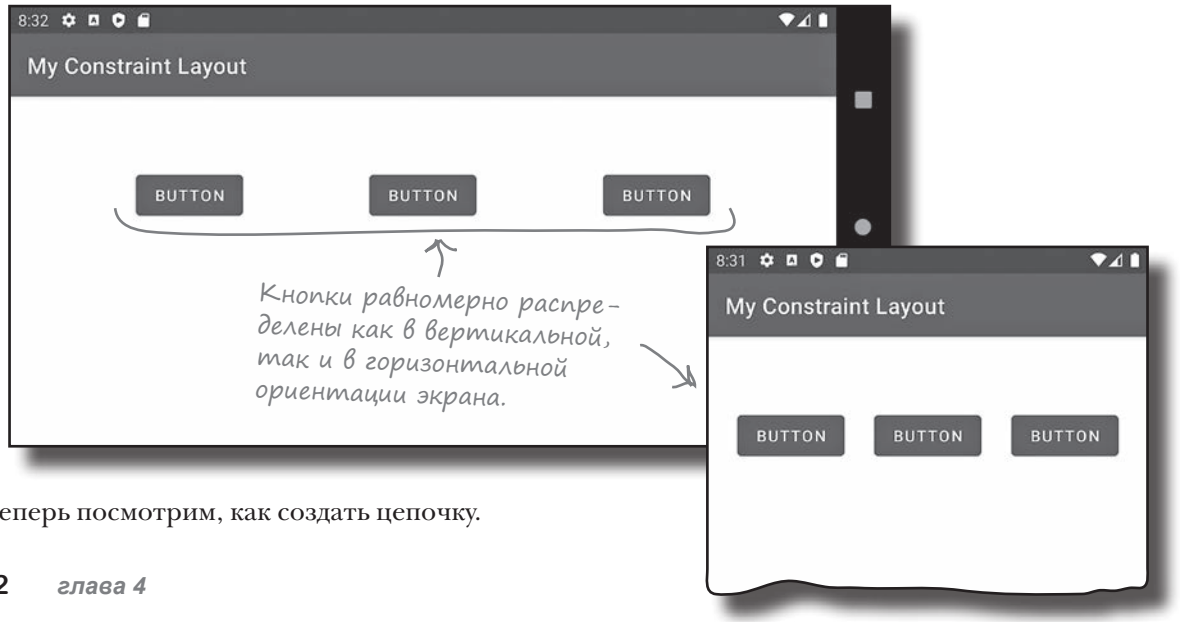
Мы создадим горизонтальную цепочку

Чтобы увидеть, как работает этот механизм, мы создадим цепочку, управляющую позицией трех кнопок. Кнопки выстраиваются в горизонтальный ряд и равномерно распределяются между сторонами эскиза:

Горизонтальная цепочка из трех кнопок. Верхние стороны выровнены и равномерно распределены между сторонами эскиза.



При запуске приложения кнопки сохраняют свои относительные позиции независимо от размера или ориентации экрана:



А теперь посмотрим, как создать цепочку.

Отдельное представление
Несколько представлений

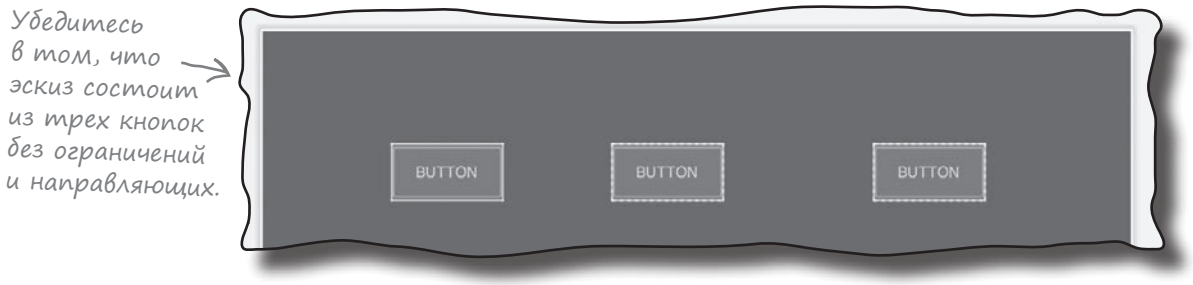
Цепочка состоит из трех кнопок

Прежде чем создавать цепочку, сначала удалите все ограничения, которые были добавлены в эскиз. Быстрее всего для этого воспользоваться кнопкой Clear All Constraints на панели инструментов визуального редактора.

Также необходимо удалить все направляющие, барьеры и текстовые поля. Последовательно выделите каждое текстовое поле и удалите его.

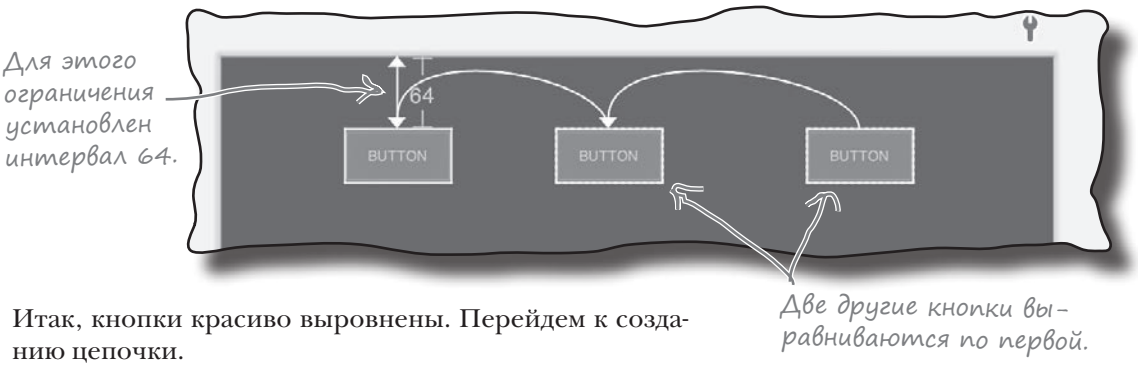
Затем добавьте в эскиз еще две кнопки, чтобы их стало три, и воспользуйтесь кнопкой «Orientation for Preview» на панели инструментов визуального редактора, чтобы переключить эскиз на горизонтальную (альбомную) ориентацию. Так вам будет проще увидеть цепочку.

После добавления кнопки эскиз должен выглядеть примерно так:



Выравнивание представлений, которые будут объединены в цепочку

Цепочки лучше всего работают с выровненными представлениями. Сначала добавьте ограничение, которое связывает первую кнопку с верхней стороной эскиза, и назначьте ему интервал 64. Затем выделите все три кнопки и воспользуйтесь кнопкой Align на панели инструментов визуального редактора для выравнивания их верхних сторон. Эскиз должен выглядеть примерно так:



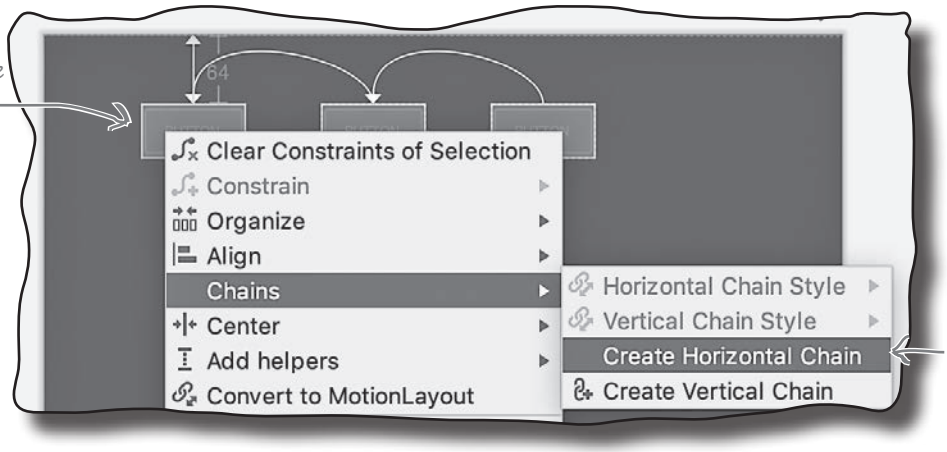
Итак, кнопки красиво выровнены. Перейдем к созданию цепочки.

Отдельное представление
Несколько представлений

Создание горизонтальной цепочки

Чтобы создать цепочку, выделите все три кнопки и щелкните правой кнопкой мыши на одной из них. В открывшемся меню выберите команду Chains, а затем вариант Create Horizontal Chain.

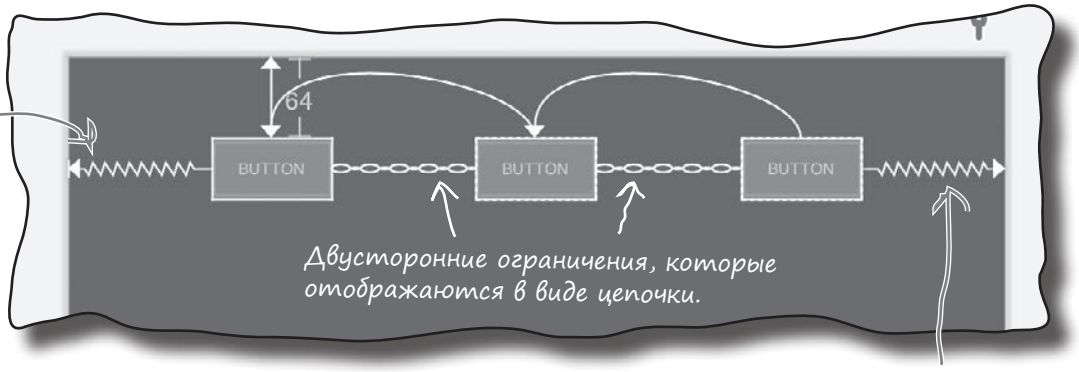
Выберите представления, которые вы хотите объединить в цепочку, а затем щелкните правой кнопкой мыши на одном из них.



Выберите вариант создания горизонтальной цепочки.

При создании горизонтальной цепочки кнопки объединяются, а первая и последняя закрепляются у вертикальных сторон эскиза. Цепочка должна выглядеть примерно так:

Ограничение, связывающее первую кнопку с левой стороной эскиза.



Это ограничение связывает последнюю кнопку с правой стороной эскиза.

По умолчанию представления в цепочке равномерно распределяются между сторонами эскиза. Чтобы изменить это поведение, щелкните правой кнопкой мыши на одном из представлений цепочки, выберите команду Chains в открывшемся меню, а затем выберите вариант Horizontal Chain Style.

Попробуйте догадаться, что делает каждый из вариантов. В этом вам поможет следующее упражнение.

Упражнение

Щелкните правой кнопкой мыши на одной из кнопок горизонтальной цепочки, выберите в открывшемся меню команду Chains, после чего выберите Horizontal Chain Style. Что происходит с представлениями при выборе каждого варианта? Нарисуйте ниже эскиз для каждого варианта. Опишите, как будут размещены представления.

Распределение (Spread):

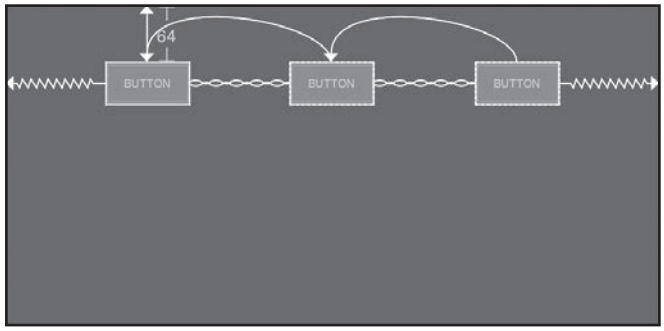
Внутреннее распределение (Spread inside):

Группировка (Packed):

Упражнение
Решение

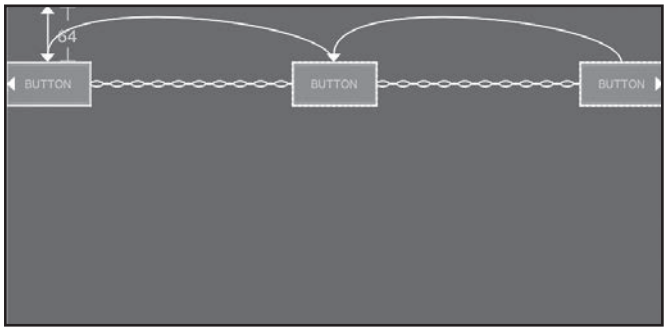
Щелкните правой кнопкой мыши на одной из кнопок горизонтальной цепочки, выберите в открывшемся меню команду Chains, после чего выберите Horizontal Chain Style. Что происходит с представлениями при выборе каждого варианта? Нарисуйте ниже эскиз для каждого варианта. Опишите, как будут размещены представления.

Распределение (Spread):



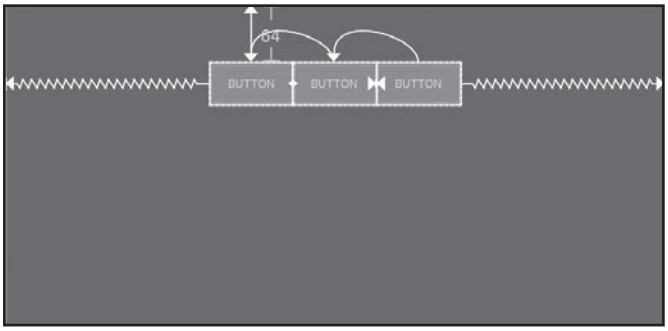
В варианте с распределением представления равномерно распределяются между сторонами эскиза.

Внутреннее распределение (Spread inside):



В варианте с внутренним распределением крайнее левое и крайнее правое представления смещаются к краям эскиза, а остальные представления равномерно распределяются между ними.

Группировка (Packed):



В варианте с группировкой представления группируются вблизи друг от друга, а вся группа выравнивается по центру.

Отдельное представление
Несколько представлений

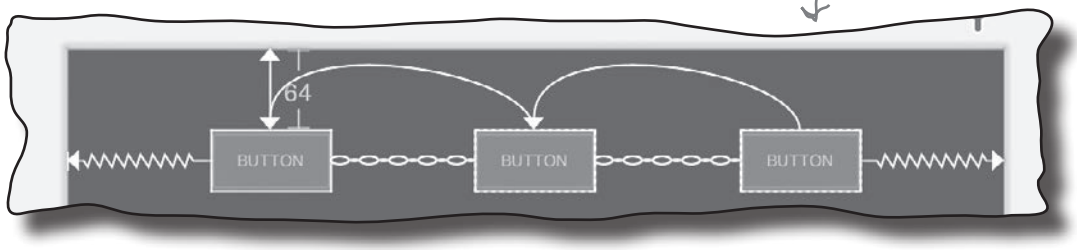
Разные стили цепочек

Как вы узнали, можно выбирать разные стили цепочек, чтобы изменить способ размещения представлений в цепочке.

При распределении представления равномерно распределяются между сторонами эскиза

По умолчанию используется стиль с **распределением**. Он используется для равномерного распределения представлений между сторонами эскиза:

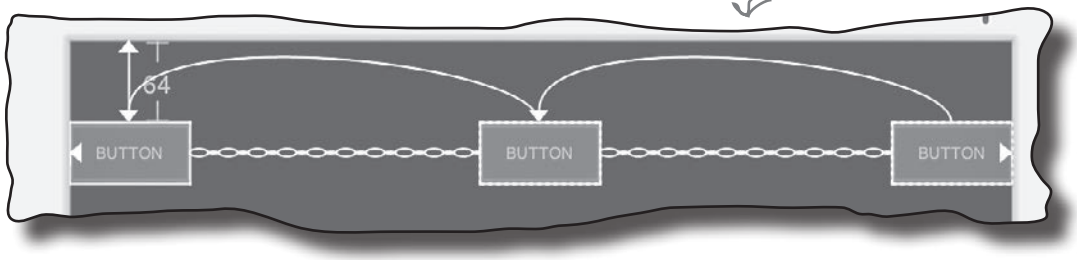
Представления распределяются равномерно.



При внутреннем распределении первое и последнее представления смещаются к сторонам

Стиль с **внутренним распределением** похож на стиль с распределением, не считая того, что первое и последнее представления смещаются к сторонам эскиза. После этого остальные представления равномерно распределяются между ними:

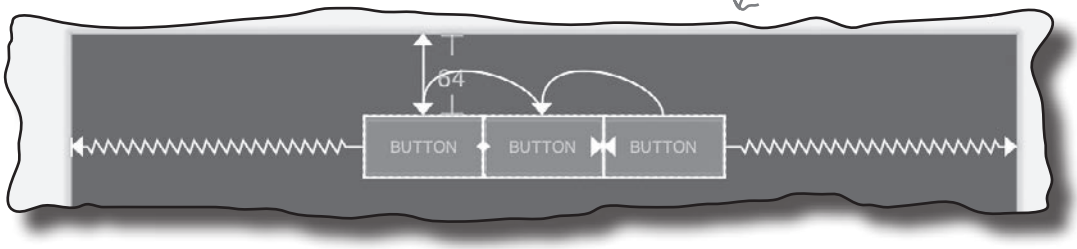
При внутреннем распределении внешние представления смещаются к краям эскиза.



Группировка сводит представления вместе

Стиль с **группировкой** используется для размещения представлений вблизи друг от друга. После этого вся группа представлений выравнивается по центру:

Группировка плотно размещает представления в центре.

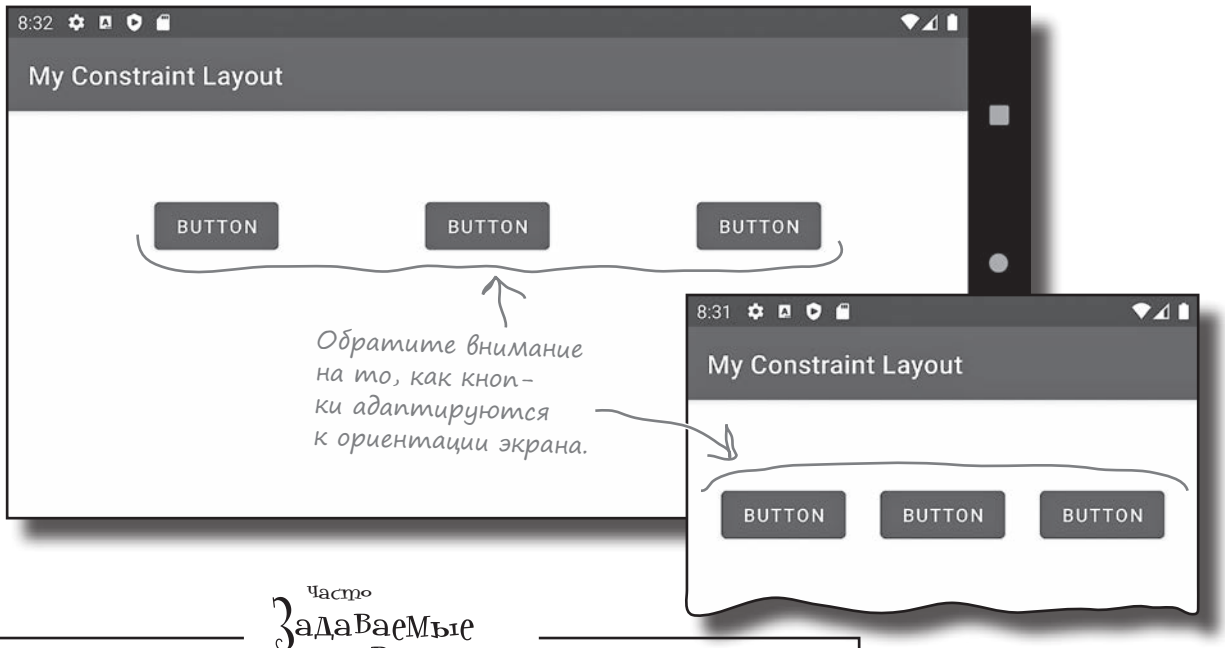


Теперь вы знаете, что делает каждый из этих вариантов.
Давайте опробуем приложение на практике.

Тест-драйв

Отдельное представление
Несколько представлений

Когда мы выбираем цепочку с распределением, кнопки равномерно распределяются на экране устройства. Распределение работает независимо от ориентации экрана.



Часто Задаваемые Вопросы

В: Могу ли я задать фиксированный размер представлений?

О: Да. Обычно так поступать не рекомендуется, потому что это означает, что представления не могут увеличиваться и уменьшаться по размерам содержимого или с учетом размеров экрана.

Но если вы уверены в том, что вам нужно представление фиксированного размера, вы можете либо обновить его атрибуты, либо использовать манипуляторы для изменения его размеров в эскизе.

В: Могу ли я присоединить цепочку к направляющей?

О: Да, можете.

В: Что произойдет, если сделать представление невидимым?

О: Все зависит от того, как именно вы это сделаете.

Если присвоить атрибуту `visibility` представления значение `invisible`, то представление не будет отображаться на экране, а его место останется пустым.

Если присвоить значение `gone`, то представление не занимает места на экране, а цепочки и т. д. будут работать так, словно представления нет.

Всегда тестируйте макеты с разными размерами и ориентациями экранов устройств и убедитесь в том, что они выглядят и работают именно так, как было задумано.

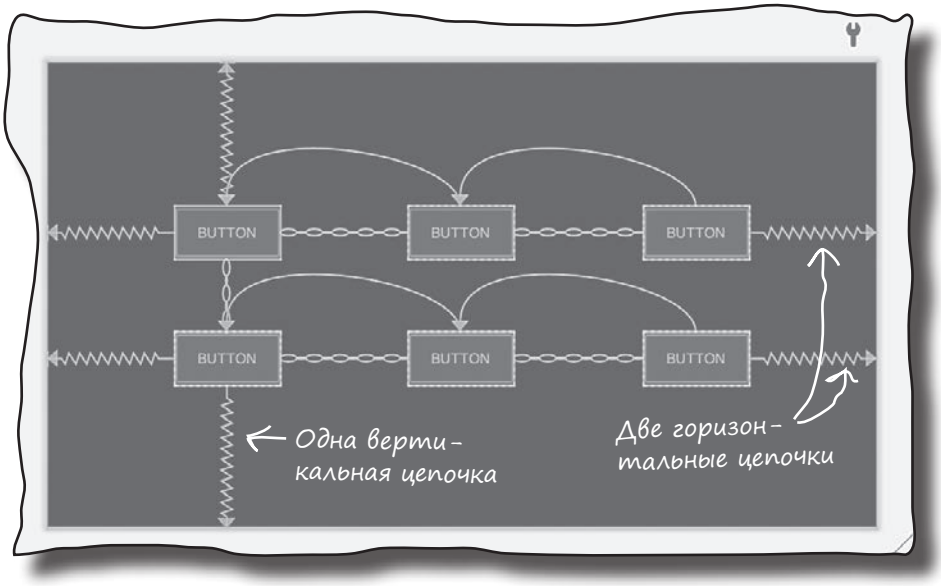
Отдельное представление
Несколько представлений

Итак, мы создали горизонтальную цепочку, но также есть варианты создания вертикальных цепочек. И я вот что подумала... Может ли представление входить как в горизонтальную, так и в вертикальную цепочку? И если может, то можно ли использовать его для создания сетки представлений?

Она права.

Макеты с ограничениями могут включать как горизонтальные, так и вертикальные цепочки, и одно представление может принадлежать цепочкам обоих типов. Этот факт может использоваться для формирования сеток представлений.

Например, в следующем эскизе шесть кнопок выстроены в сетку. Каждая строка представлена горизонтальной цепочкой, а крайние левые кнопки образуют вертикальную цепочку:

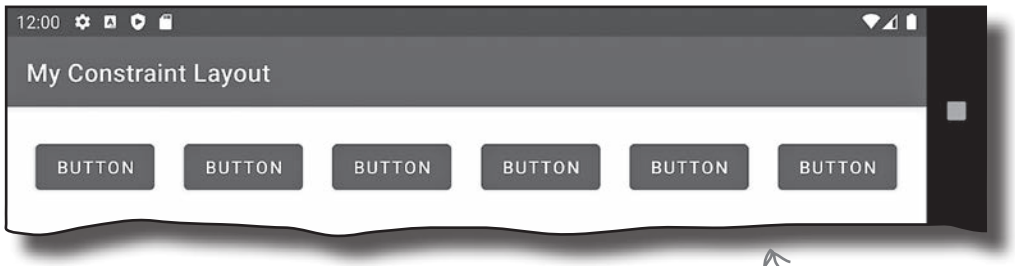


Другой способ создания сеток основан на использовании **потоков**. Давайте разберемся, что такое поток (flow) и как им пользоваться.

Поток похож на цепочку из нескольких строк

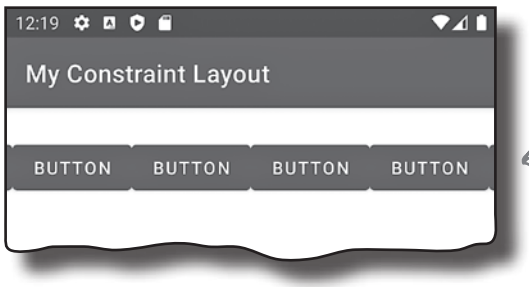
Поток напоминает цепочку, которая занимает сразу несколько строк. Например, потоки очень удобны для вывода нескольких представлений в ряд, если эти представления могут не поместиться на экране для некоторых размеров или ориентаций экранов.

Допустим, у вас имеется цепочка, которая выводит шесть кнопок в ряд по горизонтали. В горизонтальной (альбомной) ориентации устройства они отображаются так:



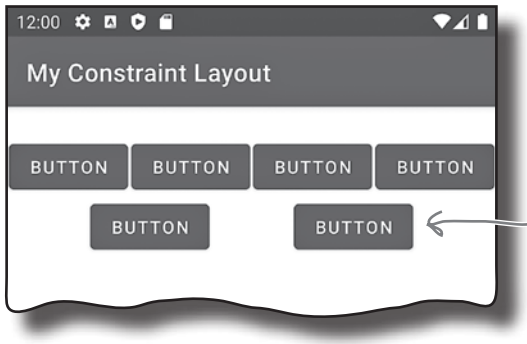
В горизонтальной ориентации кнопки помещаются на экране.

Но при переходе в вертикальную (книжную) ориентацию места для всех представлений не хватает:



Здесь места для вывода всех кнопок не хватает, поэтому кнопки выходят за край экрана.

Если заменить цепочку потоком, все представления, которые не помещаются в первой строке, «перетекают» во вторую строку:



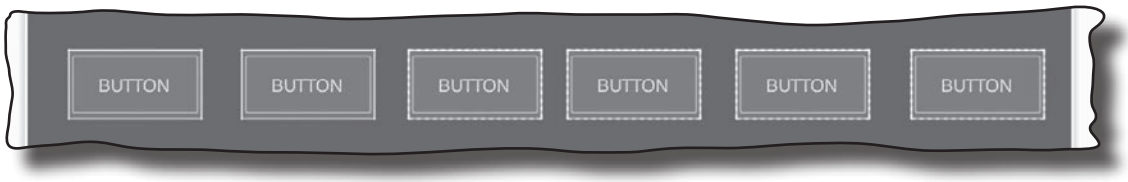
С потоком все представления, не помещающиеся в первой строке, образуют вторую строку.

Чтобы понять, как работают потоки, построим приведенный выше макет.

Отдельное представление
Несколько представлений

Добавление потока

Сначала удалите все ограничения при помощи кнопки Clear All Constraints на панели инструментов визуального редактора. Добавьте в эскиз новые кнопки, чтобы он содержал всего шесть кнопок:

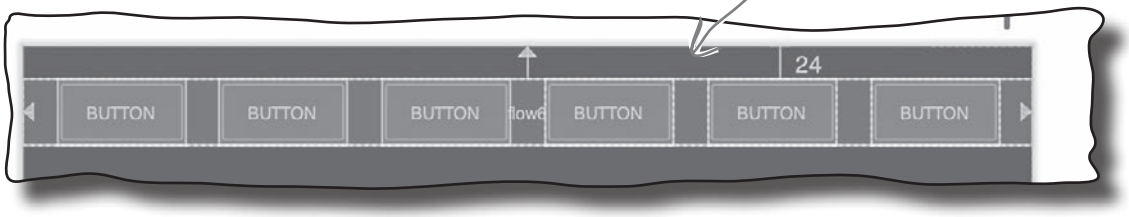


Затем выделите все кнопки, щелкните на кнопке Guidelines на панели инструментов визуального редактора и выберите вариант Flow. При этом в эскиз добавляется компонент потока.

Теперь необходимо изменить настройки компонента потока, чтобы наделить его нужным поведением. Для этого выделите поток в дереве компонентов, а затем используйте виджет эскиза или ограничения, чтобы добавить ограничения, связывающие боковые стороны и верхнюю сторону со сторонами эскиза. Задайте атрибуту `layout_width` значение «0dp», чтобы размеры подбирались по ограничениям. Наконец, найдите атрибут `flow_wrapMode` на панели Attributes и присвойте ему значение «chain».

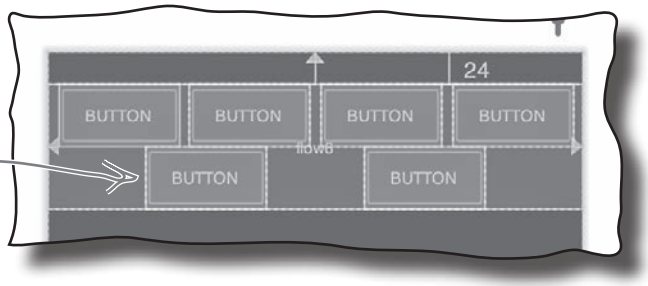
После внесения всех изменений в горизонтальной ориентации эскиз должен выглядеть примерно так:

Все кнопки добавлены в поток. Поток ограничивается по боковым сторонам и верхней стороне эскиза.



Если переключиться на вертикальную ориентацию, эскиз должен выглядеть примерно так:

Все кнопки, не помещающиеся в первой строке, «перетекают» во вторую.



Когда поток будет создан, вы сможете настроить режим отображения его представлений. Давайте посмотрим, как это делается.

Отдельное представление
Несколько представлений

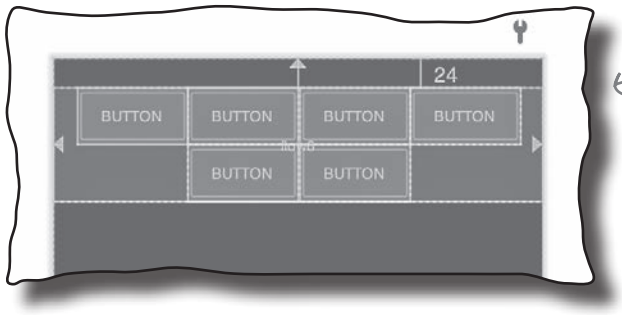
Управление внешним видом потока

Основной способ управления внешним видом потока основан на использовании его атрибута **`flow_wrapMode`**.

Использование режима «chain» для создания многострочных цепочек

Если присвоить атрибуту `flow_wrapMode` значение `chain`, поток ведет себя как гибкая цепочка, представления которой «перетекают» в следующие строки.

С этим режимом можно дополнительно настраивать внешний вид потока, изменяя значение его атрибута `flow_horizontalStyle`. Возможные значения этого атрибута — spread, spread inside и packed. С цепочками они работают точно так же, как с потоками. Например, режим packed группирует представления:

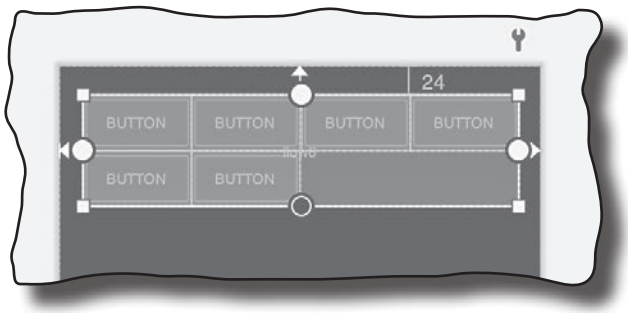


Результат, который вы получите, присвоив flow_wrapMode значение chain и используя горизонтальный стиль packed.

Выравнивание представлений в режиме «aligned»

Если присвоить атрибуту `flow_wrapMode` значение `aligned`, представления «перетекают» в следующие строки и выравниваются так, как показано ниже:

Обратите внимание на выравнивание представлений. Этот режим может использоваться для вывода представлений в виде сетки.



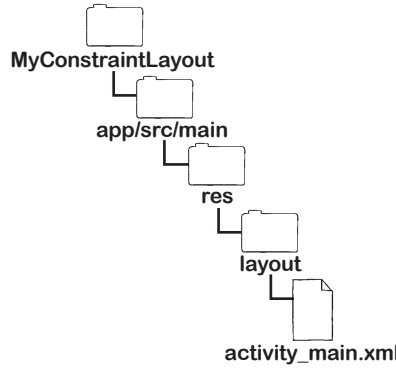
Атрибуту `flow_wrapMode` также можно задать значение `none` или оставить его без присваивания. В результате поток будет вести себя как обычная цепочка, так что представления не будут «перетекать» во вторую строку.

Отдельное представление
Несколько представлений

Полный код activity_main.xml

Иногда бывает трудно заставить поток работать так, как вам нужно, поэтому ниже приведен полный код файла *activity_main.xml*. Если потребуется, измените код в этом файле так, чтобы он соответствовал приведенному:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button6"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />
```



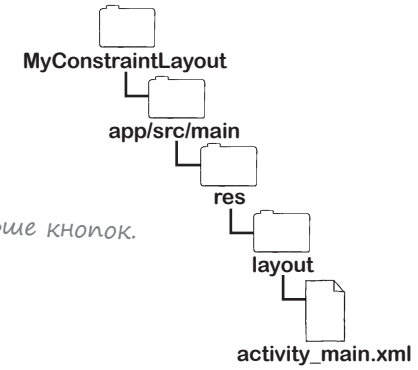
Четыре кнопки, которые мы добавили.

Продолжение на следующей странице.

Отдельное представление
Несколько представлений

activity_main.xml (продолжение)

```
    <Button
        android:id="@+id/button7"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <Button
        android:id="@+id/button8"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />

    <androidx.constraintlayout.helper.widget.Flow
        android:id="@+id/flow"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        app:constraint_referenced_ids=
            "button3,button4,button5,button6,button7,button8"
        app:flow_horizontalStyle="spread"
        app:flow_wrapMode="chain"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```



Еще больше кнопок.

А это поток.

Теперь поток включает эти представления.

Эти атрибуты определяют способ размещения представлений.

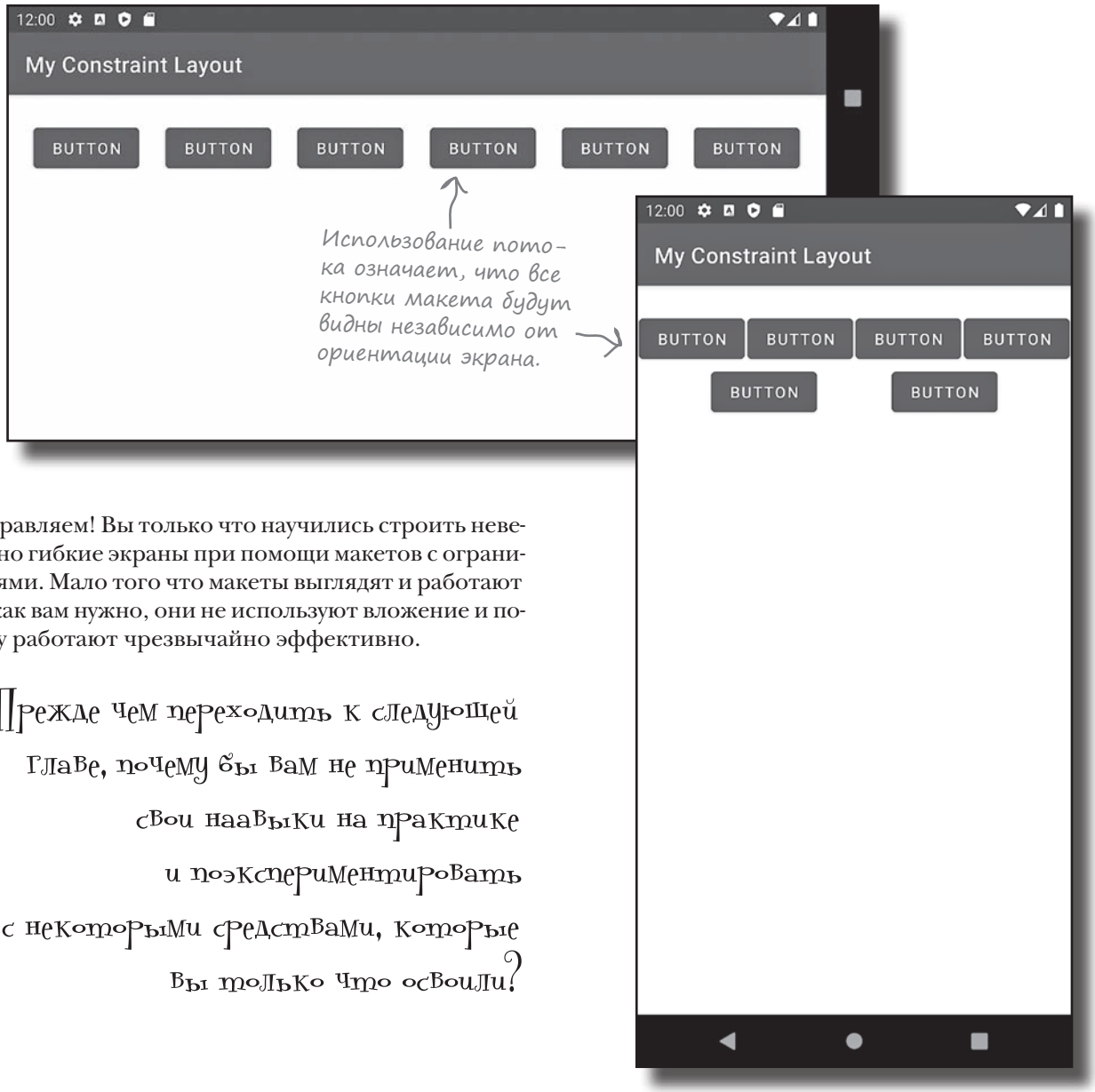
Пора опробовать наше приложение на практике.

Тест-драйв

Отдельное представление
Несколько представлений

Если использовать поток со стилем цепочки и запустить приложение, кнопки равномерно распределяются на экране устройства при горизонтальной ориентации.

Если перейти к вертикальной ориентации, все кнопки, не помещающиеся в первой строке, «перетекают» во вторую строку.

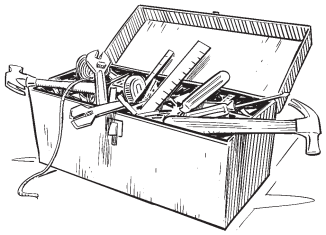


Поздравляем! Вы только что научились строить невероятно гибкие экраны при помощи макетов с ограничениями. Мало того что макеты выглядят и работают так, как вам нужно, они не используют вложение и поэтому работают чрезвычайно эффективно.

Прежде чем переходить к следующей главе, почему бы вам не применить свои навыки на практике и поэкспериментировать с некоторыми средствами, которые вы только что освоили?

Ваш инструментарий Android

ГЛАВА 4



Глава 4 осталась позади, а ваш инструментарий пополнился макетами с ограничениями.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Макеты с ограничениями разрабатывались для работы в визуальном редакторе Android Studio.
- Для размещения представлений используются ограничения. Каждое представление должно иметь хотя бы одно горизонтальное и одно вертикальное ограничение.
- Чтобы выровнять представления по центру, добавьте ограничения к противоположным сторонам представления. Измените смещение представления, чтобы изменить его позицию между ограничениями.
- Размер представления можно изменить по размерам ограничений, если представлению назначены ограничения на противоположных сторонах.
- Представления можно выровнять относительно друг друга, или же воспользоваться направляющей либо барьером.
- Направляющая размещается на фиксированном расстоянии или в процентном соотношении от стороны эскиза.
- Барьер смещается при изменении размеров представлений.
- Используйте цепочки для управления линейными группами представлений. Цепочки могут быть горизонтальными или вертикальными.
- Горизонтальные и вертикальные цепочки могут использоваться совместно для построения сеток представлений.
- В стиле цепочек с распределением представления распределяются равномерно.
- В стиле цепочек с внутренним распределением внешние представления смещаются к сторонам, а внутренние представления распределяются равномерно.
- В стиле цепочек с группировкой представления группируются в центре.
- Поток представляет собой многострочную цепочку.

Из жизни активностей

Активности образуют основу любого Android-приложения. Ранее вы видели, как создавать активности и как использовать активности для взаимодействия с пользователем. Но если вы еще не знакомы с **жизненным циклом активностей**, некоторые аспекты их поведения могут вас застать врасплох. В этой главе вы узнаете, что происходит при **создании** или **уничтожении** активностей и к каким **неожиданным последствиям** это может привести. Вы узнаете, как управлять их поведениями, когда они становятся **видимыми** или **скрываются**. Вы даже научитесь **сохранять** и **восстанавливать состояние активности**, когда это потребуется. Просто продолжайте читать дальше…

Как на самом деле работают активности?

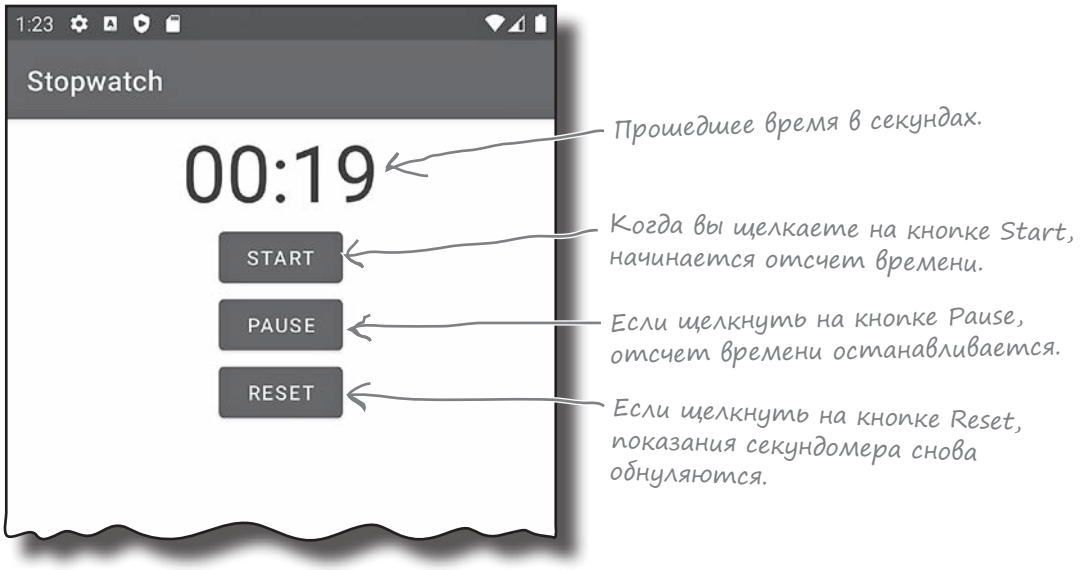
К настоящему моменту вы узнали, как создать интерактивное приложение, определяя поведение приложения в коде активности. Однако существует еще много такого в работе активностей, чего вы не знаете.

Как долго существует активность? Что происходит при исчезновении активности с экрана? Продолжает ли она выполняться? Находится ли в памяти? И что происходит, когда ваше приложение прерывается входящим телефонным звонком?

Вы хотите иметь возможность управлять поведением ваших активностей в *самых разных обстоятельствах*, поэтому пришло время глубже разобраться в том, как на самом деле работают активности. Мы рассмотрим некоторые типичные причины нарушения работоспособности ваших приложений и разберемся, как исправить их при помощи **методов жизненного цикла активностей**. Эти функции будут рассматриваться на примере приложения-секундомера.

Приложение Stopwatch

Приложение Stopwatch содержит четыре представления: надпись с прошедшим временем, кнопку Start для запуска секундомера, кнопку Pause для его приостановки и кнопку Reset, которая снова обнуляет показания таймера. Приложение выглядит так:



Приложение будет построено на нескольких ближайших страницах. Начнем с создания нового проекта.

Создание нового проекта

Создайте новый проект Android Studio по той же схеме, которая использовалась в предыдущих главах. Выберите вариант Empty Activity, введите имя «Stopwatch» с именем пакета «com.hfad.stopwatch» и подтвердите место сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android.

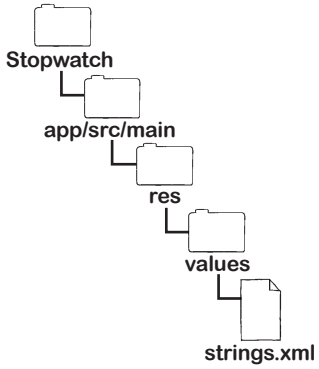
Добавление строковых ресурсов для надписей

Макет Stopwatch включает три кнопки с надписями Start, Pause и Reset. Мы добавим эти кнопки в файл *strings.xml* в виде строковых ресурсов, чтобы приложение могло загружать их значения во время выполнения.

Чтобы добавить строковые ресурсы, откройте файл *strings.xml* в папке *app/src/main/res/values* и обновите его (изменения выделены жирным шрифтом):

```
<resources>
    <string name="app_name">Stopwatch</string>
    <string name="start">Start</string>
    <string name="pause">Pause</string>
    <string name="reset">Reset</string>
</resources>
```

Эти строковые ресурсы используются в макете.



Итак, строковые ресурсы добавлены, перейдем к построению макета.

Макет включает представление Chronometer

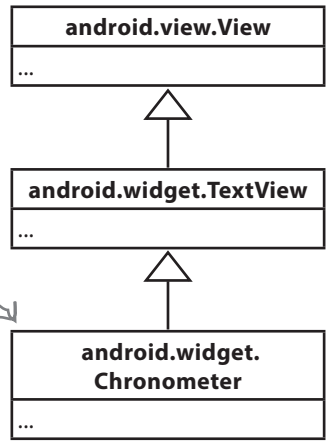
Макет состоит из трех кнопок и представления, в котором выводятся показания секундомера в секундах. Для вывода показаний будет использоваться **хронометр**.

Хронометр — разновидность надписи, которая выполняет функции простого таймера. Хронометр содержит встроенные методы, которые могут использоваться для запуска и остановки отсчета времени, и выводит прошедшее время.

Для включения хронометра в макет используется элемент `<Chronometer>`. Например, следующий код добавляет представление `Chronometer` с идентификатором «stopwatch», размер которого определяется размером содержимого:

```
<Chronometer
    android:id="@+id/stopwatch"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Chronometer — разновидность TextView с функциональностью таймера.



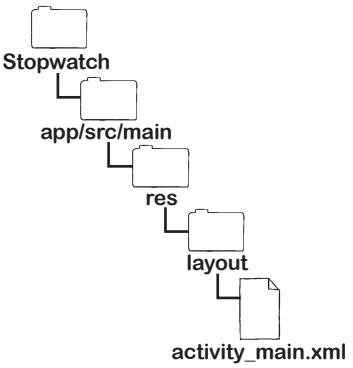
Когда мы займемся написанием кода активности, вы больше узнаете о том, как работают хронометры. Но сначала рассмотрим полный код макета.

Полный код activity_main.xml

Полный код *activity_main.xml* содержит представление `Chronometer` и три кнопки. Обновите файл *activity_main.xml*, чтобы он содержал приведенный ниже код:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".MainActivity">

    <Chronometer
        android:id="@+id/stopwatch"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="56sp" />

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start" />

    <Button
        android:id="@+id/pause_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pause" />

    <Button
        android:id="@+id/reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/reset" />
</LinearLayout>
```

Выравнивает содержимое макета горизонтально по центру.



Использует атрибут textSize, чтобы показания хронометра были крупными и хорошо читались.

Код добавляет кнопки Start, Pause и Reset.

Не забудьте!

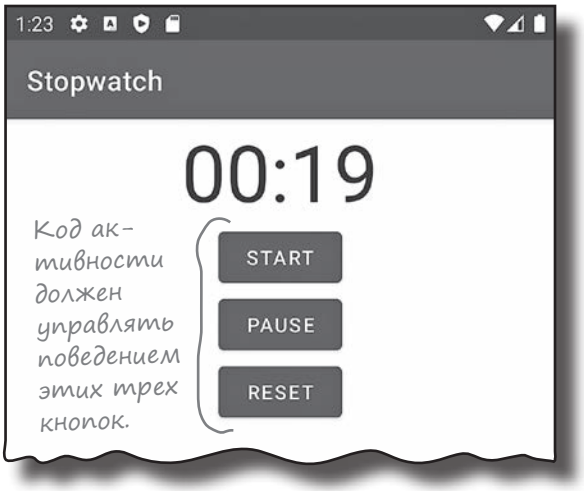
Прежде чем следовать дальше, обязательно обновите макет и код strings.xml.

Макет завершен, можно переходить к активности.

Код активности управляет работой хронометра

Код активности должен определить, каким образом кнопки Start, Pause и Reset управляют работой секундомера. Кнопка Start должна запускать секундомер (если он не был запущен ранее), кнопка Pause должна останавливать секундомер, а кнопка Reset — обнулять его показания.

Для управления секундомером кнопки будут обращаться к встроенным свойствам и методам `Chronometer`. Рассмотрим эти свойства и методы.



Ключевые свойства и методы Chronometer

Как говорилось ранее, представление `Chronometer` является разновидностью `TextView`. Оно наследует все свойства и методы `TextView`, а также определяет ряд новых свойств и методов.

Ниже перечислены все свойства и методы `Chronometer`, которые будут использоваться в коде `MainActivity`:

Свойство base

Свойство `base` элемента `Chronometer` используется для определения базового времени, то есть времени, с которого начинается отсчет времени хронометром.

Чтобы назначить текущее время как базовое, следует задать свойству `base` результат вызова `SystemClock.elapsedRealtime()`:

```
stopwatch.base = SystemClock.elapsedRealtime()
```
← Обнуляет показания времени.

`SystemClock.elapsedRealtime()` возвращает время в миллисекундах, прошедшее с момента загрузки устройства. Если присвоить свойству `base` это значение, выводимое время обнуляется.

Метод start()

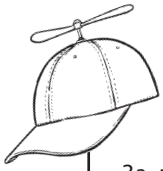
Метод `start()` запускает отсчет от базового времени:

```
stopwatch.start()
```
← Запускает хронометр.

Метод stop()

Приостанавливает отсчет времени хронометром:

```
stopwatch.stop()
```
← Приостанавливает хронометр.



Серьезное программирование

За дополнительной информацией о представлении Chronometer обращайтесь к электронной документации:

https://developer.android.com/reference/android/widget/Chronometer

Теперь вы знаете, какие свойства и методы представления `Chronometer` будут использоваться, и мы можем перейти к рассмотрению кода активности.

Полный код MainActivity.kt

Код `MainActivity` требует назначения слушателя `OnClickListener` для каждой кнопки. Слушатель кнопки Start должен запустить секундомер, если он не работает в настоящий момент, слушатель кнопки Pause должен секундомер останавливать, а слушатель кнопки Reset — обнулять показания секундомера.

В этом нам помогут три свойства: `stopwatch`, `running` и `offset`.

Свойство `stopwatch` содержит ссылку на представление `Chronometer`.

Свойство `running` содержит признак того, работает ли секундомер в настоящий момент. При щелчке на кнопке Start ему присваивается значение `true`, а при щелчке на кнопке Pause — значение `false`.

Свойство `offset` используется для вывода правильного времени на секундомере, если он был приостановлен и запущен заново. Без этого на секундомере выводилось бы неправильное время.

Также были включены два метода (`setBaseTime` и `saveOffset`), которые упрощают чтение кода.

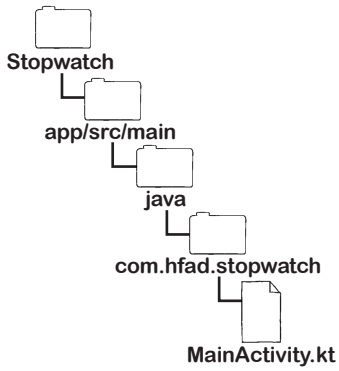
Ниже приведен полный код *MainActivity.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer

class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  // Хронометр
    var running = false  // Хронометр работает?
    var offset: Long = 0  //Базовое смещение

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        //Получение ссылки на секундомер
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)
```

Эти библиотеки используются в коде, поэтому их необходимо импортировать.

Значения трех свойств используются для управления секундомером.

Получить ссылку на свойство stopwatch можно только после вызова setContentView. До этого представление Chronometer не существует.



Продолжение на следующей странице.

Полный код MainActivity.kt (продолжение)

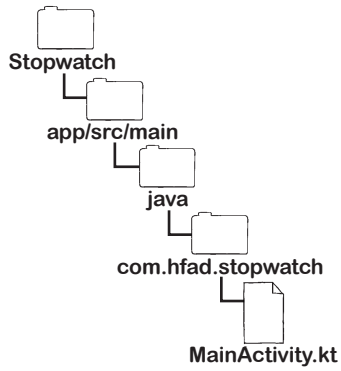
```
        //Кнопка start запускает секундомер, если он не работал
        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                stopwatch.start()
                running = true
            }
        }

        //Кнопка pause останавливает секундомер, если он работал
        val pauseButton = findViewById<Button>(R.id.pause_button)
        pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                stopwatch.stop()
                running = false
            }
        }

        //Кнопка reset обнуляет offset и базовое время
        val resetButton = findViewById<Button>(R.id.reset_button)
        resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    //Обновляет время stopwatch.base
    fun setBaseTime() {
        stopwatch.base = SystemClock.elapsedRealtime() - offset
    }

    //Сохраняет offset
    fun saveOffset() {
        offset = SystemClock.elapsedRealtime() - stopwatch.base
    }
}
```

Установить правильное время и запустить секундомер.

Сохранить время на секундомере и остановить его.

Обнулить показания секундомера.

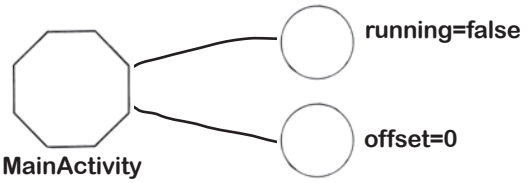
setBaseTime() и saveOffset() — вспомогательные методы, упрощающие чтение кода.



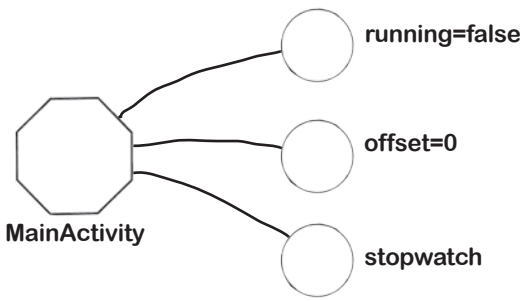
Давайте разберемся, как должен работать этот код.

Что происходит при запуске приложения

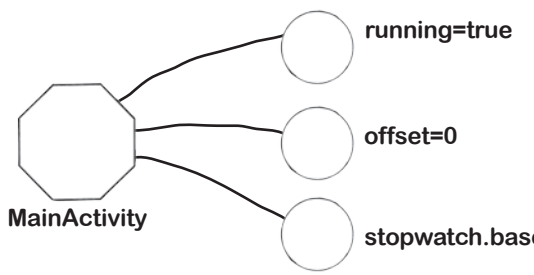
1. **Пользователь запускает приложение, MainActivity получает управление.**
Инициализируются свойства `running` и `offset`: `running` присваивается `false`, а `offset` присваивается 0.



2. **Вызывается метод onCreate активности MainActivity.**
С активностью связан макет *activity_main.xml*, а свойству `stopwatch` присваивается ссылка на представление `Chronometer`.



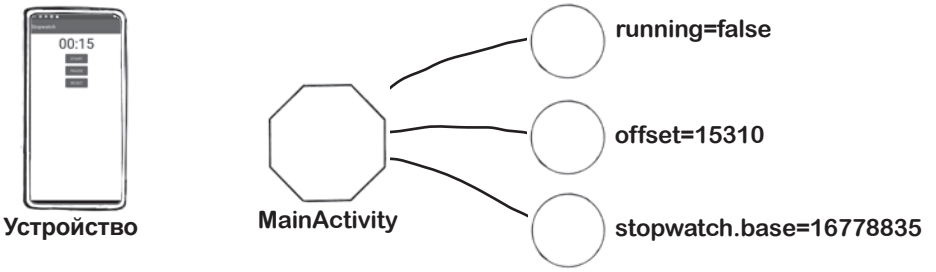
3. **Пользователь щелкает на кнопке Start.**
Свойству `stopwatch.base` присваивается значение `SystemClock.elapsedRealTime()`, вызывается его метод `start()`, а `running` присваивается `true`. Начинается отсчет времени.



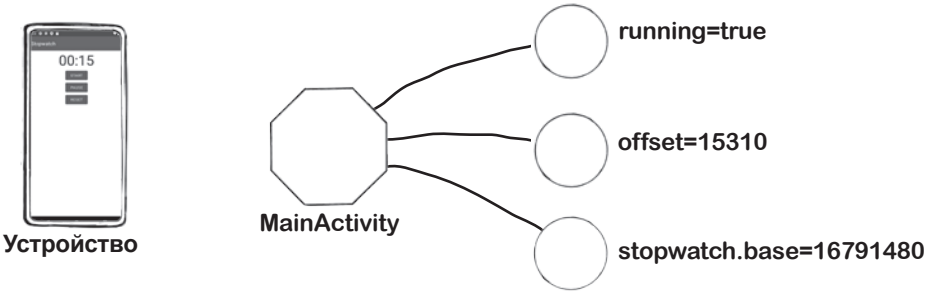
Секундомер использует время в миллисекундах от базового времени, то есть времени загрузки устройства.

История продолжается

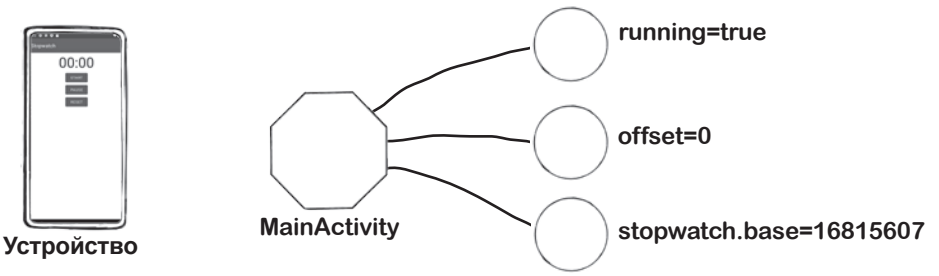
4 **Пользователь щелкает на кнопке Pause.**
`MainActivity` обновляет свойство `offset`, вызывает метод `stop()` и присвивает `running` значение `false`. Отсчет времени останавливается.



5 **Пользователь снова щелкает на кнопке Start.**
`MainActivity` использует значение `offset` для изменения свойства `stopwatch.base`, вызывает метод `start()` и присваивает `running` значение `true`. Отсчет времени запускается снова.



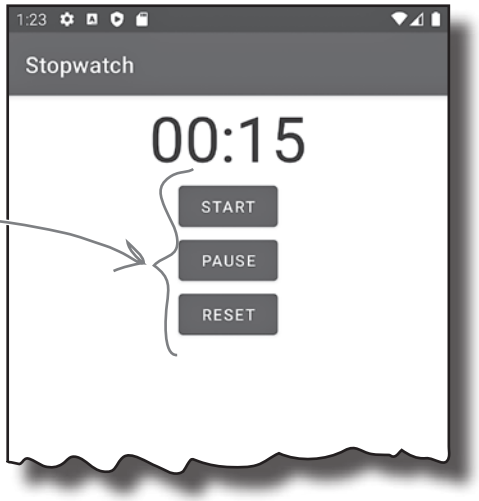
6 **Пользователь щелкает на кнопке Reset.**
Свойство `offset` обнуляется, а свойству `stopwatch.base` присваивается значение `SystemClock.elapsedRealTime()`.



А теперь опробуем приложение на практике.

Тест-драйв

При запуске приложения на первый взгляд все работает, как было задумано. Секундомер можно запускать, останавливать и обнулять без малейших проблем.



Эти кнопки работают так, как и ожидалось. Кнопка Start запускает отсчет времени, кнопка Pause останавливает его, а кнопка Reset снова обнуляет показания секундомера.

Только одна проблема...

Если повернуть устройство так, чтобы изменилась ориентация экрана, все идет насмарку. Секундомер обнуляется и перестает работать.

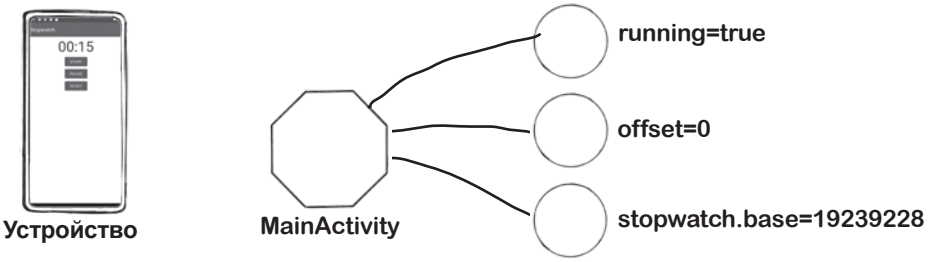
Если повернуть устройство (с включенным режимом автоповорота), секундомер обнуляется и перестает работать.

Почему приложение перестает работать при повороте экрана? Чтобы понять это, придется заглянуть под капот.

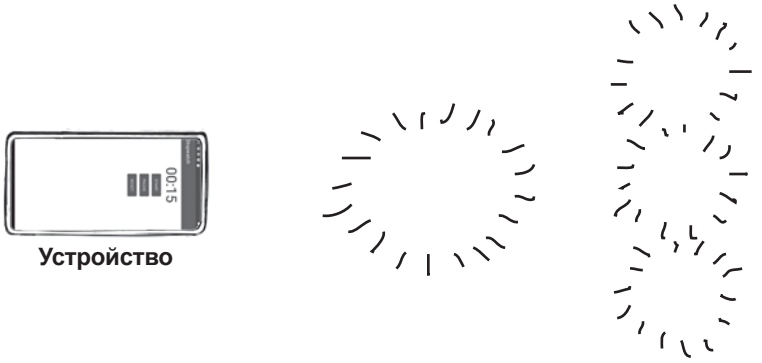
Что происходит при запуске приложения

При выполнении кода происходят следующие события:

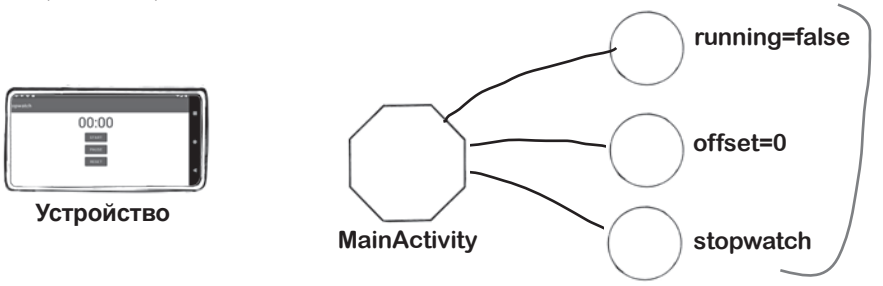
1 **Секундомер запускается кнопкой Start.**
Отсчет времени начинается, и свойству `running` присваивается значение `true`.



2 **Экран устройства поворачивается.**
Android видит, что ориентация экрана изменилась, и **уничтожает `MainActivity`**. Значения свойств активности теряются.



3 **MainActivity перезапускается.**
Активность создается заново, все свойства инициализируются заново, а `onCreate()` выполняется снова. Секундомер не перезапускается, так как метод `onCreate()` не отдал ему команду это сделать.

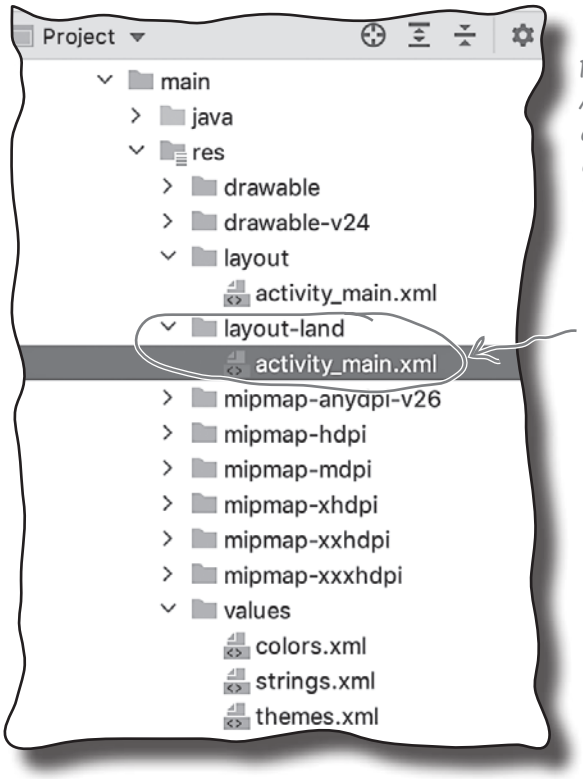


Значения всех свойств сбрасываются, поэтому показания обнуляются.

Поворот экрана изменяет конфигурацию устройства

Когда Android запускает приложение и передает управление активности, при этом учитывается **конфигурация устройства**. Под этим подразумевается как конфигурация физического устройства (размер экрана, ориентация экрана, наличие подключенной клавиатуры), так и параметры конфигурации, заданные пользователем (скажем, локальный контекст).

Система Android должна знать конфигурацию устройства при запуске активности, потому что конфигурация может влиять на то, какие ресурсы необходимы приложению. Например, приложение может использовать другой макет, если экран устройства находится в горизонтальном режиме вместо вертикального, и другой набор строковых значений, если выбран французский локальный контекст.



Приложения Android могут содержать несколько версий одного файла ресурсов. Например, если вы хотите использовать другую версию activity_main.xml, а устройство находится в горизонтальном режиме, этот файл следует поместить в папку layout-land.

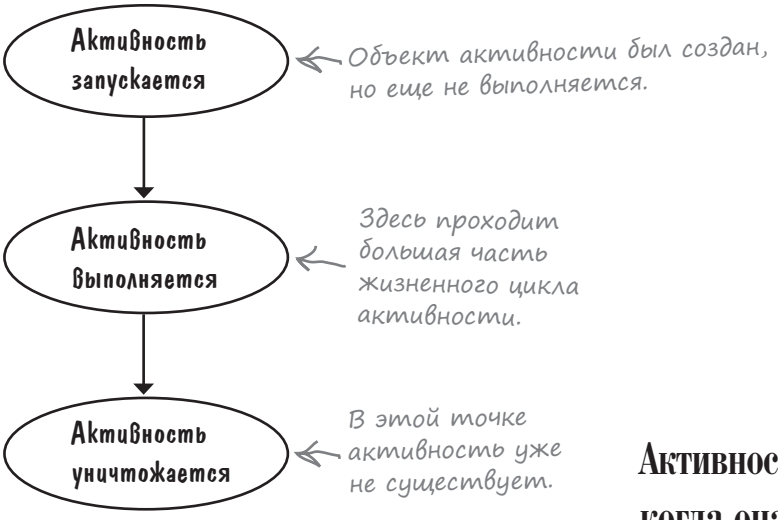
Конфигурация устройства включает как параметры, выбранные пользователем (например, локальный контекст), так и параметры, относящиеся к физическому устройству (например, ориентация и размер экрана). При изменении любых из этих параметров активность уничтожается, а потом создается заново.

При изменении конфигурации устройства все компоненты, отображающие пользовательский интерфейс, должны быть обновлены для новой конфигурации. Например, при повороте экрана Android замечает, что ориентация изменилась, и рассматривает это как изменение конфигурации устройства. Текущая активность уничтожается и создается заново, чтобы были загружены новые ресурсы, соответствующие новой конфигурации.

Состояние Выполнения

Когда Android создает и уничтожает активность, она переходит из состояния запуска в состояние выполнения, затем в состояние уничтожения.

Основное состояние активности — состояние ***выполнения*** (или ***активное*** состояние). Активность выполняется, когда она находится на переднем плане экрана, обладает фокусом и пользователь может взаимодействовать с ней. Активность проводит большую часть своего жизненного цикла в этом состоянии. Активность начинает выполняться после запуска, а в конце своего жизненного цикла активность ***уничтожается***.



Когда активность переходит в состояние запуска или в состояние уничтожения, срабатывают ключевые методы жизненного цикла активности: **`onCreate()`** и **`onDestroy()`**. Это методы жизненного цикла, наследуемые активностью, которые могут переопределяться.

Метод `onCreate()` вызывается сразу же после запуска активности. В этом методе происходит вся нормальная настройка активности, например вызов `setContentView()`. Этот метод должен переопределяться всегда. Если вы *не* переопределите его, вы не сможете сообщить Android, какой макет должен использоваться активностью.

Метод `onDestroy()` — последняя возможность что-то сделать перед уничтожением активности. Активность может уничтожаться в разных ситуациях: например, если она получила приказ завершиться, если активность создается заново из-за изменения конфигурации устройства или если Android решает уничтожить активность для экономии памяти.

На следующей странице более подробно рассказано о том, как эти методы сочетаются с состояниями активности.

Активность выполняется, когда она находится на переднем плане экрана.

Метод onCreate( ) вызывается при первом создании активности; в нем должна выполняться нормальная настройка активности.

Метод onDestroy( ) вызывается непосредственно перед уничтожением активности.

Жизненный цикл активности: от рождения до смерти

Перед вами краткий обзор жизненного цикла активности от рождения до смерти. Как будет показано позднее в этой главе, некоторые подробности опущены, потому что сейчас нас интересуют только методы `onCreate()` и `onDestroy()`.

1. **Активность запускается.**
Создается объект активности и выполняется конструктор.

2. **Метод onCreate() выполняется сразу же после запуска активности.**
В методе `onCreate()` следует размещать весь код инициализации, так как этот метод всегда вызывается после того, как активность была запущена, но до начала ее выполнения.

3. **Выполняемая активность видна на переднем плане, и пользователь может взаимодействовать с ней.**
Именно в этом состоянии активность проводит большую часть времени.

4. **Метод onDestroy() выполняется непосредственно перед уничтожением активности.**
Метод `onDestroy()` позволяет выполнить необходимые завершающие действия, например освободить ресурсы.

5. **После выполнения метода onDestroy() активность уничтожается.**
Активность перестает существовать.

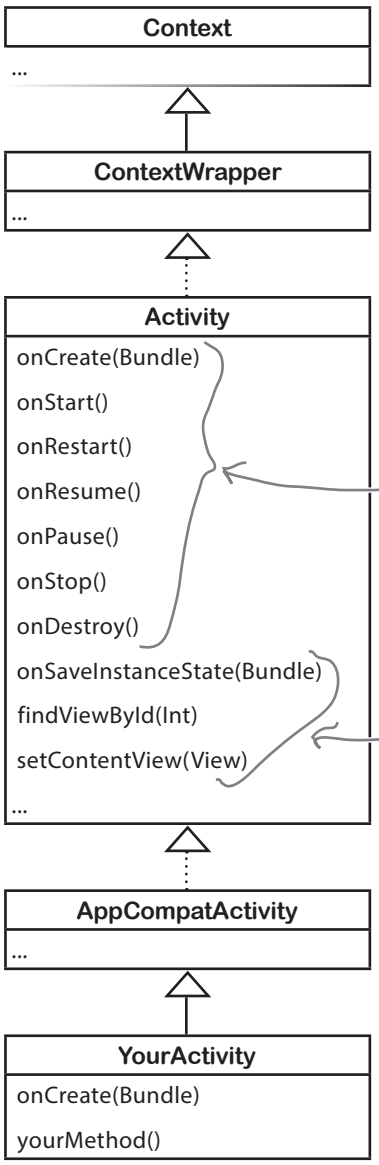
`onCreate()` и `onDestroy()` — два основных метода жизненного цикла активностей. Откуда же взялись эти методы?

Если на устройстве возникает критическая нехватка памяти, метод onDestroy() может не вызываться перед уничтожением активности.

Активность наследует методы жизненного цикла

Методы жизненного цикла активностей определяются в классе `android.app.Activity`. Каждая созданная вами активность является подтипом этого класса, поэтому она наследует эти методы.

На следующей диаграмме показаны основные представители иерархии классов вашей активности:



Абстрактный класс Context
(android.content.Context)
Предоставляет доступ к глобальной информации о среде выполнения приложения (например, ресурсам и операциям приложения).

Класс ContextWrapper
(android.content.ContextWrapper)
Опосредованная реализация для `Context`.

Класс Activity
(android.app.Activity)
Класс `Activity` реализует версии методов жизненного цикла по умолчанию. Также он определяет такие методы, как `findViewById()` `setContentView()`.

Методы жизненного цикла активностей. Вы больше узнаете о них в оставшейся части главы.

Эти методы не являются методами жизненного цикла, но они все равно очень полезны. Многие из них уже использовались в предшествующих главах.

Класс AppCompatActivity
(androidx.appcompat.app.AppCompatActivity)
Активность, которая является частью Android Jetpack. Позволяет использовать новую функциональность в старых версиях Android.

Класс YourActivity
(com.hfad.foo)
Большая часть поведения вашей активности реализуется унаследованными методами суперкласса. Вам остается лишь переопределить нужные методы.

Теперь вы знаете, как ваша активность получает свои методы жизненного цикла.
Давайте посмотрим, как обработать изменения конфигурации устройства.

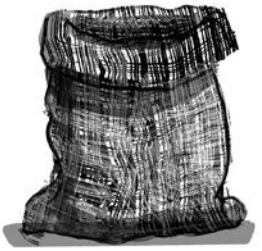
Сохранение текущего состояния в объекте Bundle

Как было показано ранее, в приложении Stopwatch проблемы возникли при повороте устройства. Активность `MainActivity` была уничтожена и создана заново, в результате чего все значения свойств были потеряны и представление `Chronometer` было сброшено.

Чтобы исправить эту ошибку, следует сохранить состояние `Chronometer` и значения свойств при уничтожении активности и восстановить их при ее повторном создании. Для сохранения значений будет использоваться объект **`Bundle`**.

`Bundle` — объект, предназначенный для хранения пар «ключ–значение». Перед уничтожением активности Android позволяет сохранить пары «ключ–значение» в `Bundle`. Затем этот объект `Bundle` используется новым экземпляром активности при повторном создании. Таким образом, использование `Bundle` дает возможность восстановить состояние активности при повороте экрана устройства.

Прежде чем смотреть, как `Bundle` передается между экземплярами активностей, посмотрим, как происходит добавление и загрузка значений.



Объект Bundle предназначен для хранения пар «ключ-значение». Эти объекты часто используются для восстановления значений свойств при уничтожении и повторном создании активностей.

Сохранение значений методами put

Пары «ключ-значение» сохраняются в `Bundle` методами `put`. Метод `putLong` сохраняет значение `Long` в `Bundle`, `putBoolean()` сохраняет `Boolean`, и т. д. Например, следующий код сохраняет в объекте `Bundle` с именем `bundle` ключ «answer» и значение 42 типа `Int`:

```
bundle.putInt("answer", 42)
```

В `Bundle` можно сохранить несколько пар «ключ–значение». Каждый ключ должен относиться к типу `String?`, а значение определяется вызываемым методом. Например, невозможно сохранить значение `Int` в `Bundle` вызовом метода `putString()` — компилятор запротестует.

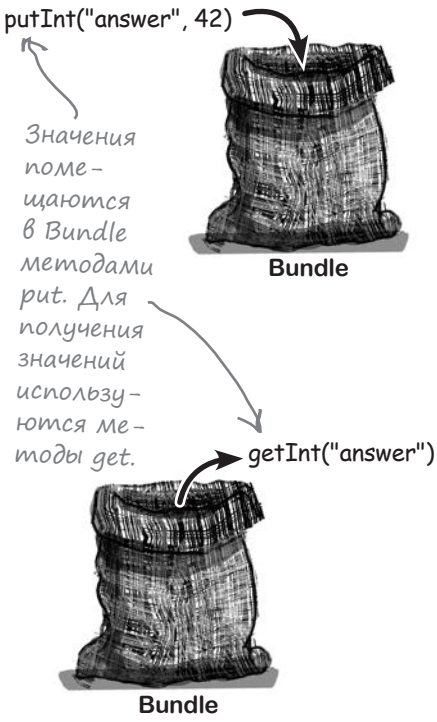
Чтение значений методами get

После того как пара «ключ–значение» будет сохранена в `Bundle`, значение можно прочитать одним из методов `get` объекта `Bundle`. Например, метод `getLong()` предназначен для чтения значений `Long`, а метод `getBoolean()` — для чтения значений `Boolean`.

Каждый метод `get` получает один параметр: имя ключа для значения, которое вы хотите получить. Например, следующий код получает значение, сохраненное с ключом «answer» в объекте `Bundle` с именем `bundle`:

```
val answerOfLife = bundle.getInt("answer")
```

Теперь вы знаете, как сохранять и читать значения из объектов `Bundle`, и мы можем воспользоваться этим объектом для сохранения и восстановления состояния активности.



Сохранение состояния вызовом onSaveInstanceState()

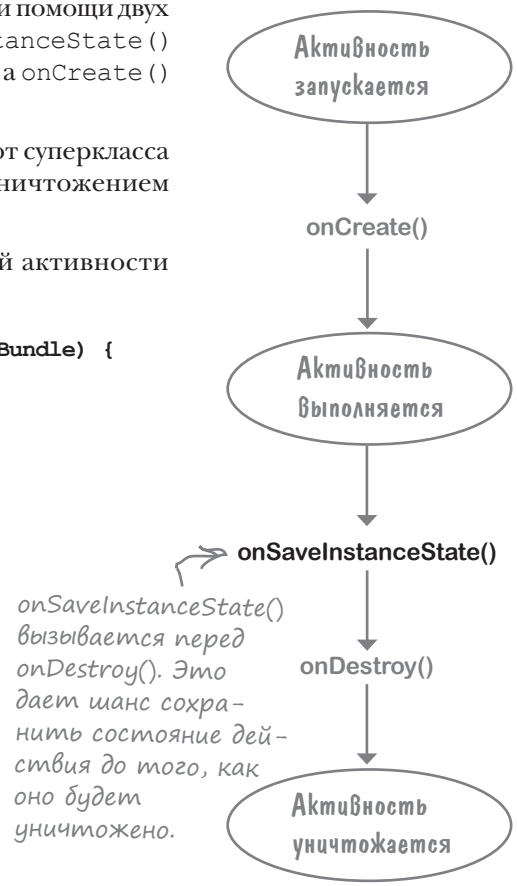
Объекты `Bundle` передаются между экземплярами активности при помощи двух методов: **`onSaveInstanceState()`** и **`onCreate()`**. `onSaveInstanceState()` сохраняет значения в `Bundle` перед уничтожением активности, а `onCreate()` снова загружает `Bundle` при повторном создании активности.

Каждая активность наследует метод `onSaveInstanceState()` от суперкласса `Activity`. Этот метод вызывается непосредственно перед уничтожением активности и получает один параметр: `Bundle`.

Метод `onSaveInstanceState()` переопределяется в вашей активности кодом следующего вида:

```
override fun onSaveInstanceState(savedInstanceState: Bundle) {
    savedInstanceState.putInt("answer", 42)
    super.onSaveInstanceState(savedInstanceState)
}
```

Приведенный пример сохраняет в `savedInstanceState` (объекте `Bundle`, переданном `onSaveInstanceState()`) ключ «answer» со значением 42 типа `Int`. Затем он вызывает версию `onSaveInstanceState()` из суперкласса, которая сохраняет состояние иерархии представлений.



Восстановление состояния в onCreate()

Как вы уже знаете, метод `onCreate()` вызывается при повторном создании активности, и он получает один параметр `Bundle?`. Если активность создается с нуля, то значение `Bundle?` будет равно `null`, но при повторном создании активности это будет объект `Bundle`, который использовался `onSaveInstanceState()`.

Метод `onCreate()` используется для чтения значений из `Bundle` кодом следующего вида:

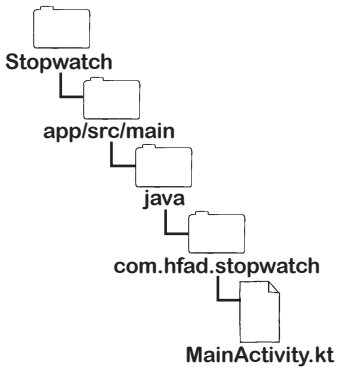
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    if (savedInstanceState != null) {
        var answer = savedInstanceState.getInt("answer")
    }
    ...
}
```

Значение answer должно восстанавливаться только при уничтожении и повторном создании активности, а значит, объект Bundle, переданный onCreate(), отличен от null.

Какие же изменения необходимо внести в код приложения Stopwatch?

Обновленный код MainActivity.kt

Мы обновили код `MainActivity`, чтобы при повороте устройства состояние сохранялось в `Bundle` методом `onSaveInstanceState()` и восстанавливалось в `onCreate()`. Обновите файл *MainActivity.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer

class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //Хронометр
    var running = false  // Хронометр работает?
    var offset: Long = 0  //Базовое смещение

    //Добавление строк для ключей, используемых с Bundle
    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        //Получение ссылки на секундомер
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)

        //Восстановление предыдущего состояния
        if (savedInstanceState != null) {
            offset = savedInstanceState.getLong(OFFSET_KEY)
            running = savedInstanceState.getBoolean(RUNNING_KEY)
            if (running) {
                stopwatch.base = savedInstanceState.getLong(BASE_KEY)
                stopwatch.start()
            } else setBaseTime()
        }
```

Эти три константы будут использоваться как имена значений, добавляемых в Bundle.

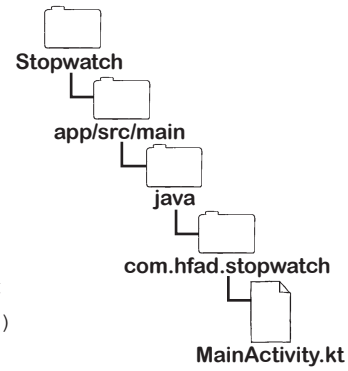
Восстановить состояние offset и running.

Установить время на секундомере и запустить его, если он должен выполняться.

Продолжение на следующей странице.

Обновленный код MainActivity.kt (продолжение)

```
        //Кнопка start запускает секундомер, если он не работал
        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                stopwatch.start()
                running = true
            }
        }

        //Кнопка pause останавливает секундомер, если он работал
        val pauseButton = findViewById<Button>(R.id.pause_button)
        pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                stopwatch.stop()
                running = false
            }
        }

        //Кнопка reset обнуляет offset и базовое время
        val resetButton = findViewById<Button>(R.id.reset_button)
        resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putLong(OFFSET_KEY, offset)
        savedInstanceState.putBoolean(RUNNING_KEY, running)
        savedInstanceState.putLong(BASE_KEY, stopwatch.base)
        super.onSaveInstanceState(savedInstanceState)
    }

    ...
}
```



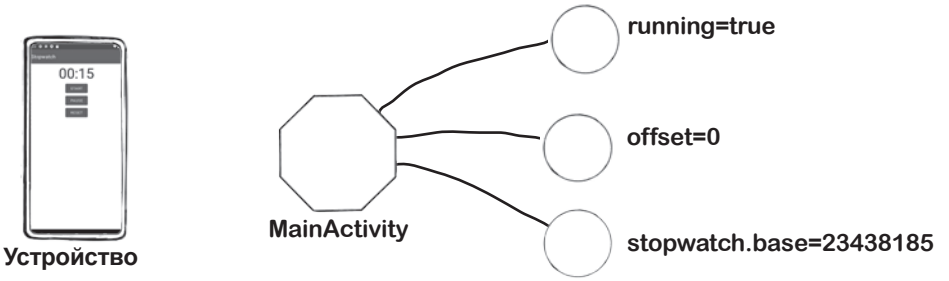
Используйте метод onSaveInstanceState для сохранения свойств offset, running и stopwatch.base.

Код методов saveOffset() и setBaseTime() опущен, так как их обновлять не нужно.

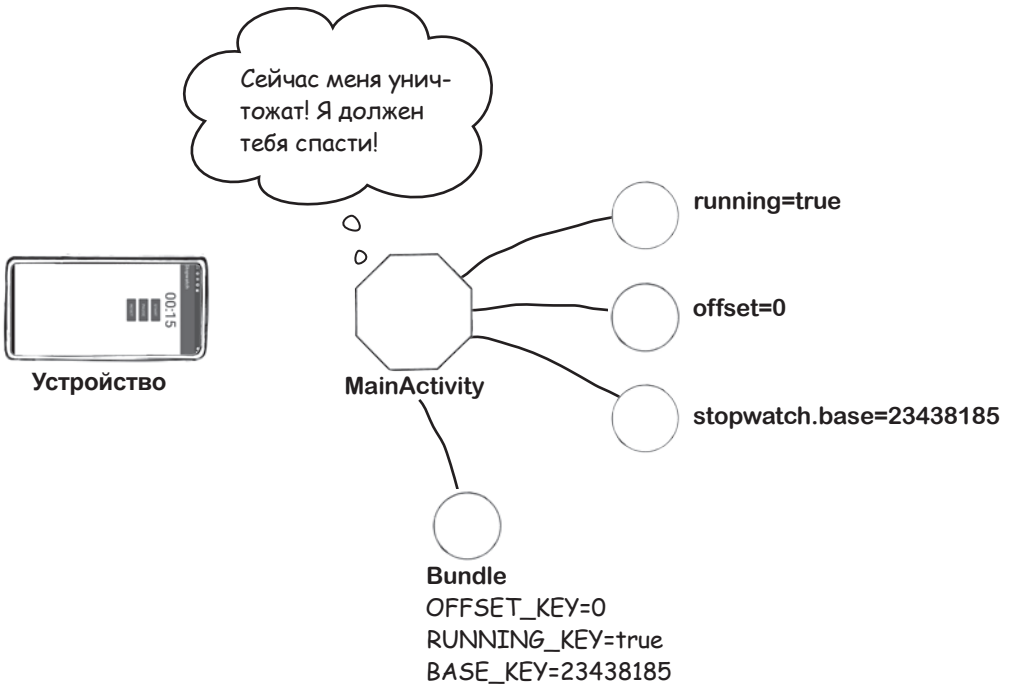
Давайте разберемся в том, как работает код, и опробуем приложение на практике.

Что происходит при запуске приложения

1. **Пользователь запускает приложение и щелкает на кнопке Start.**
Секундомер запускается, и свойству `running` присваивается `true`.

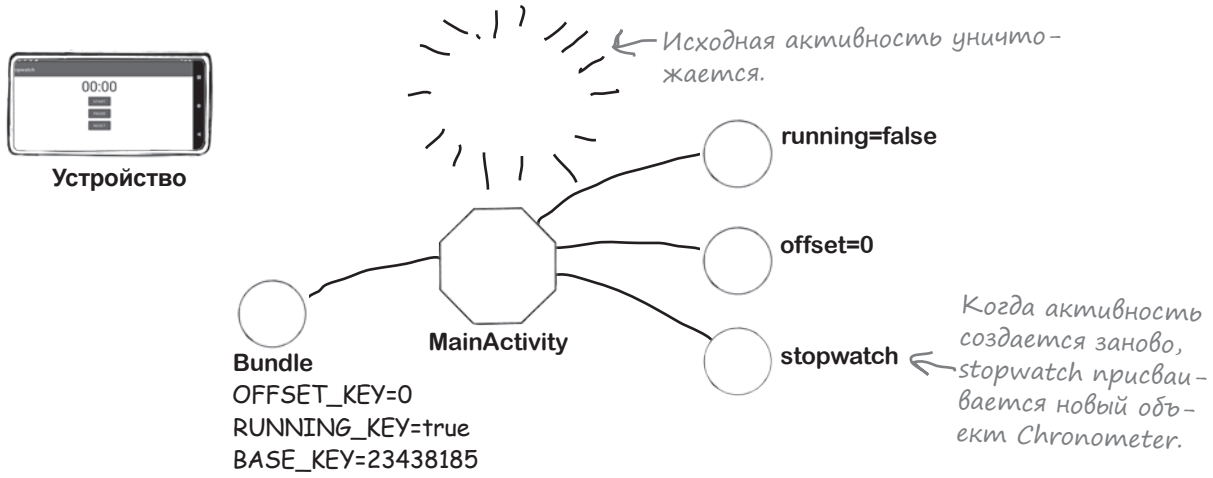


2. **Пользователь поворачивает экран устройства.**
Android рассматривает происходящее как изменение конфигурации и готовится к уничтожению активности. Перед уничтожением активности вызывается метод `onSaveInstanceState()`. Метод `onSaveInstanceState()` сохраняет значения `offset`, `running` и `stopwatch.base` в объекте `Bundle`.

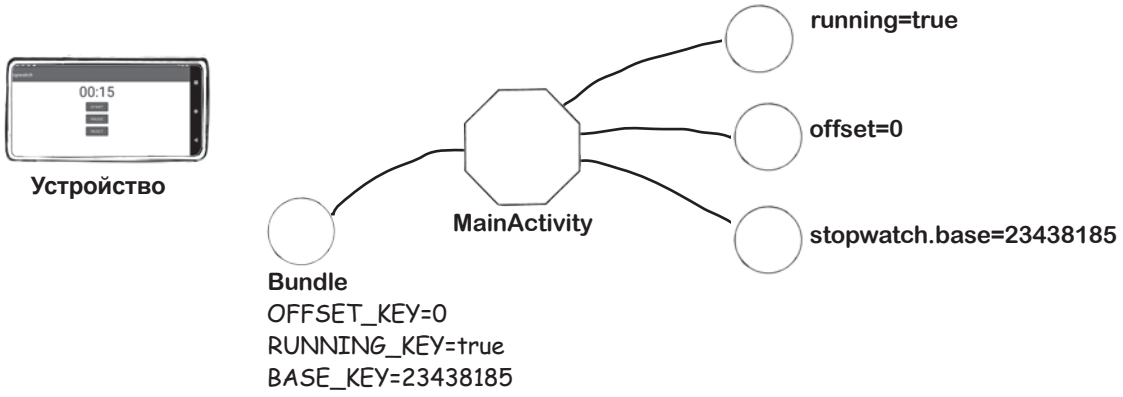


История продолжается

3 **Android уничтожает активность, а затем создает ее заново.**
Значения свойств активности сбрасываются.



4 **Вызывается метод onCreate().**
Значения, сохраненные в `Bundle`, используются для инициализации свойств.



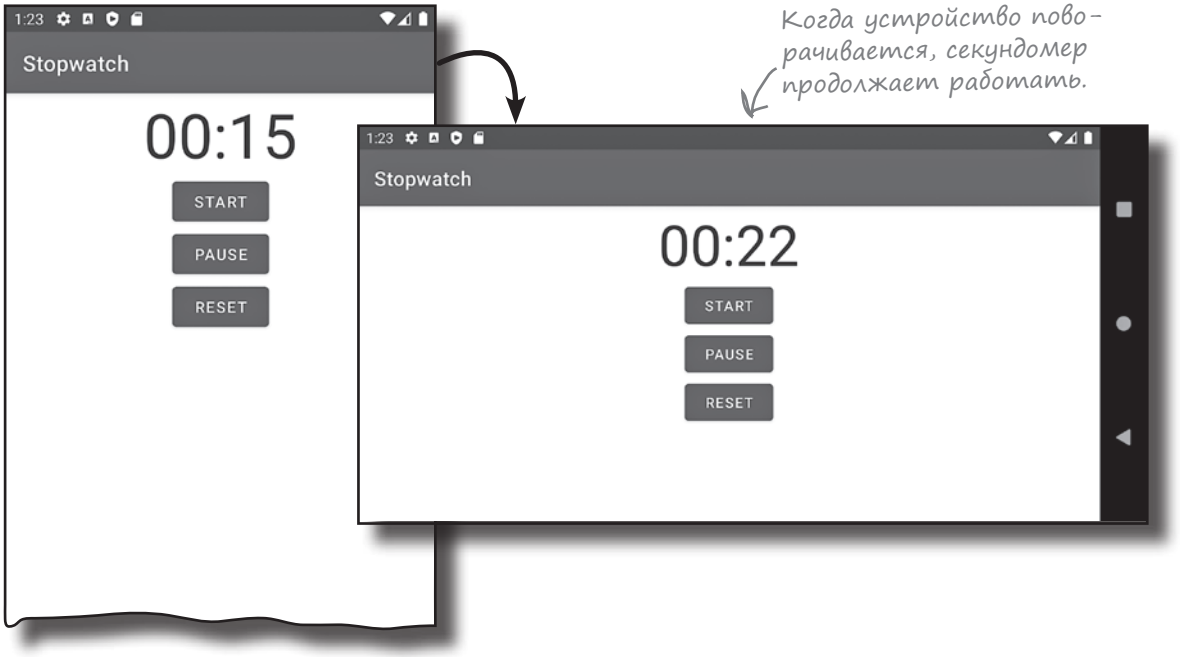
5 **Секундомер снова запускается и выводит правильное время.**

Давайте проверим, как работает приложение.

Тест-драйв

Внесите изменения в код активности и запустите приложение.

Кнопка Start запускает секундомер, и он продолжает работать при повороте экрана устройства.



Часто Задаваемые Вопросы

В: Напомните, почему Android создает заново активность при простом повороте экрана?

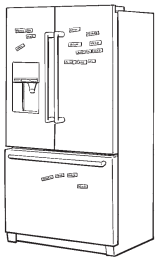
О: Метод `onCreate()` обычно используется для настройки экрана. Если код `onCreate()` зависит от конфигурации экрана (например, при использовании разных макетов для горизонтальной и вертикальной ориентации), метод `onCreate()` должен вызываться при каждом изменении конфигурации. Если пользователь сменит локальный контекст, пользовательский интерфейс необходимо создать заново на выбранном языке.

В: Почему Android не сохраняет все переменные экземпляра автоматически? Почему мне нужно писать весь этот код самостоятельно?

О: Далеко не всегда нужно сохранять все переменные экземпляра. Например, в вашей программе может использоваться переменная для хранения текущей ширины экрана. Значение такой переменной должно быть вычислено заново при следующем вызове `onCreate()`.

В: Вы упоминали о том, что `AppCompatActivity` является частью Android Jetpack. Это отдельная библиотека? Нужно ли обновлять файлы *build.gradle*?

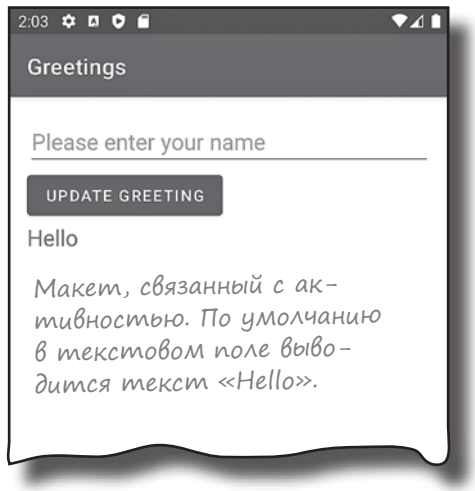
О: `AppCompatActivity` является частью библиотеки `AppCompat`. Мы не предлагали вам обновлять файлы *build.gradle*, потому что среда Android Studio уже включила эту библиотеку за вас. Если захотите удостовериться в этом, найдите зависимость `AppCompat` в файле *build.gradle*.



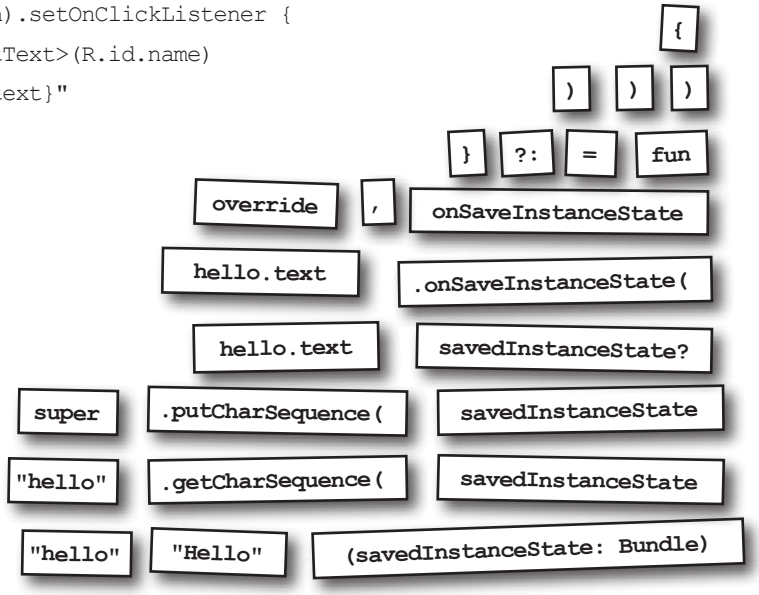
Развлечения с магнитами

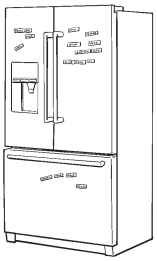
Приведенная ниже активность предназначена для нового приложения Greetings. Пользователь вводит имя в текстовом поле (с идентификатором `name`), а затем щелкает на кнопке (с идентификатором `button`), после чего текст в текстовом поле (с идентификатором `hello`) обновляется и в него включается введенное имя.

Если вы запустите приложение и повернете экран, текст в текстовом поле `hello` теряется. Разложите магниты в приведенном ниже коде, чтобы текст восстанавливался при повороте экрана.




```
...
class MainActivity : AppCompatActivity() {

    lateinit var hello: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        hello = findViewById<TextView>(R.id.hello)

Здесь должен размещаться некоторый код.

        findViewById<Button>(R.id.button).setOnClickListener {
            val name = findViewById<EditText>(R.id.name)
            hello.text = "Hello ${name.text}"
        }
    }

И здесь еще один фрагмент кода.

}
```

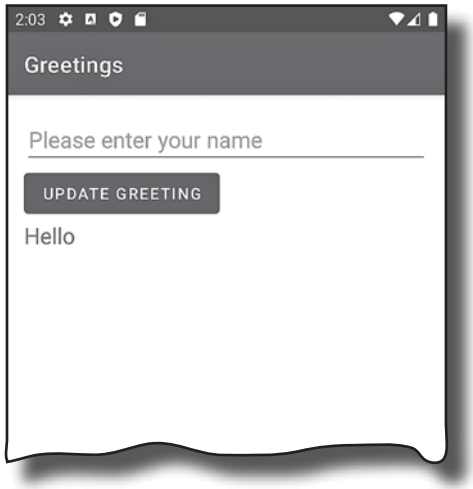




Развлечения с магнитами. Решение

Приведенная ниже активность предназначена для нового приложения Greetings. Пользователь вводит имя в текстовом поле (с идентификатором `name`), а затем щелкает на кнопке (с идентификатором `button`), после чего текст в текстовом поле (с идентификатором `hello`) обновляется и в него включается введенное имя.

Если вы запустите приложение и повернете экран, текст в текстовом поле `hello` теряется. Разложите магниты в приведенном ниже коде, чтобы текст восстанавливался при повороте экрана.




```
...
class MainActivity : AppCompatActivity() {

    lateinit var hello: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        hello = findViewById<TextView>(R.id.hello)
        hello.text = savedInstanceState?.getCharSequence("hello") ?: "Hello"
        findViewById<Button>(R.id.button).setOnClickListener {
            val name = findViewById<EditText>(R.id.name)
            hello.text = "Hello ${name.text}"
        }
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        super.onSaveInstanceState(savedInstanceState.putCharSequence("hello", hello.text))
    }
}
```

Восстанавливает текст hello, если значение savedInstanceState отлично от null.

Переопределяет метод onSaveInstanceState.

Вызывает метод суперкласса.

Помещает значение hello.text в объект Bundle savedInstanceState.

Жизнь активности не ограничивается созданием и уничтожением

До настоящего момента мы рассмотрели фазы создания и уничтожения в жизненном цикле (и немного того, что происходит между ними); также вы узнали, как обрабатываются изменения конфигурации, например изменение ориентации экрана. Однако в жизни приложения существуют и другие события, обработка которых поможет вам добиться того, чтобы приложение работало так, как вам нужно.

Например, предположим, что во время работы секундомера поступил телефонный звонок. Хотя секундомер и не виден на экране, он будет продолжать работу. Но что, если вы хотите, чтобы секундомер на это время останавливался и запускался снова, когда приложение опять становится видимым?

Даже если вы не хотите, чтобы ваш секундомер вел себя подобным образом, это отличный повод рассмотреть другие методы жизненного цикла.

Старт, остановка и перезапуск

К счастью, использование правильных методов жизненного цикла позволяет легко обрабатывать действия, связанные с видимостью приложения. Кроме методов onCreate() и onDestroy(), связанных с началом и завершением всего жизненного цикла активности, также существуют другие методы жизненного цикла, связанные с видимостью активности.

Есть три ключевых метода жизненного цикла, связанных с переходами активности в видимое или невидимое состояние: `onStart()`, `onStop()` и `onRestart()`. Как и методы `onCreate()` и `onDestroy()`, они наследуются от класса Android `Activity`.

Ниже приведена краткая сводка этих методов:

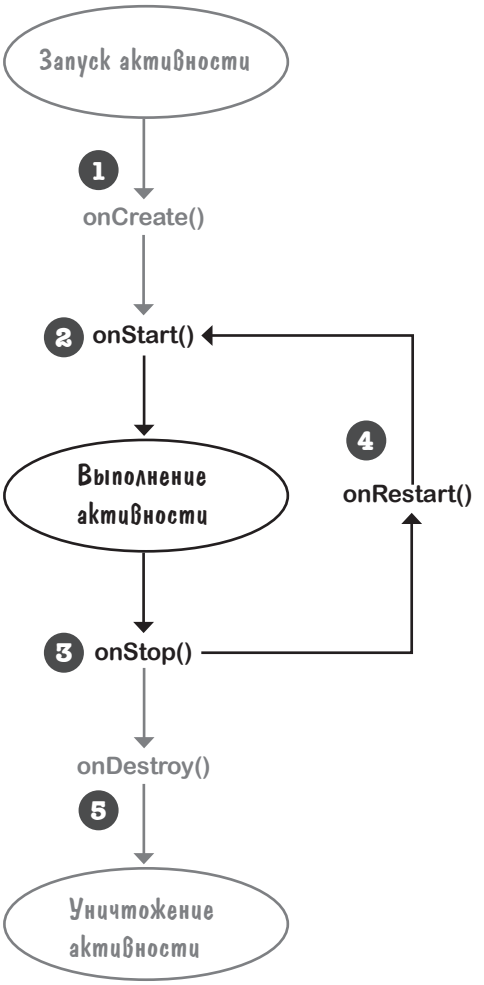
Активность находится в состоянии остановки, если она полностью закрыта другой активностью и невидима для пользователя. При этом активность продолжает существовать на заднем плане и сохраняет всю информацию состояния.

Метод	Когда вызывается
onStart()	Когда активность становится видимой для пользователя.
onStop()	Когда активность перестает быть видимой. Это может произойти из-за того, что она полностью закрывается другой активностью, отображаемой поверх нее, или из-за того, что активность готовится к уничтожению.
onRestart()	Когда активность, ставшая невидимой, снова должна появиться на экране.

На следующей странице разберемся, какое отношение эти методы имеют к методам `onCreate()` и `onDestroy()`.

Жизненный цикл активности: видимость

Ниже диаграмма жизненного цикла активности, приводившаяся ранее в этой главе, дополняется методами `onStart()`, `onStop()` и `onRestart()` (аспекты, которые сейчас представляют для нас интерес, выделены жирным шрифтом):



1. **Активность запускается, выполняется ее метод onCreate().**
   Выполняется код инициализации активности из метода `onCreate()`. В этой точке активность не видна, так как метод `onStart()` еще не вызывался.

2. **Метод onStart() выполняется после метода onCreate(). Он вызывается, когда активность собирается стать видимой.**
   После выполнения метода `onStart()` пользователь видит активность на экране.

3. **Метод onStop() выполняется, когда активность перестает быть видимой пользователю.**
   После выполнения метода `onStop()` активность не видна на экране.

4. **Если активность снова становится видимой пользователю, вызывается метод onRestart(), за которым следует вызов onStart().**
   Активность может проходить через этот цикл многократно, если она несколько раз скрывается и снова становится видимой.

5. **Наконец, активность уничтожается.**
   Метод `onStop()` вызывается перед `onDestroy()`.

Необходимо реализовать еще два метода жизненного цикла

Чтобы обновить приложение Stopwatch, необходимо сделать две вещи. Во-первых, необходимо реализовать метод `onStop()` активности, чтобы отсчет времени останавливался, если приложение стало невидимым. Когда это будет сделано, необходимо реализовать метод `onRestart()`, чтобы отсчет времени возобновлялся, когда приложение становится видимым.

Начнем с метода `onStop()`.

В переопределениях методов жизненного цикла активности должен вызываться метод суперкласса. Если этого не сделать, компилятор выдаст сообщение об ошибке.

Реализация остановки секундомера в onStop()

Переопределите метод `onStop()`, добавив в активность следующий метод:

```
override fun onStop() {
    super.onStop()
    //Код, выполняемый при остановке
активности
}
```

В строке:

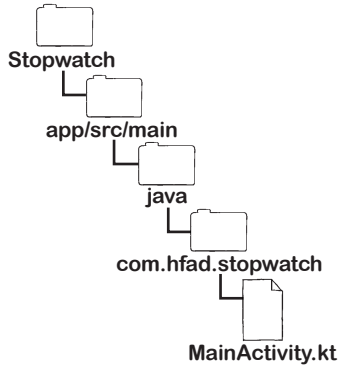
```
super.onStop()
```

вызывается метод `onStop()` суперкласса `Activity`. Эта строка кода должна добавляться во всех переопределениях метода `onStop()`, чтобы активность выполнила все действия метода жизненного цикла суперкласса. При попытке обойти этот шаг Android сгенерирует исключение. Сказанное относится ко всем методам жизненного цикла; если вы переопределяете любые методы жизненного цикла `Activity` в своей активности, необходимо вызвать метод суперкласса, иначе компилятор начнет протестовать.

Отсчет времени должен останавливаться при вызове метода `onStop()`. Для этого необходимо вызвать метод `saveOffset()` (чтобы позднее отсчет времени можно было продолжить с момента остановки), а затем остановить секундомер. Код выглядит так:

```
override fun onStop() {
    super.onStop()
    if (running) {
        saveOffset()
        stopwatch.stop()
    }
}
```

Через пару страниц этот код будет добавлен в файл MainActivity.kt.



Итак, теперь секундомер останавливается, когда активность становится невидимой. Теперь нужно сделать следующий шаг — снова запустить отсчет времени, когда активность станет видимой.

Перезапуск отсчета времени, когда приложение становится видимым

Когда активность снова становится видимой, вызываются два ключевых метода жизненного цикла: `onStart()` и `onRestart()`.

Метод `OnStart()` вызывается, когда активность должна стать видимой. Это может произойти как при ее создании, так и тогда, когда она снова становится видимой после потери видимости.

Метод onStart() вызывается каждый раз, когда активность становится видимой.

Метод `onStart()` реализуется кодом следующего вида:

```
override fun onStart() {
    super.onStart()
    //Код, выполняемый при запуске активности
}
```

Как и в случае с методом `onStop()`, строка:

```
super.onStart()
```

вызывает метод `onStart()` суперкласса `Activity`.

Метод `onRestart()` вызывается перед `onStart()`, но только когда активность становится видимой после потери видимости. Он *не* вызывается, когда активность становится видимой впервые.

Метод onRestart() вызывается тогда, когда активность, ставшая невидимой, снова появляется на экране.

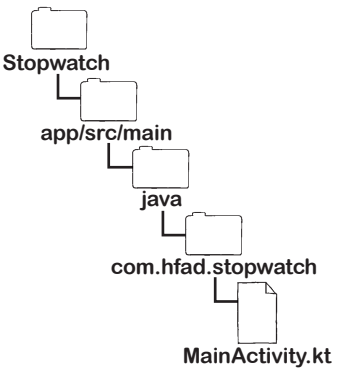
Метод `onRestart()` реализуется кодом следующего вида:

```
override fun onRestart() {
    super.onRestart()
    //Код, выполняемый при перезапуске активности
}
```

В приложении Stopwatch отсчет времени должен возобновляться тогда, когда активность снова становится видимой, так что мы реализуем метод `onRestart()`. Код выглядит так:

Мы добавим этот метод в файл MainActivity.kt на следующей странице.

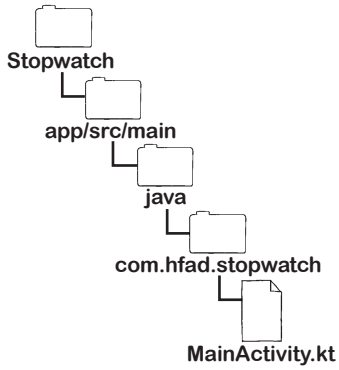
```
override fun onRestart() {
    super.onRestart()
    if (running) {
        setBaseTime()
        stopwatch.start()
        offset = 0
    }
}
```



Обновленный код MainActivity.kt

Ниже приведен код файла *MainActivity.kt* с методами `onStop()` и `onRestart()`; внесите изменения, выделенные жирным шрифтом:

```
...
class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //Хронометр
    var running = false  //Хронометр работает?
    var offset: Long = 0  //Базовое смещение

    //Добавление строк для ключей, используемых с Bundle
    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onStop() {
        super.onStop()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }

    override fun onRestart() {
        super.onRestart()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    ...
}
```



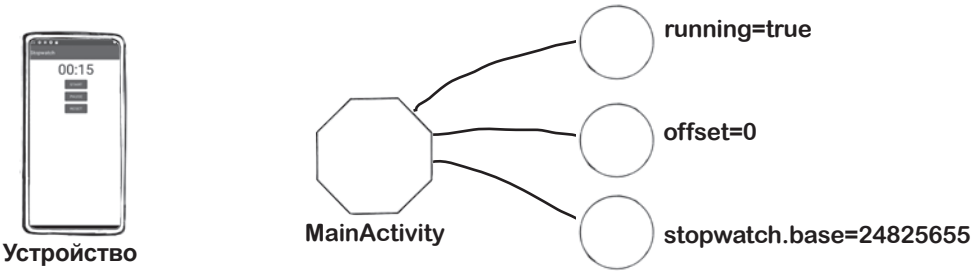
Изменять метод onCreate() не нужно, поэтому мы опустили код.

Не забудьте добавить методы onStop() и onRestart() в свой код.

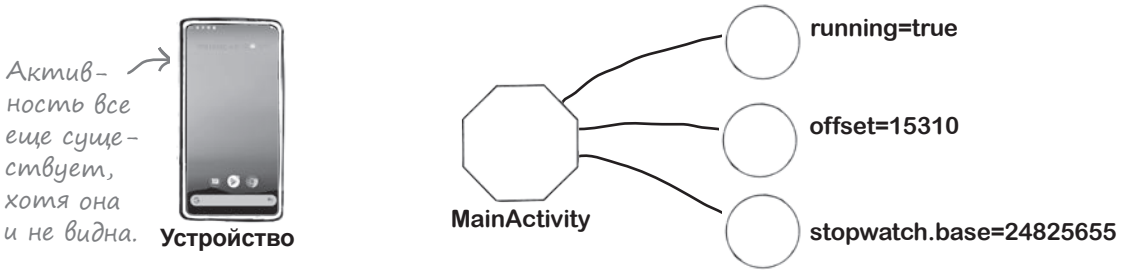
Код методов saveOffset(), setBaseTime() и onSaveInstanceState() не приводится, так как обновлять их не нужно.

Что происходит при запуске приложения

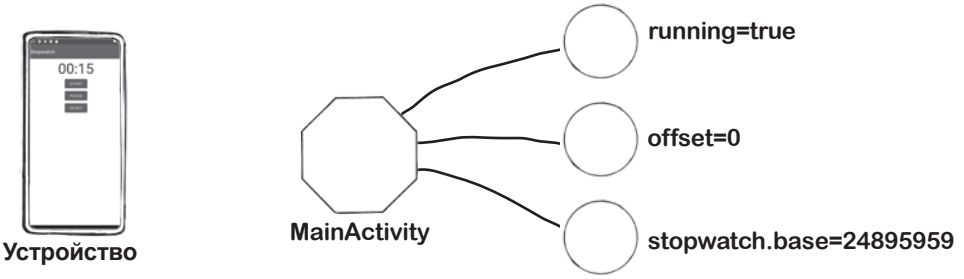
1. **Пользователь запускает приложение и щелкает на кнопке Start, чтобы запустить отсчет времени.**
Отсчет времени начинается, свойству `running` присваивается значение `true`.



2. **Пользователь переходит к домашнему экрану устройства, так что приложение Stopwatch перестает быть видимым.**
Вызывается метод `onStop()`, свойство `offset` обновляется, а отсчет времени останавливается.



3. **Пользователь снова переходит к приложению Stopwatch.**
Вызывается метод `onRestart()`, свойство `stopwatch.base` обновляется, отсчет времени возобновляется, а свойству `offset` присваивается `0`. Отсчет времени продолжается с правильного момента.



Тест-драйв

Запустите приложение. Щелчок на кнопке Start запускает секундомер. Отсчет времени останавливается, когда приложение становится невидимым, и возобновляется, когда приложение снова появляется на экране.

Часто Задаваемые Вопросы

В: А нельзя было воспользоваться методом `onStart()` вместо `onRestart()`, чтобы возобновить отсчет времени?

О: Можно. Когда активность становится видимой после того, как она какое-то время оставалась невидимой, вызываются методы `onRestart()` и `onStart()`, так что мы могли воспользоваться любым методом.

В: Тогда почему мы использовали `onRestart()`, а не `onStart()`?

О: В отличие от `onRestart()`, `onStart()` также вызывается при первом появлении активности после ее создания. Это означает, что решение с `onRestart()` чуть эффективнее решения с `onStart()`.

Упражнение

Код активности справа изменяет значение свойства с именем `x` на различных стадиях жизненного цикла активности. Удастся ли вам определить, какие методы жизненного цикла будут вызываться и итоговое значение `x` после каждого из этих действий?

1. Пользователь открывает приложение, и оно начинает работать.

2. Пользователь открывает приложение, и оно начинает работать. Пользователь поворачивает устройство.

3. Пользователь открывает приложение, и оно начинает работать. Пользователь переходит на домашний экран Android, а затем возвращается к приложению.

```
package com.hfad.lifecycleexercise

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    var x = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        x = savedInstanceState?.getInt("x") ?: 2
    }

    override fun onRestart() {
        super.onRestart()
        x -= 7
    }

    override fun onStart() {
        super.onStart()
        x += 6
    }

    override fun onStop() {
        super.onStop()
        x *= 2
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putInt("x", x + 1)
        super.onSaveInstanceState(savedInstanceState)
    }
}
```

Упражнение Решение

Код активности справа изменяет значение свойства с именем `x` на различных стадиях жизненного цикла активности. Удастся ли вам определить, какие методы жизненного цикла будут вызываться и итоговое значение `x` после каждого из этих действий?

1. Пользователь открывает приложение, и оно начинает работать.

```
onCreate(): x = 2
onStart():. x = 8
The final value of x is 8.
```

2. Пользователь открывает приложение, и оно начинает работать. Пользователь поворачивает устройство.

```
onCreate(): x = 2
onStart(): x = 8
onStop(): x = 16
onSaveInstanceState(): x = 16
onCreate(): x = 17
onStart(): x = 23
The final value of x is 23.
```

Хотя onSaveInstanceState() сохраняет значение 17 в Bundle, значение x не задается до выполнения метода onCreate().

Здесь метод onRestart() не вызывается, потому что приложение не теряет видимости.

3. Пользователь открывает приложение, и оно начинает работать. Пользователь переходит на домашний экран Android, а затем возвращается к приложению.

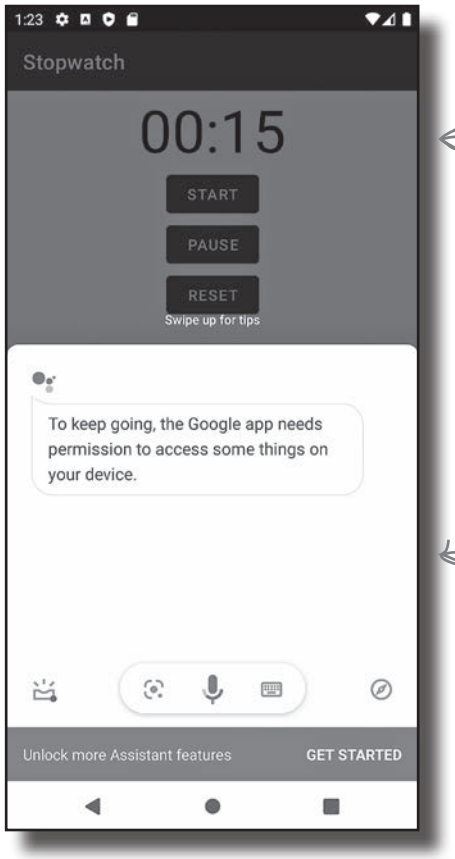
```
onCreate(): x = 2
 onStart(): x = 8
 onStop(): x = 16
onRestart(): x = 9
onStart(): x = 15
The final value of x is 15.
```

```
package com.hfad.lifecycleexercise

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    var x = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        x = savedInstanceState?.getInt("x") ?: 2
    }

    override fun onRestart() {
        super.onRestart()
        x -= 7
    }

    override fun onStart() {
        super.onStart()
        x += 6
    }

    override fun onStop() {
        super.onStop()
        x *= 2
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putInt("x", x + 1)
        super.onSaveInstanceState(savedInstanceState)
    }
}
```

А если активность видна только частично?

Ранее вы видели, что происходит при создании и уничтожении активностей и что происходит, когда активность становится видимой или невидимой. Но необходимо рассмотреть еще одну ситуацию: **когда активность видима, но не обладает фокусом**.

Если активность видима, но не обладает фокусом, она **приостанавливается**. Это может произойти, если над вашей активностью отображается другая активность, которая не занимает весь экран или прозрачна. Верхняя активность обладает фокусом, но нижняя активность остается видимой и поэтому находится в приостановленном состоянии.



Активность секундомера остается видимой, но она частично скрыта и не имеет фокуса. В таком случае она находится в приостановленном состоянии.

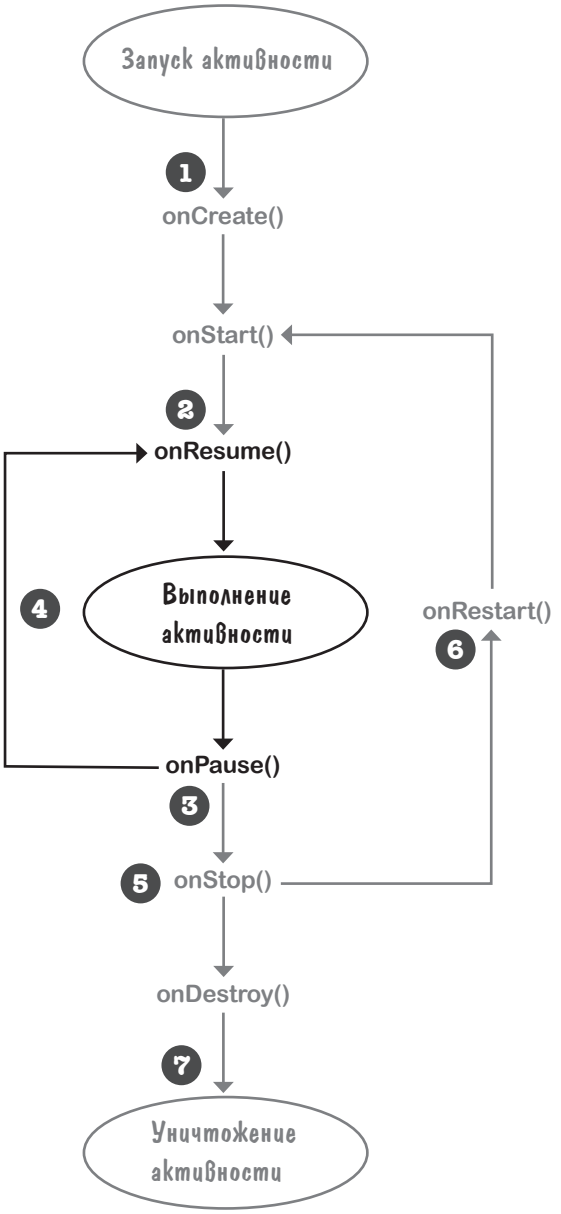
Это активность приложения Google Assistant, которое отображается поверх секундомера.

Активность находится в приостановленном состоянии, если она потеряла фокус, но остается видимой пользователю. При этом активность продолжает существовать и поддерживает всю свою информацию состояния.

Существуют два метода жизненного цикла, которые обрабатывают приостановку активности и ее повторную активизацию: **`onPause()`** и **`onResume()`**. Метод `onPause()` вызывается, если активность остается видимой, но фокус принадлежит другой активности. Метод `onResume()` вызывается непосредственно перед тем, как ваша активность будет готова к взаимодействию с пользователем. Посмотрим, как эти методы вписываются в общую картину жизненного цикла активности.

Жизненный цикл переднего плана

Дополним диаграмму жизненного цикла, которая приводилась ранее в этой главе. На этот раз в нее добавляются методы `onResume()` и `onPause()` (новые фрагменты выделены черным):



1. **Активность запускается, вызываются ее методы onCreate() и onStart().**
   В этой точке активность видна, но не обладает фокусом.

2. **Выполняется метод onResume(). Он вызывается в тот момент, когда активность собирается перейти на передний план.**
   После выполнения метода `onResume()` активность обладает фокусом, и пользователь может взаимодействовать с ней.

3. **Метод onPause() выполняется, когда активность перестает находиться на переднем плане.**
   После выполнения метода `onPause()` активность все еще видима, не не обладает фокусом.

4. **Если активность снова переходит на передний план, вызывается метод onResume().**
   Активность может пройти этот цикл многократно, если она многократно теряет фокус, а затем снова получает его.

5. **Если активность перестает быть видимой для пользователя, вызывается метод onStop().**
   После выполнения метода `onStop()` активность невидима на экране.

6. **Если активность снова становится видимой пользователю, вызывается метод onRestart(), за которым следуют вызовы onStart() и onResume().**
   Активность может пройти через этот цикл многократно.

7. **Наконец, активность уничтожается.**
   При переходе активности из состояния выполнения в состояние уничтожения перед уничтожением активности вызываются методы `onPause()` и `onStop()`.

Приостановка отсчета времени при приостановке активности

Вернемся к приложению Stopwatch.

К настоящему моменту мы заставили отсчет времени приостанавливаться, если приложение Stopwatch невидимо, и перезапускаться, когда приложение снова становится видимым. Для этого были переопределены методы `onStop()` и `onRestart()`.

Сделаем так, чтобы приложение точно так же вело себя при частичной видимости. Отсчет времени должен приостанавливаться при приостановке активности и снова запускаться при возобновлении ее работы. Какие же методы жизненного цикла для этого нужно реализовать?

В двух словах, следует использовать методы `onPause()` и `onResume()`, но можно пойти еще дальше. Мы используем эти методы для *замены* уже реализованных вызовов `onStop()` и `onRestart()`.

Если снова взглянуть на диаграмму жизненного цикла, методы `onPause()` и `onResume()`, помимо методов `onStop()`, `onStart()` и `onRestart()`, вызываются при каждой остановке и перезапуске активности. Так как мы хотим, чтобы приложение вело себя одинаково независимо от того, приостанавливается активность или останавливается, можно воспользоваться методами `onPause()` и `onResume()` в обеих ситуациях.

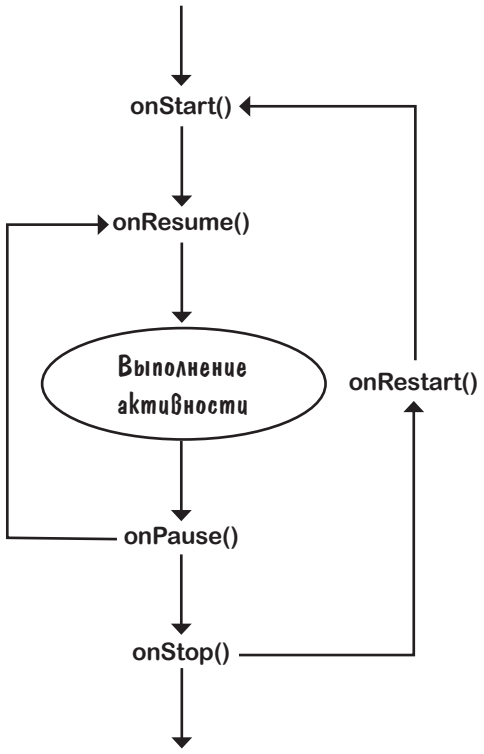
Реализация методов `onPause()` и `onResume()` выглядит примерно так:

```
override fun onPause() {
    super.onPause()
    //Код, выполняемый при приостановке активности
}

override fun onResume() {
    super.onResume()
    //Код, выполняемый при возобновлении активности
}
```

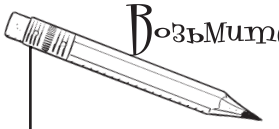
Как и прежде, вызовы `super.onPause()` и `super.onResume()` обязательны. Они вызывают методы `onPause()` и `onResume()` суперкласса `Activity`, и без этих вызовов код не будет компилироваться.

Мы хотим заменить вызовы методов `onStop()` и `onRestart()` в *MainActivity.kt* реализациями `onPause()` и `onResume()`. Проверьте, удастся ли вам реализовать необходимые изменения в коде, в следующем упражнении.

Метод onResume() вызывается при перезапуске или возобновлении активности. Так как приложение должно делать одно и то же независимо от того, запускается оно или возобновляется, можно реализовать метод onResume() вместо onRestart().



Метод onPause() вызывается при приостановке или остановке активности. Это означает, что мы можем реализовать метод onPause() вместо onStop().



Возьмите в руку карандаш

Отсчет времени должен останавливаться каждый раз, когда `MainActivity` теряет фокус, и продолжаться при его повторном получении. Обновите приведенный ниже код *MainActivity.kt* и включите в него необходимые изменения.

Здесь приводится не полный код MainActivity.kt — только те части, которые нужно будет изменить.

```
...
class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //Хронометр
    var running = false  //Хронометр работает?
    var offset: Long = 0  //Базовое смещение
    ...

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onStop() {
        super.onStop()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }

    override fun onRestart() {
        super.onRestart()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    ...
}
```

Возьмите в руку карандаш

Решение

Отсчет времени должен останавливаться каждый раз, когда `MainActivity` теряет фокус, и продолжаться при его повторном получении. Обновите приведенный ниже код *MainActivity.kt* и включите в него необходимые изменения.

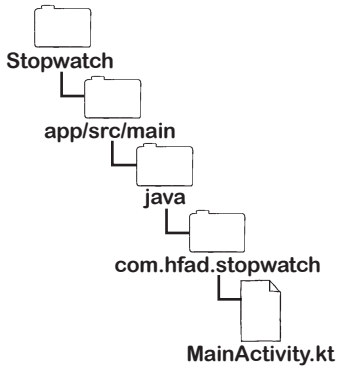
```
...
class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //Хронометр
    var running = false  //Хронометр работает?
    var offset: Long = 0  //Базовое смещение
    ...

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun ~~onStop()~~ onPause() {
        ~~super.onStop()~~
        super.onPause()
        if (running) {
            saveOffset()
            stopwatch.stop()
        }
    }

    override fun ~~onRestart()~~ onResume() {
        ~~super.onRestart()~~
        super.onResume()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    ...
}
```

Необходимо заменить onStop() на onPause() и вызвать версию метода onPause() из суперкласса.

Аналогичным образом необходимо заменить onRestart() на onResume().

Полный код MainActivity.kt

Ниже приведен полный код *MainActivity.kt* для завершенного приложения; обновите свою версию кода (изменения выделены жирным шрифтом):



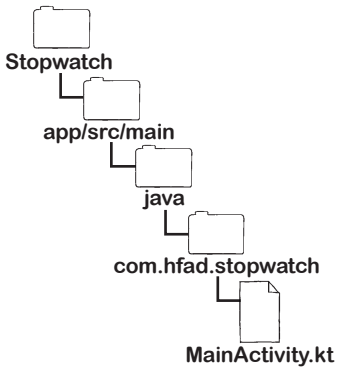

```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
import android.widget.Button
import android.widget.Chronometer

class MainActivity : AppCompatActivity() {

    lateinit var stopwatch: Chronometer  //Хронометр
    var running = false  //Хронометр выполняется?
    var offset: Long = 0  //Базовое смещение

    //Добавление строк для ключей, используемых с Bundle
    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        //Получение ссылки на секундомер
        stopwatch = findViewById<Chronometer>(R.id.stopwatch)

        //Восстановление предыдущего состояния
        if (savedInstanceState != null) {
            offset = savedInstanceState.getLong(OFFSET_KEY)
            running = savedInstanceState.getBoolean(RUNNING_KEY)
            if (running) {
                stopwatch.base = savedInstanceState.getLong(BASE_KEY)
                stopwatch.start()
            } else setBaseTime()
        }
```

Код на этой странице изменять не нужно.

Продолжение на следующей странице.

Полный код MainActivity.kt (продолжение)

```
    //Кнопка start запускает секундомер, если он не работал
    val startButton = findViewById<Button>(R.id.start_button)
    startButton.setOnClickListener {
        if (!running) {
            setBaseTime()
            stopwatch.start()
            running = true
        }
    }

    //Кнопка pause останавливает секундомер, если он работал
    val pauseButton = findViewById<Button>(R.id.pause_button)
    pauseButton.setOnClickListener {
        if (running) {
            saveOffset()
            stopwatch.stop()
            running = false
        }
    }

    //Кнопка reset обнуляет offset и базовое время
    val resetButton = findViewById<Button>(R.id.reset_button)
    resetButton.setOnClickListener {
        offset = 0
        setBaseTime()
    }
}

override fun ~~onStop()~~ onPause() {
    ~~super.onStop()~~
    super.onPause()
    if (running) {
        saveOffset()
        stopwatch.stop()
    }
}
```



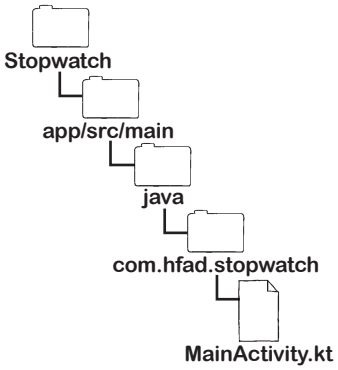
onStop() заменяется на onPause().

Продолжение на следующей странице.

Полный код MainActivity.kt (продолжение)

```
    override fun ~~onRestart()~~ onResume() {
        ~~super.onRestart()~~
        super.onResume()
        if (running) {
            setBaseTime()
            stopwatch.start()
            offset = 0
        }
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putLong(OFFSET_KEY, offset)
        savedInstanceState.putBoolean(RUNNING_KEY, running)
        savedInstanceState.putLong(BASE_KEY, stopwatch.base)
        super.onSaveInstanceState(savedInstanceState)
    }

    //Обновляет время stopwatch.base
    fun setBaseTime() {
        stopwatch.base = SystemClock.elapsedRealtime() - offset
    }

    //Сохраняет offset
    fun saveOffset() {
        offset = SystemClock.elapsedRealtime() - stopwatch.base
    }
}
```

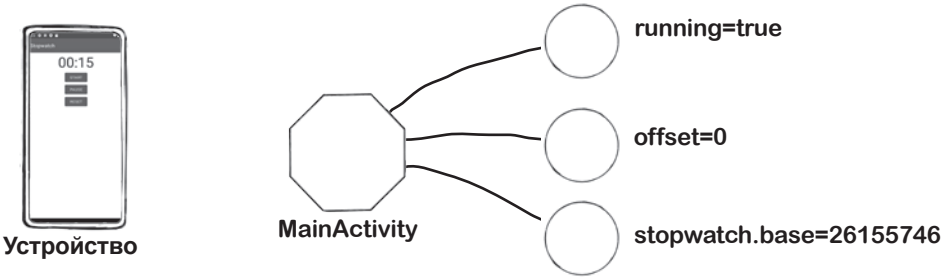
onRestart заменяется на onResume().



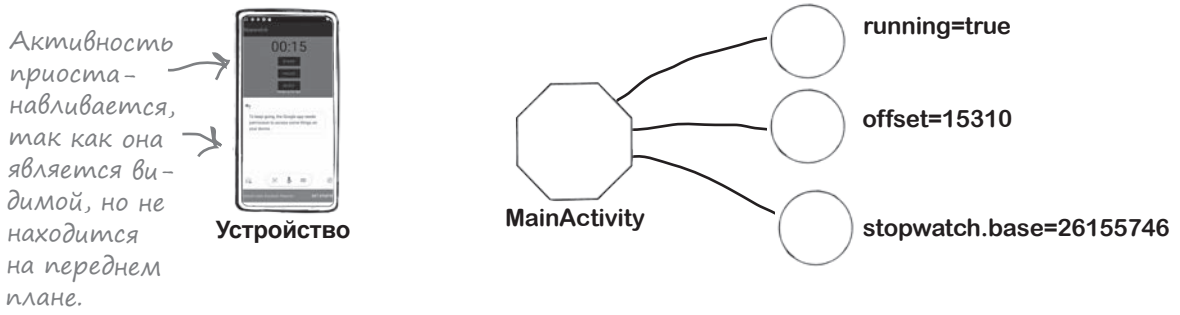
Прежде чем переходить к тестовому запуску, разберемся, что же происходит при запуске кода.

Что происходит при запуске приложения

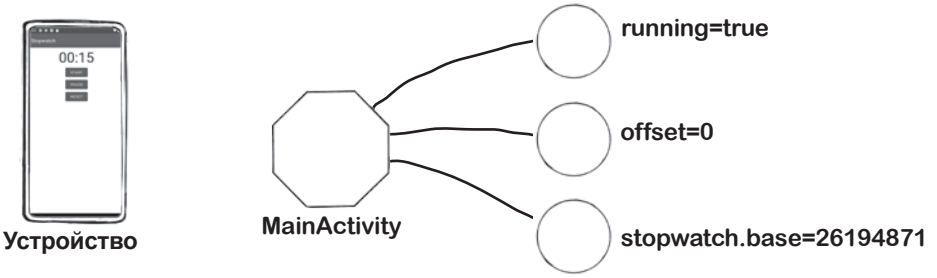
1. **Пользователь запускает приложение и щелкает на кнопке Start.**
Секундомер запускается, и свойству `running` присваивается `true`.



2. **Другая активность выходит на передний план, оставляя MainActivity частично видимой.**
Вызывается метод `onPause()`, обновляется свойство `offset`, и секундомер останавливается.



3. **MainActivity возвращается на передний план.**
Вызывается метод `onResume()`, обновляется свойство `stopwatch.bas`, запускается секундомер, а свойство `offset` обнуляется. Отсчет начинается с правильного времени.



Тест-драйв

Когда вы запускаете приложение и щелкаете на кнопке Start, запускается отсчет времени; он приостанавливается, когда приложение частично закрывается другой активностью, и запускается снова при возвращении приложения на передний план.

Поздравляем! Вы построили работающее приложение-секундомер, а также научились пользоваться методами жизненного цикла Android для управления поведением вашего приложения.

Краткая сводка методов жизненного цикла активностей

Метод	Когда вызывается	Следующий метод
onCreate()	При первоначальном создании активности. Используется для обычной подготовки, например, создания представлений. Также предоставляет объект `Bundle` с ранее сохраненным состоянием активности.	`onStart()`
onRestart()	Когда активность остановилась, но до ее повторного запуска.	`onStart()`
onStart()	Когда активность становится видимой. Далее следует вызов `onResume()`, если активность выходит на передний план, или `onStop()`, если активность становится невидимой.	`onResume()` или `onStop()`
onResume()	Когда активность выходит на передний план.	`onPause()`
onPause()	Когда ваша активность уходит с переднего плана из-за того, что другая активность возобновляет работу. Следующая активность не начнет выполняться до завершения этого метода, поэтому код метода должен завершиться быстро. Далее следует вызов `onResume()`, если активность возвращается на передний план, или `onStop()`, если она становится невидимой.	`onResume()` или `onStop()`
onStop()	Когда активность становится невидимой. Ее может закрывать другая активность или эта активность уничтожается. Далее следует вызов `onRestart()`, если активность снова становится видимой, или `onDestroy()`, если активность уничтожается.	`onRestart()` или `onDestroy()`
onDestroy()	Когда активность должна быть уничтожена или перед завершением активности.	Нет

СТАНЬ активностью

Справа приведены фрагменты кода активности. Представьте себя на месте активности и скажите, какой код будет выполняться в каждой из перечисленных ниже ситуаций. Мы решили первую задачу, чтобы вам было проще взяться за дело.

Пользователь запускает активность и начинает работать с ней.

Фрагменты A, G, D. Активность создается, становится видимой и получает фокус.

Пользователь запускает активность, начинает работать с ней, а затем переключается на другое приложение.

Сложный вопрос.

Пользователь запускает активность, начинает работать с ней, поворачивает устройство, переключается на другое приложение, а затем возвращается к активности.

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(
            savedInstanceState: Bundle?) {
        A //Выполняется код A
        ...
    }

    override fun onPause() {
        B //Выполняется код B
        ...
    }

    override fun onRestart() {
        C //Выполняется код C
        ...
    }

    override fun onResume() {
        D //Выполняется код D
        ...
    }

    override fun onStop() {
        E //Выполняется код E
        ...
    }

    fun onRecreate() {
        F //Выполняется код F
        ...
    }

    override fun onStart() {
        G //Выполняется код G
        ...
    }

    override fun onDestroy() {
        H //Выполняется код H
        ...
    }
}
```

СТАНЬ активностью. Решение

Справа приведены фрагменты кода активности. Представьте себя на месте активности и скажите, какой код будет выполняться в каждой из перечисленных ниже ситуаций. Мы решили первую задачу, чтобы вам было проще взяться за дело.

Пользователь запускает активность и начинает работать с ней.

Фрагменты A, G, D. Активность создается, становится видимой и получает фокус.

Пользователь запускает активность, начинает работать с ней, а затем переключается на другое приложение.

Фрагменты A, G, D, B, E. Активность создается, становится видимой и получает фокус. Когда пользователь переключается на другое приложение, активность теряет фокус и перестает быть видимой пользователю.

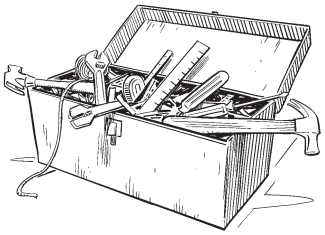
Пользователь запускает активность, начинает работать с ней, поворачивает устройство, переключается а другое приложение, а затем возвращается к активности.

Фрагменты A, G, D, B, E, H, A, G, D, B, E, C, G, D. Сначала активность создается, становится видимой и получает фокус. При повороте устройства активность теряет фокус, перестает быть видимой и уничтожается. Затем она создается снова, становится видимой и получает фокус. Когда пользователь переключается на другое приложение и обратно, активность теряет фокус, теряет видимость, снова становится видимой и получает фокус.

```
...
class MainActivity : AppCompatActivity() {

    override fun onCreate(
            savedInstanceState: Bundle?) {
        //Выполняется код A
        ...
    }

    override fun onPause() {
        //Выполняется код B
        ...
    }

    override fun onRestart() {
        //Выполняется код C
        ...
    }

    override fun onResume() {
        //Выполняется код D
        ...
    }

    override fun onStop() {
        //Выполняется код E
        ...
    }

    fun onRecreate() {
        //Выполняется код F
        ...
    }

    override fun onStart() {
        //Выполняется код G
        ...
    }

    override fun onDestroy() {
        //Выполняется код H
        ...
    }
}
```

Метод жизненного цикла с именем onRecreate() не существует.

Ваш инструментарий Android

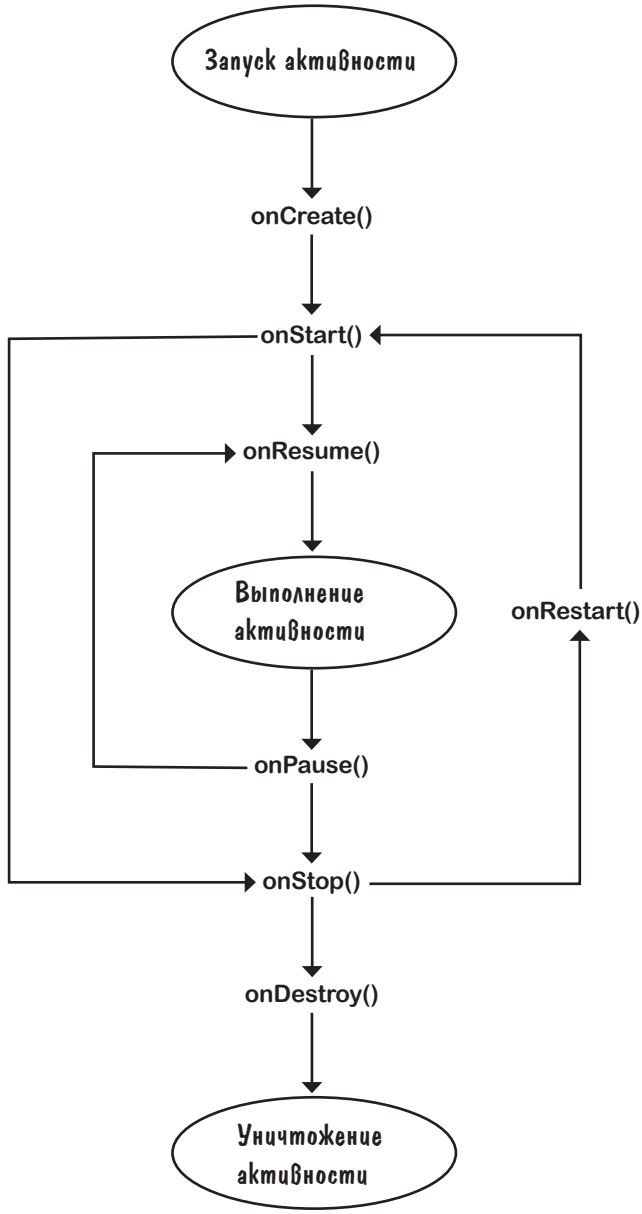


Глава 5 подходит к концу. В ней ваш инструментарий дополнился методами жизненного цикла активности.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Представление `Chronometer` реализует таймер. Класс `Chronometer` является субклассом `TextView`.
- Изменение конфигурации устройства приводит к уничтожению и повторному созданию активности.
- Ваша активность наследует методы жизненного цикла от класса `android.app.Activity`. Если вы переопределяете любые из этих методов, в переопределении необходимо вызвать метод суперкласса.
- Метод `onSaveInstanceState(Bundle)` позволяет активности сохранить свое состояние перед ее уничтожением. Объект `Bundle` может использоваться для восстановления состояния в `onCreate()`.
- Для добавления значений в `Bundle` в `onSaveInstanceState()` используется вызов `bundle.put...("name", value)`.
- Для чтения значений из `Bundle` в `onCreate()` используется вызов `bundle.get...("name")`.
- Методы `onCreate()` и `onDestroy()` связаны с созданием и уничтожением активности.
- Методы `onRestart()`, `onStart()` и `onStop()` связаны с видимостью активности.
- Методы `onResume()` и `onPause()` вызываются при получении и потере фокуса активностью.



Найди свой путь

Для большинства приложений одного экрана недостаточно. До настоящего момента мы рассматривали приложения с одним экраном; для простых приложений этого хватает. А если в вашем приложении **более сложные требования**? В этой главе вы научитесь использовать **фрагменты** и компонент **Navigation** для построения **приложений с несколькими экранами**. Вы узнаете, что **фрагменты похожи на субактивности**, имеющие собственные методы. Вы научитесь **проектировать эффективные графы навигации**. В последней части главы представлен **навигационный хост** и **навигационный контроллер**; вы узнаете, как они используются для перемещения в приложениях.

Многим приложениям недостаточно одного экрана

У всех приложений, которые вы строили до сих пор, было кое-что общее: все они состояли из одного экрана. Каждое приложение было единственной активностью с соответствующим макетом, который определяет внешний вид приложения и его взаимодействие с пользователем.

Однако большинство реальных приложений содержит *более* одного экрана. Например, почтовый клиент может использовать один экран для создания сообщений и другой экран для вывода списка полученных сообщений. Приложение-календарь может выводить список событий на одном экране и подробное описание отдельного события на другом.

Чтобы показать, как строятся многоэкранные приложения, мы создадим приложение Secret Message. Это приложение состоит из заставки, второго экрана, на котором пользователь вводит сообщение, и третьего экрана для вывода зашифрованной версии сообщения.

Приложение должно выглядеть примерно так:

Каждый экран является фрагментом

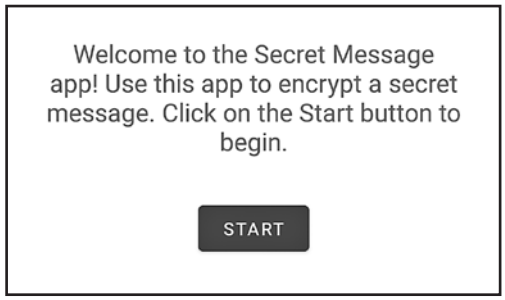
Приложение Secret Message состоит из трех разных экранов. Каждый экран будет построен в виде отдельного **фрагмента**. Фрагмент представляет собой своего рода вложенную активность, которая отображается внутри макета другой активности. Фрагмент содержит код Kotlin, который управляет его поведением, и макет, определяющий его внешний вид.

Три фрагмента, которые будут использоваться в нашем приложении:

Фрагмент содержит код Kotlin, который управляет его поведением, и макет, определяющий его внешний вид.

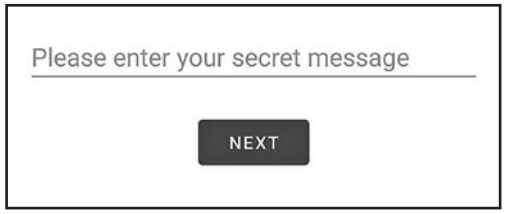
WelcomeFragment

Главный экран приложения, на котором должен размещаться краткий вводный текст и кнопка. Кнопка используется для перехода к следующему экрану — `MessageFragment`.



MessageFragment

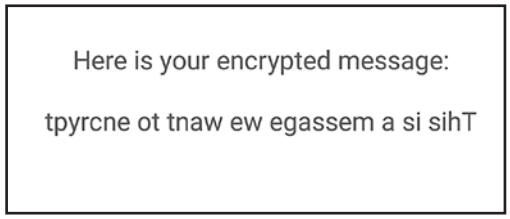
На этом экране пользователь вводит сообщение в текстовом поле. Когда пользователь щелкает на кнопке, приложение переходит к экрану `EncryptFragment`.



Все это фрагменты. Каждый фрагмент определяет экран, у каждого есть собственный макет и поведение.

EncryptFragment

Последний экран приложения. Он шифрует сообщение, введенное пользователем, и выводит результат.



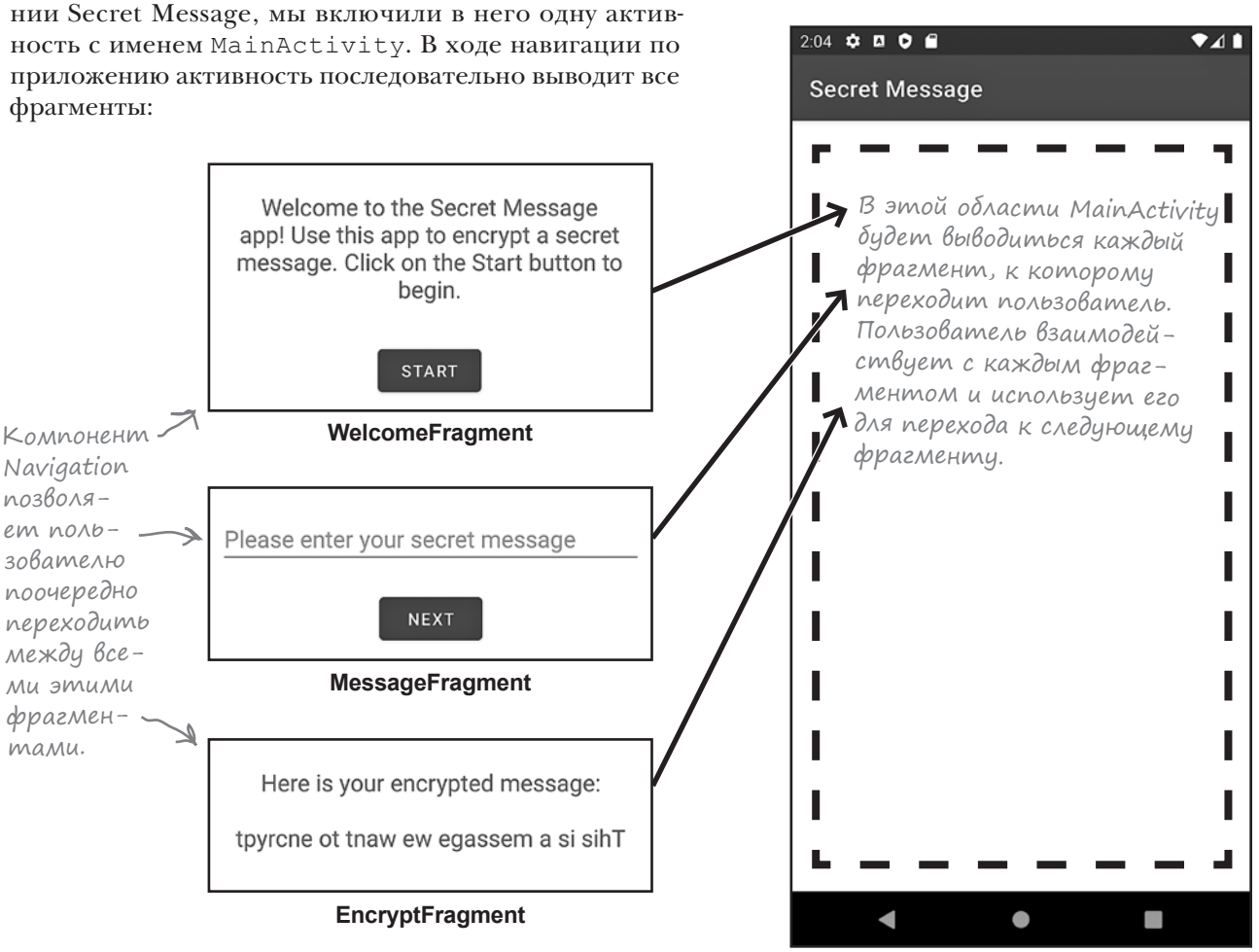
Как видите, пользователь должен каким-то образом переходить между тремя фрагментами. Как же это происходит?

Переходы между экранами с использованием компонента Navigation

Компонент Navigation является частью Android Jetpack.

Для перехода между фрагментами удобнее всего использовать компонент Android **Navigation**. Компонент Navigation входит в Android Jetpack и помогает реализовать стандартный механизм навигации в приложении.

Чтобы использовать компонент Navigation в приложении Secret Message, мы включили в него одну активность с именем `MainActivity`. В ходе навигации по приложению активность последовательно выводит все фрагменты:



В процессе работы над приложением Secret Message вы больше узнаете о фрагментах и о компоненте Navigation. А пока разберем последовательность действий для его создания.

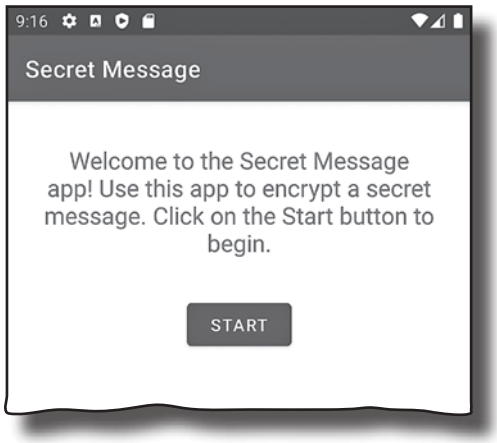
Что мы собираемся сделать

В этой главе мы собираемся построить приложение Secret Message, а в следующей завершим его.

Вот что будет сделано в этой главе:

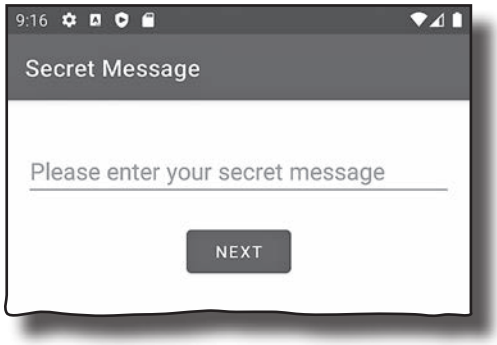
1. **Создание и отображение WelcomeFragment.**
   На этом шаге мы создадим фрагмент `WelcomeFragment`, который станет первым экраном приложения. Фрагмент будет отображаться в макете `MainActivity`, чтобы пользователь видел его при запуске приложения.

На этом шаге фрагмент WelcomeFragment отображается в макете MainActivity.



2. **Переход к MessageFragment.**
   Затем мы создадим второй фрагмент с именем `MessageFragment`. Приложение будет переходить к нему, когда пользователь щелкает на кнопке в макете `WelcomeFragment`. Для реализации навигации в приложении используется компонент Android Navigation.

Когда пользователь щелкает на кнопке Start фрагмента WelcomeFragment, MainActivity отображает MessageFragment.



Начнем с создания нового проекта для приложения.

WelcomeFragment
MessageFragment

Создание нового проекта

Для приложения Secret Message нам потребуется новый проект; создайте его по той же схеме, которая использовалась в предыдущих главах. Выберите вариант Empty Activity, введите имя «Secret Message» и имя пакета «com.hfad.secretmessage», подтвердите каталог для хранения проекта по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android.

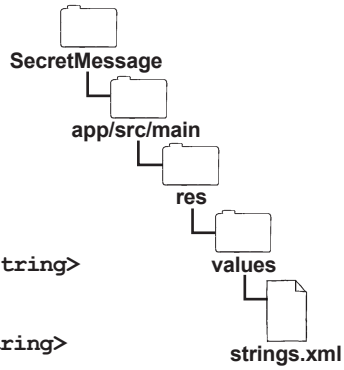
Добавление строковых ресурсов

Прежде чем создавать какие-либо фрагменты, необходимо добавить в проект несколько строковых ресурсов. Они будут использоваться для вывода текста в макетах фрагментов: надписей на кнопках, пояснительного текста на первом экране.

Чтобы добавить строковые ресурсы, откройте файл *strings.xml* в папке *SecretMessage/app/src/main/res/values*, а затем добавьте в него следующие ресурсы (выделены жирным шрифтом):

Обязательно добавьте эти строки.

```
<resources>
    <string name="app_name">Secret Message</string>
    <string name="welcome_text">Welcome to the Secret Message app!
        Use this app to encrypt a secret message.
        Click on the Start button to begin.</string>
    <string name="start">Start</string>
    <string name="message_hint">Please enter your secret message</string>
    <string name="next">Next</string>
    <string name="encrypt_text">Here is your encrypted message:</string>
</resources>
```



Часто Задаваемые Вопросы

В: Для чего нужны фрагменты? Разве нельзя создать многоэкранное приложение с несколькими активностями?

О: Да, можно создать многоэкранное приложение с несколькими активностями. Более того, еще несколько лет назад такие приложения создавались именно таким способом.

Но после выхода компонента Navigation команда Android рекомендует использовать фрагменты для построения многоэкранных приложений. Компонент Navigation предназначен прежде всего для работы с фрагментами, поэтому сейчас этот способ реализации навигации считается стандартным.

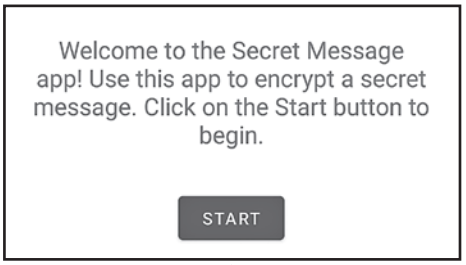
WelcomeFragment
MessageFragment

Добавление фрагмента WelcomeFragment

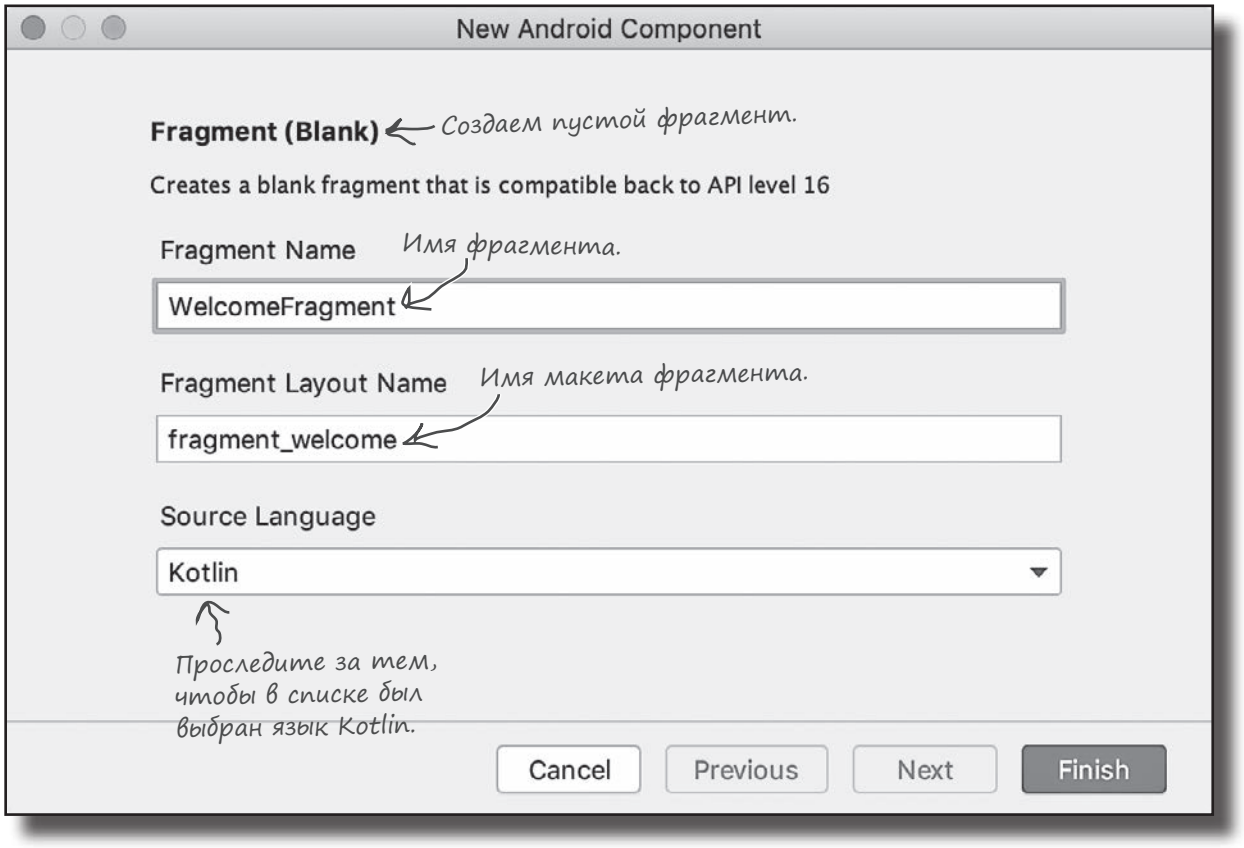
Мы собираемся добавить в проект фрагмент с именем `WelcomeFragment`. Он станет первым экраном, который видит пользователь при запуске приложения. В нем будет выводиться краткое описание приложения и кнопка.

Чтобы добавить фрагмент, выделите пакет *com.hfad.secretmessage* в папке *app/src/main/java* folder на панели проекта, откройте меню File и выберите команду New→Fragment→Fragment (Blank).

Вам будет предложено указать, как вы хотите настроить конфигурацию нового фрагмента. Введите имя фрагмента «WelcomeFragment» и имя макета «fragment_welcome». Убедитесь в том, что для проекта выбран язык Kotlin, и щелкните на кнопке Finish.



Так должен выглядеть фрагмент WelcomeFragment.



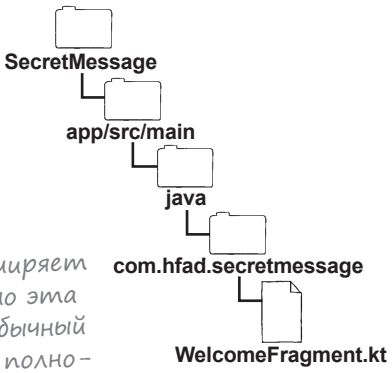
Когда вы нажмете кнопку Finish, Android Studio создаст новый фрагмент и добавит его в проект.

WelcomeFragment
MessageFragment

Как выглядит код фрагмента

При создании нового фрагмента Android Studio добавляет в проект два файла: файл с кодом Kotlin, управляющий поведением фрагмента, и файл макета, описывающий его внешний вид.

Начнем с рассмотрения кода Kotlin. Перейдите к пакету *com.hfad.secretmessage* в папке *app/src/main/java* и откройте файл *WelcomeFragment.kt*. Затем **замените код**, сгенерированный Android Studio, следующим кодом:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class WelcomeFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        // Заполнение макета для этого фрагмента
        return inflater.inflate(R.layout.fragment_welcome, container, false)
    }
}
```

WelcomeFragment расширяет класс Fragment. Именно эта строка превращает обычный традиционный класс в полноценный фрагмент.

Класс переопределяет onCreateView(). Метод вызывается тогда, когда фрагмент должен быть выведен на экран.

Сообщает Android, что фрагмент использует макет fragment_welcome.xml.

Код фрагмента похож на код активности

Приведенный выше код определяет базовый фрагмент. Как видите, код фрагмента очень похож на код активности. Однако вместо расширения `AppCompatActivity` в нем расширяется **`Fragment`**.

Класс `androidx.fragment.app.Fragment` является частью Android Jetpack и используется для определения базового фрагмента. Он предоставляет новейшую функциональность фрагмента, при этом сохраняя обратную совместимость со старыми версиями Android.

Фрагмент переопределяет метод `onCreateView()`, который вызывается в тот момент, когда Android потребуется макет фрагмента. Этот метод переопределяется почти всеми фрагментами, поэтому его стоит рассмотреть более подробно.

Не забудьте!

Обязательно обновите содержимое WelcomeFragment.kt кодом, приведенным слева.

WelcomeFragment
MessageFragment

Метод onCreateView() фрагмента

Метод `onCreateView()` вызывается в тот момент, когда Android потребуется обратиться к макету фрагмента. Переопределять этот метод необязательно, но так как вам необходимо реализовать его каждый раз, когда вы определяете фрагмент с макетом, вы будете переопределять его практически для каждого создаваемого вами фрагмента.

Метод получает три параметра:

Метод onCreateView() вызывается в тот момент, когда Android потребуется пользовательский интерфейс фрагмента.

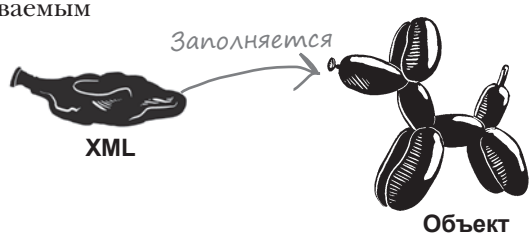
```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
}
```

Первый параметр, `LayoutInflater`, используется для заполнения макета фрагмента. Как вы узнали в главе 3, заполнение макета преобразует его представления XML в объекты.

Второй параметр, `ViewGroup?`, определяет представление `ViewGroup` в макете активности, используемое для отображения фрагмента.

Вы узнаете об этом через несколько страниц.

Последний параметр, `Bundle?`, используется в том случае, если ранее вы сохранили состояние фрагмента, а теперь хотите восстановить его. Он работает по аналогии с параметром `Bundle?`, передаваемым методу `onCreate()` активности.



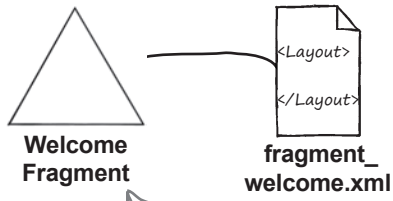
Заполнение макета фрагмента и его возвращение

Метод `onCreateView()` возвращает `View?` — заполненную версию макета фрагмента.

Для заполнения макета используется метод `inflate()` класса `LayoutInflater`:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    return inflater.inflate(R.layout.fragment_welcome, container, false)
}
```

Приведенный выше код эквивалентен вызову метода `setContentView()` активности, так как он используется для заполнения макета фрагмента и преобразования его в иерархию объектов `View`. Например, приведенный выше код заполняет макет фрагмента `WelcomeFragment` разметкой *fragment_welcome.xml*.



После того как макет фрагмента будет заполнен, иерархия `View` вставляется в макет активности и отображается на экране.

Итак, мы рассмотрели код Kotlin `WelcomeFragment` и можем рассмотреть его макет.

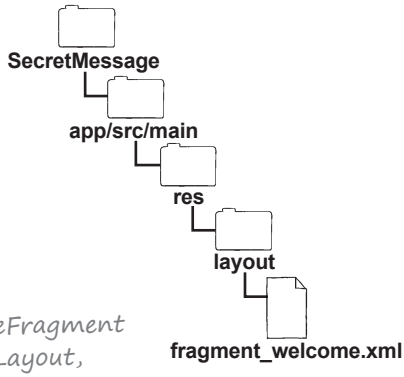
WelcomeFragment использует fragment_welcome.xml для своего макета.

Код макета фрагмента похож на код макета активности

Как говорилось выше, фрагменты используют файлы макетов для описания своего внешнего вида. Код макета фрагмента не отличается от кода макета активности, так что **в коде макета фрагмента можно использовать любые представления и группы представлений, которые вам уже известны**.

Мы заменим код макета по умолчанию, сгенерированный за нас средой Android Studio, линейным макетом, который содержит текстовое представление с кратким описанием макета, и кнопку, которая будет использоваться для перехода к следующему фрагменту позднее в этой главе.

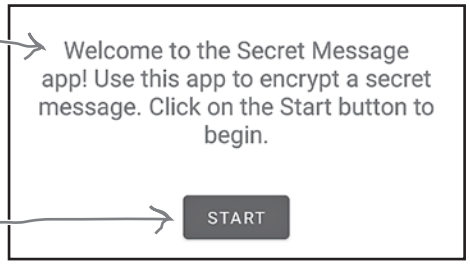
Откройте файл *fragment_welcome.xml* из папки *app/src/main/res/layout* и замените его содержимое следующим кодом:




```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".WelcomeFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:layout_marginTop="20dp"
        android:textSize="20sp"
        android:text="@string/welcome_text" />

    <Button
        android:id="@+id/start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:text="@string/start" />
</LinearLayout>
```

Для макета WelcomeFragment используется LinearLayout, но вместо него можно использовать любые типы макетов, о которых вы уже узнали.

Этот код выводит пояснительный текст, содержащийся в строковом ресурсе welcome_text.



Этот код определяет кнопку Start. Позднее она будет использоваться для перехода к следующему фрагменту.

И это весь код, который понадобится для фрагмента `WelcomeFragment` (и его макета) в данный момент. Теперь разберемся, как отобразить его в приложении.

Отображение фрагмента в FragmentContainerView

Чтобы вывести фрагмент, необходимо включить его в макет активности. Например, в данном приложении мы собираемся отобразить фрагмент `WelcomeFragment`, добавив его в файл макета `MainActivity` *activity_main.xml*.

Для добавления фрагмента в макет используется **`FragmentContainerView`**. Это разновидность `FrameLayout`, используемая для отображения фрагментов, а для ее добавления в файл макета используется код следующего вида:

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.hfad.secretmessage.WelcomeFragment" />
```

Добавляет FragmentContainerView.

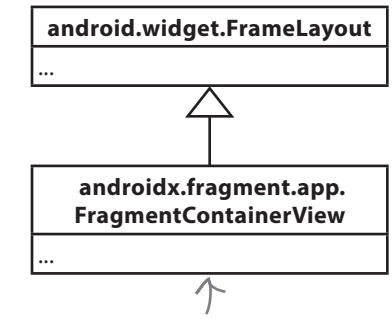
FragmentContainerView необходим идентификатор (ID).

Полное имя фрагмента вместе с пакетом.

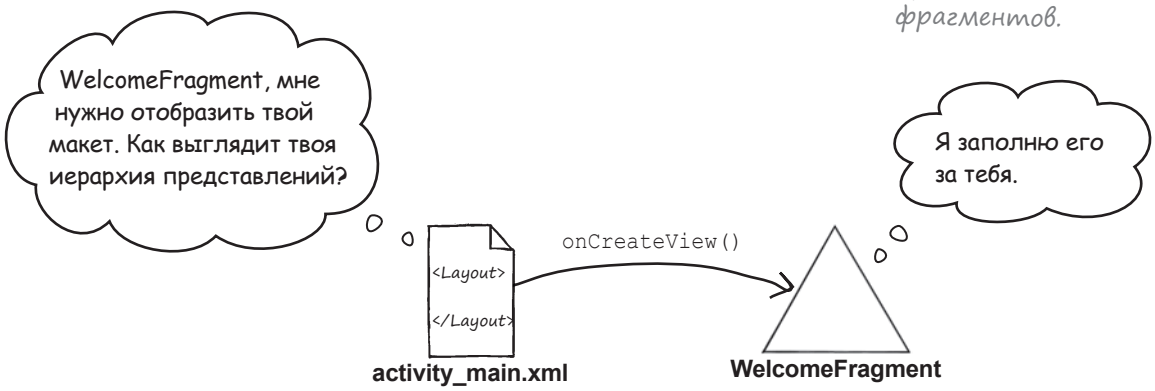
Чтобы указать, какой фрагмент необходимо отобразить, присвойте атрибуту `android:name` представления `FragmentContainerView` полное имя фрагмента вместе с пакетом. В приложении Secret Message должен отображаться фрагмент с именем `WelcomeFragment` из пакета *com.hfad.secretmessage*, поэтому значение атрибута `android:name` задается следующим образом:

```
android:name="com.hfad.secretmessage.WelcomeFragment"
```

Когда Android создает макет активности, `FragmentContainerView` заполняется объектом `View`, возвращаемым методом `onCreateView()` фрагмента. Этот объект `View` содержит пользовательский интерфейс фрагмента, так что вы можете рассматривать `FragmentContainerView` как зарезервированное место, в которое должен быть вставлен макет фрагмента:



FragmentContainerView — разновидность FrameLayout, предназначенная для хранения фрагментов.

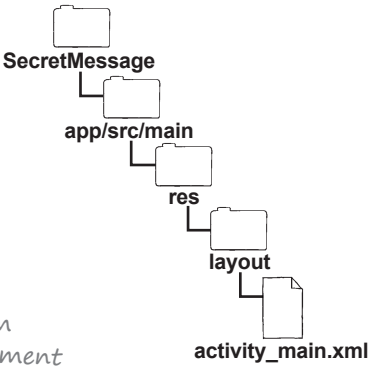


Теперь вы знаете, как добавить фрагмент в макет. Давайте добавим `WelcomeFragment` в макет `MainActivity`.

Обновление кода activity_main.xml

Мы хотим, чтобы в активности `MainActivity` отображался фрагмент `WelcomeFragment`, а это означает, что в ее макет нужно добавить `FragmentContainerView`.

Ниже приведен полный код *activity_main.xml*: обновите код, чтобы он включал приведенные ниже изменения:



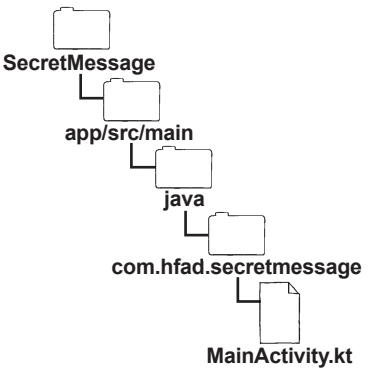

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:name="com.hfad.secretmessage.WelcomeFragment"
    tools:context=".MainActivity" />
```

Макет должен включать только FragmentContainerView, поэтому мы добавляем этот элемент в корне макета.

Отображает WelcomeFragment в макете MainActivity.

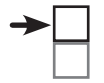
Полный код MainActivity.kt

Чтобы в `MainActivity` отображался фрагмент, никакой дополнительный код Kotlin добавлять не нужно, потому что элемент `FragmentContainerView` макета сделает все за вас. Вам остается лишь проследить за тем, чтобы код в *MainActivity.kt* выглядел так:




```
package com.hfad.secretmessage

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

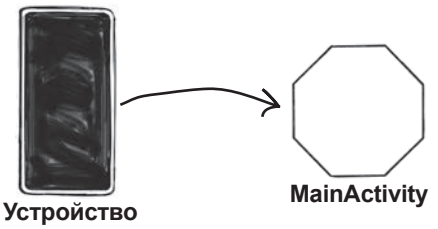
Давайте посмотрим, что происходит при запуске приложения.



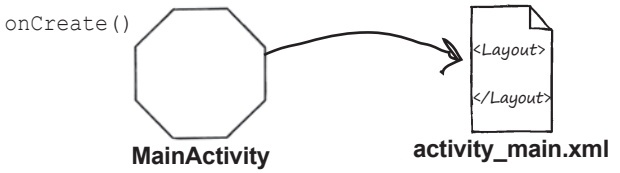
Что делает код

При запуске приложения происходят следующие события:

1. **При запуске приложения создается MainActivity.**

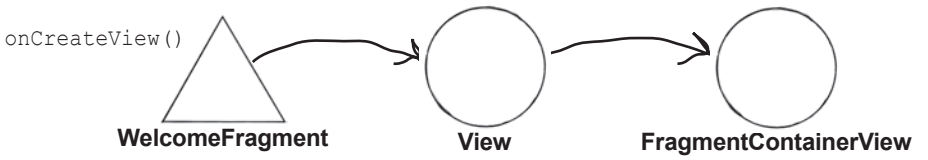


2. **Выполняется метод onCreate() активности MainActivity.**
Метод `onCreate()` указывает, что макет `MainActivity` определяется разметкой из файла *activity_main.xml*.



3. **Разметка activity_main.xml включает FragmentContainerView.**
Атрибут `android:name` элемента указывает, что он должен отобразить `WelcomeFragment`.

4. **Вызывается метод onCreateView() фрагмента WelcomeFragment, который заполняет его макет.**
Заполненная иерархия представлений `WelcomeFragment` добавляется в элемент `FragmentContainerView` в макете `MainActivity`.



WelcomeFragment
MessageFragment

Продолжение истории

5 **Наконец, MainActivity отображается на устройстве.**
Так как `FragmentContainerView` включает `WelcomeFragment`, этот фрагмент отображается на экране.

Устройство

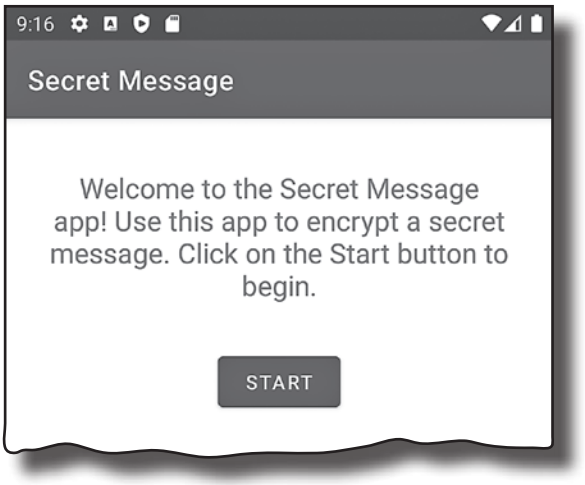
Фрагмент WelcomeFragment отображается в MainActivity. Вы сможете получше рассмотреть его в описанном ниже тест-драйве.

Итак, вы знаете, что происходит при выполнении этого кода. Давайте проведем тест-драйв приложения.

Тест-драйв

При запуске приложения Secret Message запускается активность `MainActivity`. Элемент `FragmentContainerView` из макета `MainActivity` включает `WelcomeFragment`, поэтому макет фрагмента отображается на устройстве.

WelcomeFragment отображается на экране.

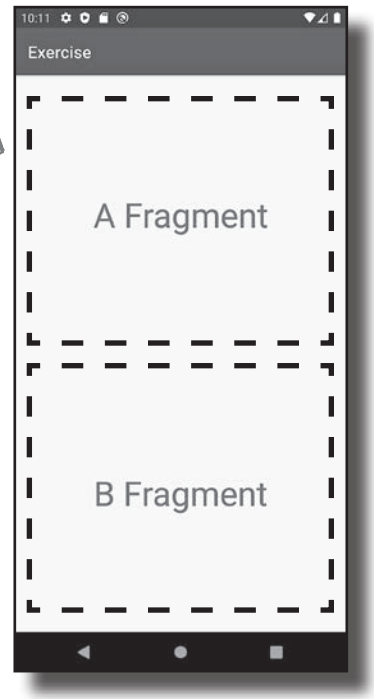


Вы узнали, как создавать и отображать фрагменты. Прежде чем строить второй фрагмент и разбираться в том, как перейти к нему, проверьте свои силы на следующем упражнении.

У бассейна

Возьмите фрагменты кода из бассейна и разместите их в пустых строках линейного макета. Каждый фрагмент можно использовать не более одного раза, причем в бассейне есть и лишние фрагменты. Ваша задача — добиться того, чтобы в линейном макете отображались два фрагмента, `AFragment` и `BFragment`, как на схеме справа. Подсказка: оба фрагмента находятся в пакете с именем *com.hfad.exercise*.

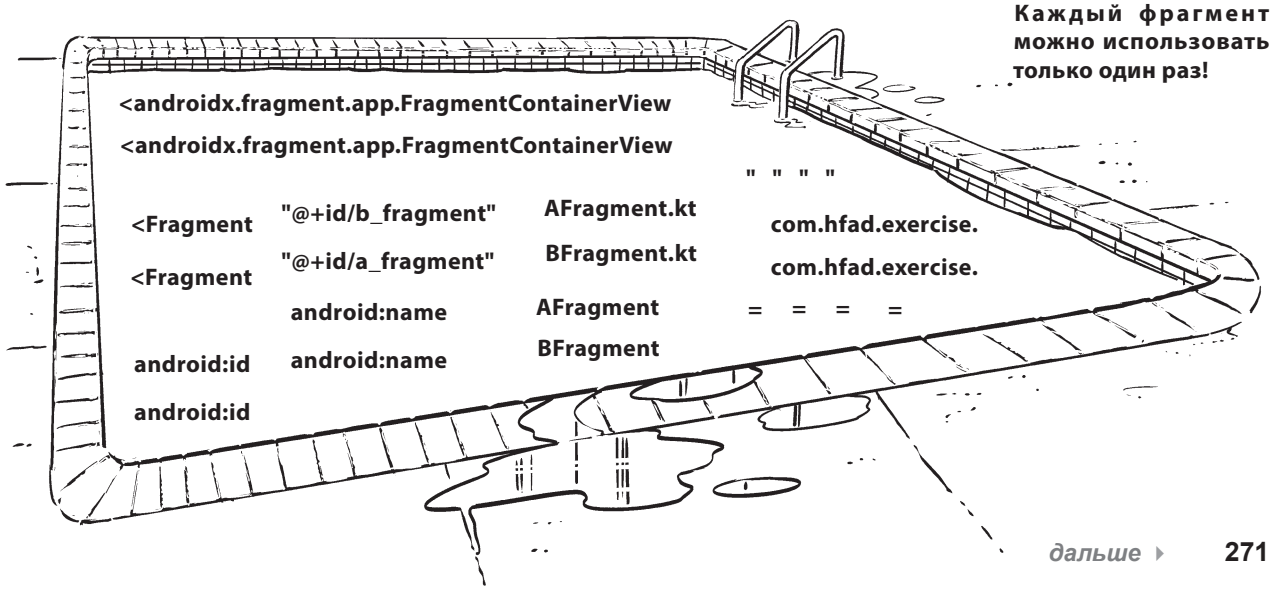
AFragment должен заполнять верхнюю половину экрана, а BFragment заполняет нижнюю половину.



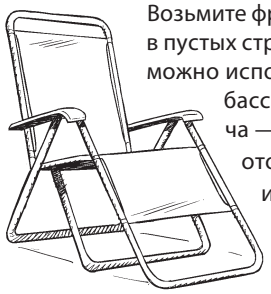

```
<LinearLayout ...
    android:orientation="vertical">
```

Некоторые атрибуты LinearLayout пропущены.

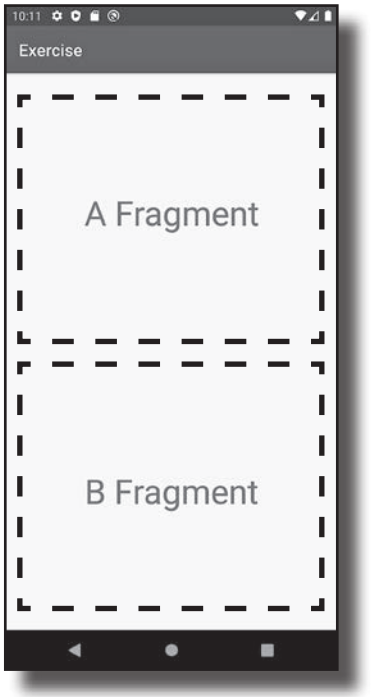
```
    .....
    .....
    .....
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />

    .....
    .....
    .....
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />
</LinearLayout>
```



У бассейна. Решение



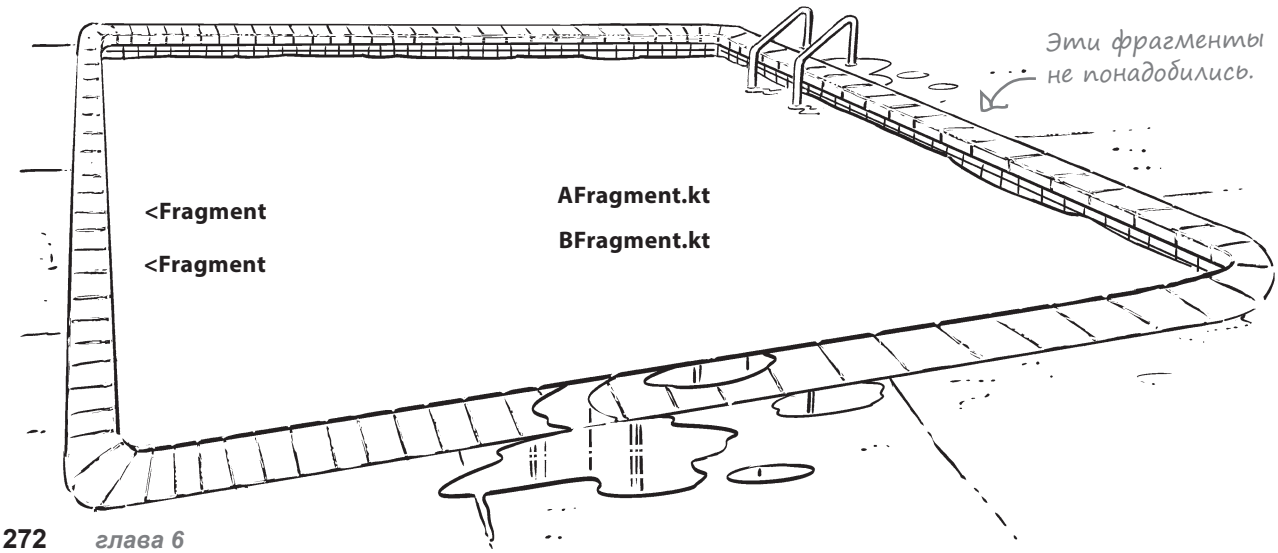
Возьмите фрагменты кода из бассейна и разместите их в пустых строках линейного макета. Каждый фрагмент можно использовать не более одного раза, причем в бассейне есть и лишние фрагменты. Ваша задача — добиться того, чтобы в линейном макете отображались два фрагмента, `AFragment` и `BFragment`, как на схеме справа. Подсказка: оба фрагмента находятся в пакете с именем *com.hfad.exercise*.




```
<LinearLayout ...
    android:orientation="vertical">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/a_fragment"
        android:name="com.hfad.exercise.AFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/b_fragment"
        android:name="com.hfad.exercise.BFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
</LinearLayout>
```

Этот элемент используется для добавления фрагмента в макет.

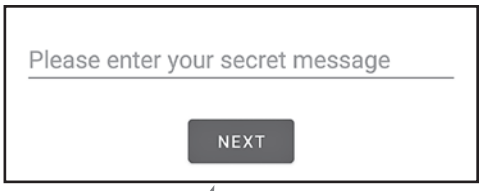
Атрибут android:name используется для задания полного имени фрагмента.



WelcomeFragment
MessageFragment

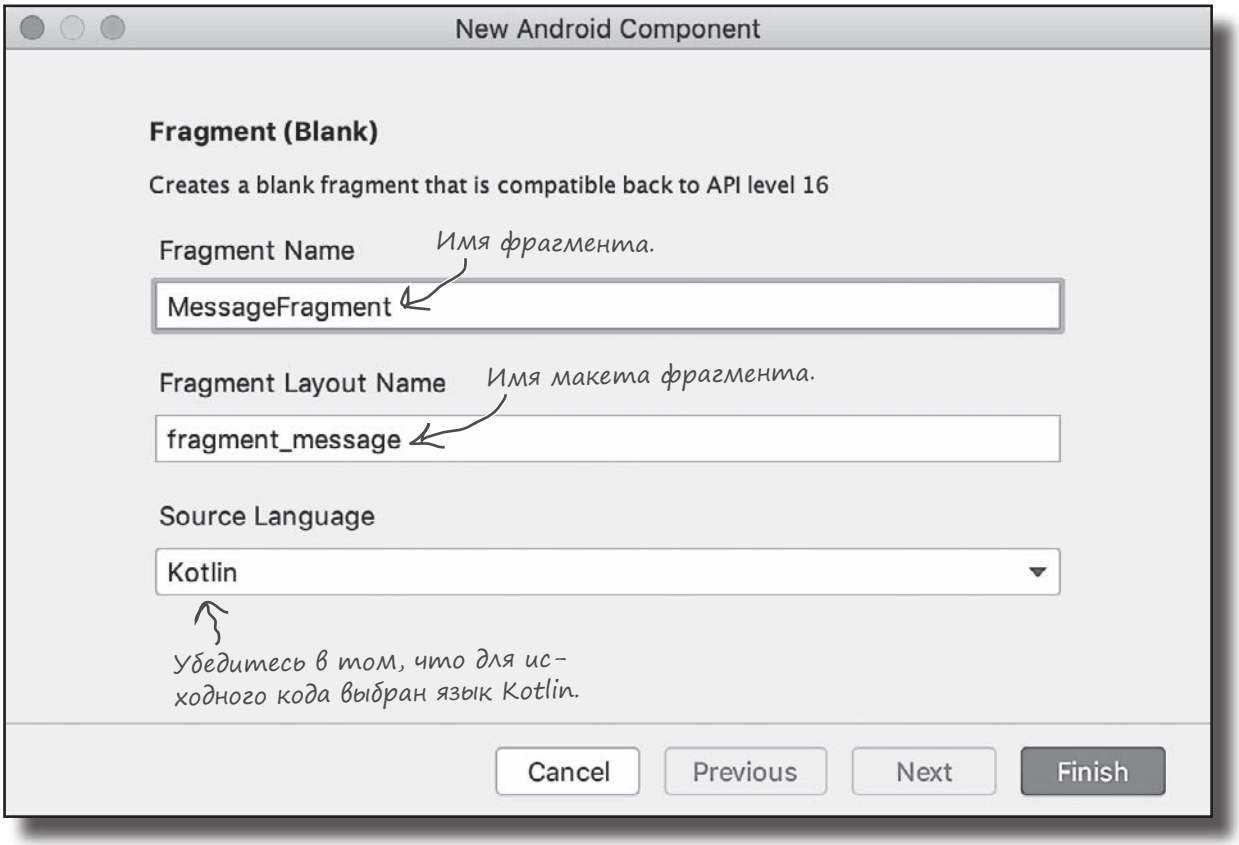
Создание MessageFragment

К настоящему моменту мы создали фрагмент с именем `WelcomeFragment`, который отображается в макете `MainActivity`. Затем мы создадим новый фрагмент с именем `MessageFragment`; приложение будет переходить к этому фрагменту, когда пользователь щелкает на кнопке Start в `WelcomeFragment`.



Так будет выглядеть MessageFragment.

Фрагмент `MessageFragment` добавляется точно так же, как добавлялся фрагмент `WelcomeFragment`. Выделите пакет *com.hfad.secretmessage* в папке *app/src/main/java* на панели Project Explorer, откройте меню File и выберите команду New→Fragment→Fragment (Blank). Введите имя фрагмента «MessageFragment» и имя макета «fragment_message» и убедитесь в том, что выбран язык Kotlin. Затем щелкните на кнопке Finish, чтобы добавить фрагмент и его макет в проект.



Обновление макета MessageFragment

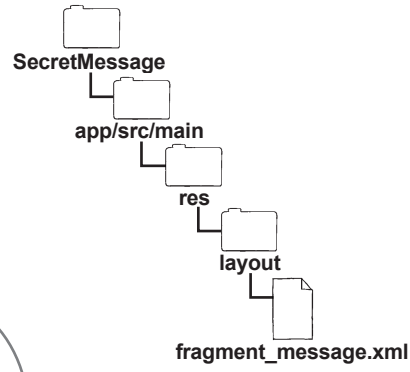
WelcomeFragment
MessageFragment

При создании `MessageFragment` Android Studio добавляет в проект два новых файла: *MessageFragment.kt* (определяет поведение фрагмента) и *fragment_message.xml* (определяет его внешний вид). Мы обновим оба файла, начиная с макета.

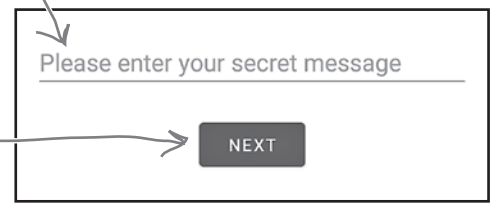
Фрагмент должен содержать текстовое поле, в котором пользователь будет вводить сообщение, и кнопку, которая позднее будет использоваться для навигации. Вы уже знаете код, используемый для добавления этих представлений, поэтому обновите разметку *fragment_message.xml* и приведите ее к следующему виду:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".MessageFragment">

    <EditText
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:hint="@string/message_hint"
        android:inputType="textMultiLine" />

    <Button
        android:id="@+id/next"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:text="@string/next" />
</LinearLayout>
```

Для макета фрагмента используется элемент LinearLayout.



Пользователь вводит сообщение в текстовом поле.



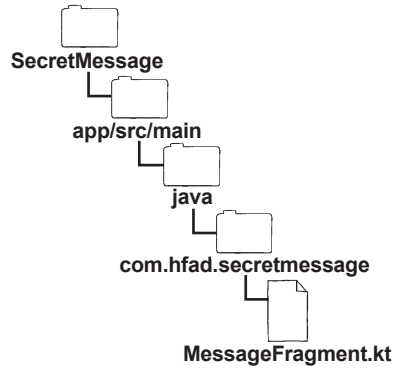
В следующей главе кнопка будет использоваться для перехода к другому фрагменту.

Это вся разметка, необходимая для макета `MessageFragment`. Перейдем к обновлению его кода Kotlin.

Обновление MessageFragment.kt

Код `MessageFragment` на Kotlin определяет поведение фрагмента. Пока от нас требуется совсем немного — нужно убедиться в том, что среда Android Studio не добавила лишний код, из-за которого фрагмент будет работать не так, как требуется.

Откройте пакет *com.hfad.secretmessage* из папки *app/src/main/java* и откройте файл *MessageFragment.kt*. Затем замените код, сгенерированный Android Studio, следующим:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class MessageFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        //Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_message, container, false)
    }
}
```

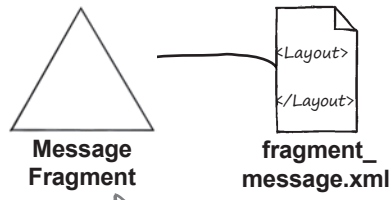
MessageFragment расширяет класс Fragment.

Фрагмент переопределяет onCreateView().

Используйте fragment_message.xml для макета фрагмента.

Приведенный выше код — все, что необходимо *MessageFragment.kt* для определения базового фрагмента. Как и код, приведенный для `WelcomeFragment`, он расширяет класс `Fragment` и переопределяет метод `onCreateView()`. Этот метод заполняет макет фрагмента и возвращает его корневое представление. Он вызывается каждый раз, когда приложению потребуется отобразить фрагмент.

Мы завершили написание всей разметки и кода Kotlin, необходимого для `MessageFragment`. Далее нужно заставить `WelcomeFragment` переходить к этому фрагменту по нажатии кнопки. Как же это делается?



MessageFragment использует файл fragment_message.xml для своего макета.

WelcomeFragment
MessageFragment

Применение компонента Navigation для перехода между фрагментами

Как говорилось ранее в этой главе, стандартный способ навигации между фрагментами использует компонент Android Navigation.

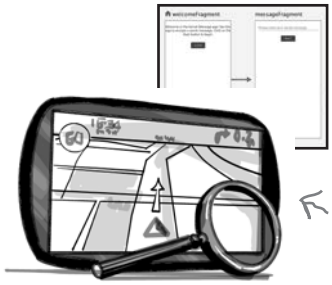
Компонент Navigation является частью Android Jetpack — семейства библиотек, плагинов и инструментов, которые вы добавляете в свои проекты. Он в высшей степени гибок и упрощает многие сложности навигации между фрагментами (например, транзакции фрагментов и операции со стеком возврата), которые прежде реализовывались намного сложнее.

Механизм навигации между фрагментами состоит из трех основных частей:

Граф навигации

Граф навигации содержит всю информацию, относящуюся к навигации, которая требуется вашему приложению. Он описывает возможные пути, по которым может переходить пользователь в процессе навигации в приложении.

Граф навигации представляет собой ресурс в формате XML, но обычно он редактируется в визуальном редакторе.



Граф навигации

Граф навигации — что-то вроде GPS-навигатора, который сообщает вам, как добраться до того или иного места.

Хост навигации

Хост навигации представляет собой пустой контейнер, используемый для отображения фрагмента, к которому вы переходите. Хост навигации добавляется в макет активности.

Хост навигации

Хост навигации — пустой контейнер, который добавляется в макет активности.

Контроллер навигации

Контроллер навигации управляет тем, какой фрагмент отображается в хосте навигации при переходах пользователя в приложении. Для взаимодействия с контроллером навигации используется код Kotlin.

Контроллер навигации

Контроллер навигации управляет тем, какой фрагмент отображается в процессе навигации в приложении.

Все три части будут использоваться для реализации навигации в приложении Secret Message. Начнем с добавления библиотеки компонента Navigation в проект.

WelcomeFragment
MessageFragment

Добавление компонента Navigation в проект с использованием Gradle

Как вы узнали в главе 4, для включения в приложение дополнительных библиотек, инструментов и плагинов необходимо внести изменения в файлы *build.gradle*. Когда вы создаете новый проект, Android Studio автоматически включает два таких файла: для проекта и для приложения.

Чтобы добавить компонент Navigation, нужно отредактировать *обе* версии *build.gradle*. Начнем с обновления версии проекта.

Всегда щелкайте на ссылке Sync Now после обновления файлов *build.gradle*.

Любые внесенные изменения (такие, как добавление библиотек) вступают в силу только после синхронизации.

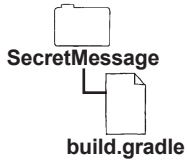
Добавление номера версии в файл build.gradle проекта

Начнем с добавления в файл *build.gradle* проекта новой переменной, которая определяет используемую версию компонента Navigation. Использование переменной для номера версии означает, что при добавлении дополнительных библиотек компонента Navigation (мы займемся этим в следующей главе) номера версий отдельных библиотек не смешиваются и сохраняют последовательность.

Чтобы добавить переменную, откройте файл *SecretMessage/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `buildscript`:

```
buildscript {
    ext.nav_version = "2.3.5"
    ...
}
```

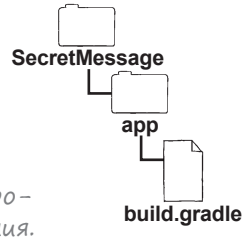
Используется эта версия компонента Navigation.



Добавление зависимости в файл build.gradle приложения

Затем зависимость для библиотеки необходимо добавить в версию файла *build.gradle* приложения.

Откройте файл *SecretMessage/app/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в разделе `dependencies`:



Не забудьте добавить эту строку в файл build.gradle приложения.

```
dependencies {
    ...
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    ...
}
```

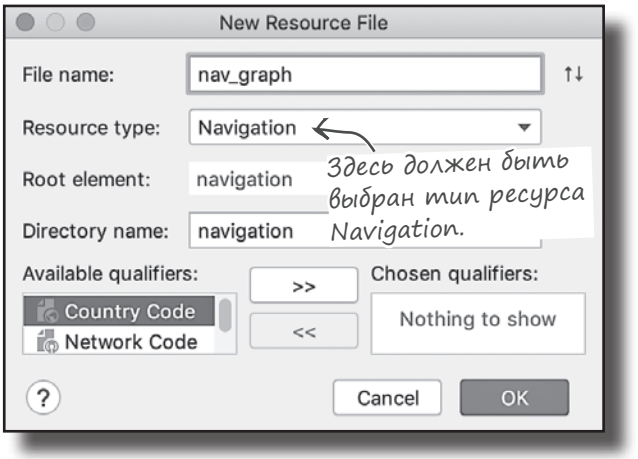
После внесения этих изменений щелкните на ссылке Sync Now, появившейся в верхней части редактора кода. Она синхронизирует внесенные изменения в проекте и добавляет библиотеку.

WelcomeFragment
MessageFragment

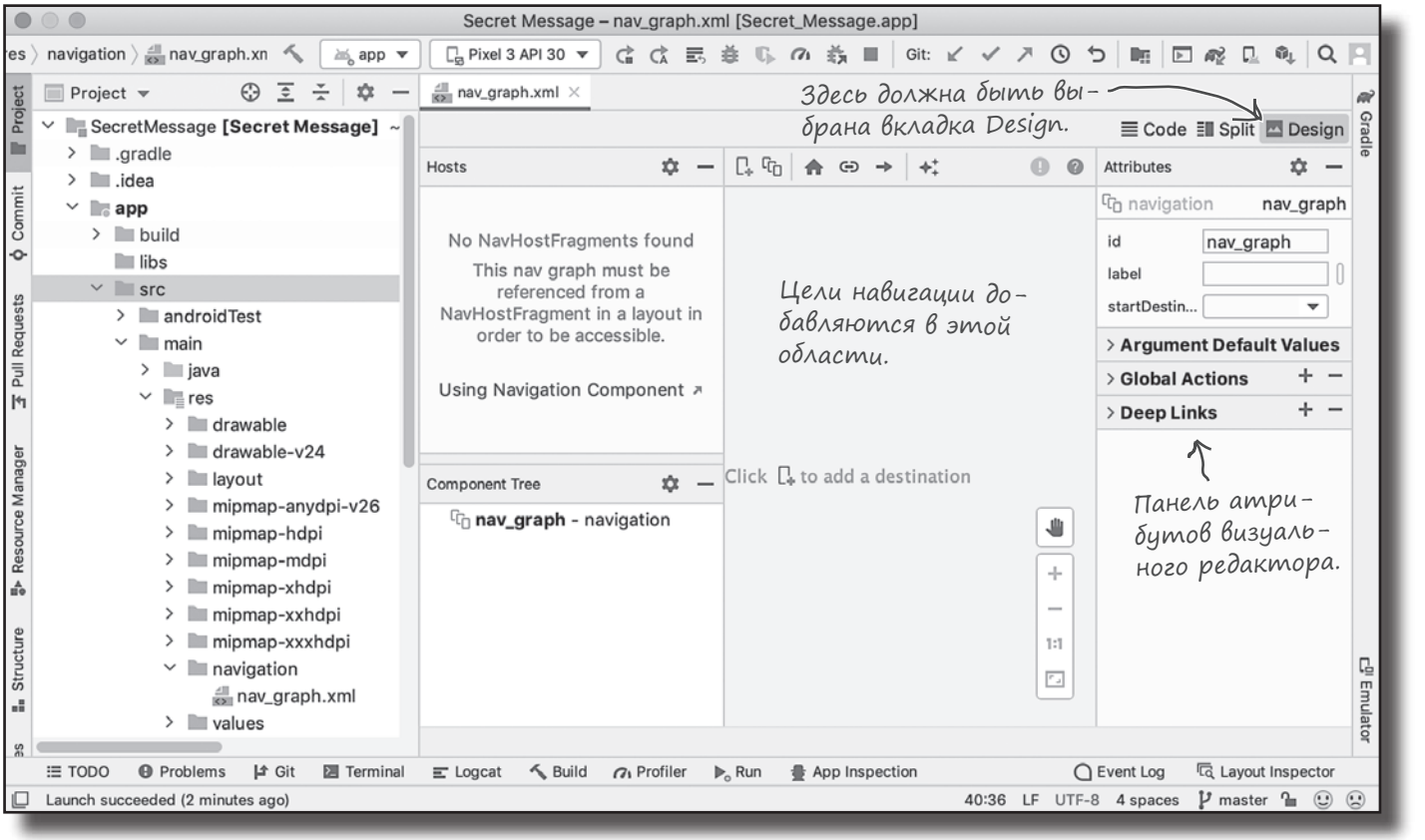
Создание графа навигации

После добавления главной библиотеки компонента Navigation в проект Secret Message можно переходить к реализации навигации.

Начнем с добавления графа навигации в проект. Выберите папку *SecretMessage/app/src/main/res* на панели проекта, затем выберите команду File→New→Android Resource File. Введите имя файла «nav_graph», выберите тип ресурса «Navigation» и щелкните на кнопке OK. Кнопка создает файл с пустым графом навигации *nav_graph.xml* в папке *SecretMessage/app/src/main/res/navigation*.



После того как граф навигации будет создан, откройте его (если он не был открыт ранее) двойным щелчком на файле *nav_graph.xml* на панели проекта. Файл открывается в визуальном редакторе графов навигации, который выглядит так:



WelcomeFragment
MessageFragment

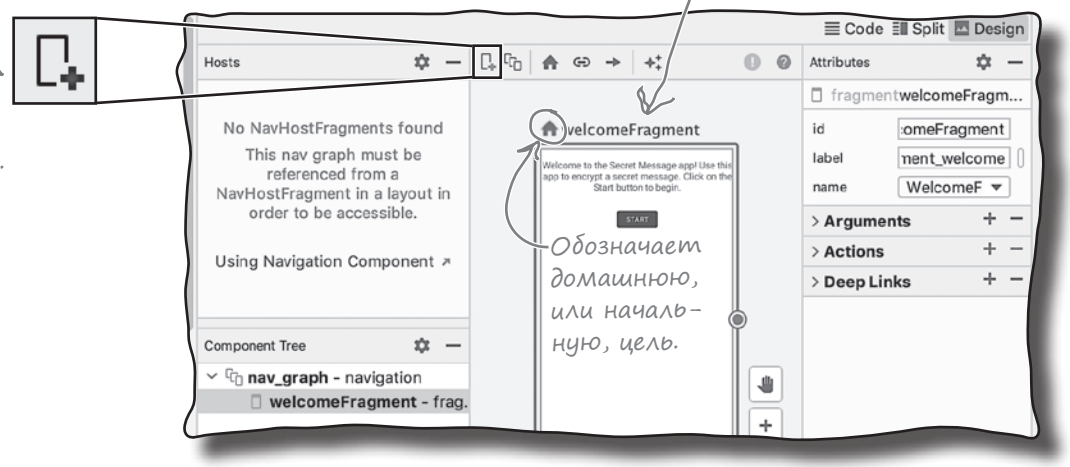
Добавление фрагментов в граф навигации

Мы хотим, чтобы пользователь переходил от `WelcomeFragment` к `MessageFragment`, поэтому эти фрагменты нужно добавить в граф навигации как **цели**. Цель представляет собой экран в приложении — обычно фрагмент, — к которому может переходить пользователь.

Начнем с добавления `WelcomeFragment`, так как это первый экран, который видит пользователь при запуске приложения. Щелкните на кнопке New Destination в верхней части визуального редактора, выберите вариант «fragment_welcome» (макет `WelcomeFragment`). При этом `WelcomeFragment` добавляется в граф навигации, после чего он будет выглядеть так:

WelcomeFragment добавляется в граф навигации как новая цель.

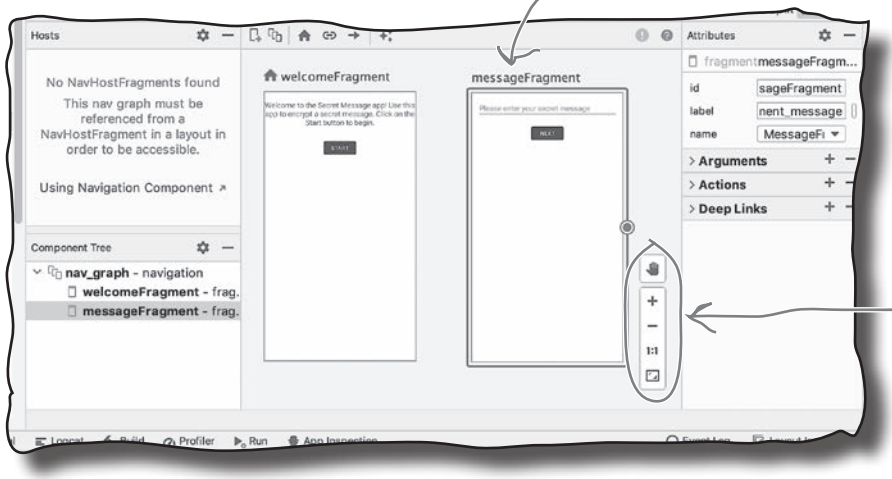
Эта кнопка добавляет новые цели.



Обозначает домашнюю, или начальную, цель.

Затем добавьте `MessageFragment` в граф навигации. Щелкните на кнопке New Destination button и выберите вариант «fragment_message». При этом в граф навигации добавляется второй фрагмент:

MessageFragment добавляется как вторая цель.



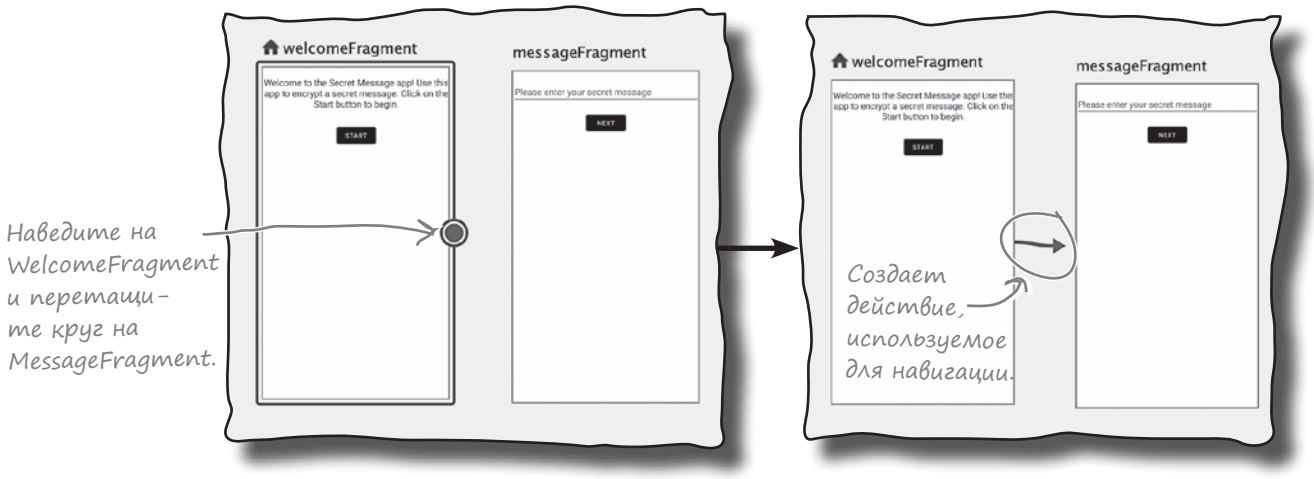
Эти инструменты предназначены для изменения размера или позиции фрагментов, чтобы их было удобнее просматривать.

Соединение фрагментов действиями

WelcomeFragment
MessageFragment

Затем необходимо указать, что пользователь может перейти от `WelcomeFragment` к `MessageFragment`; для этого используются **действия**. Действия соединяют цели на графе навигации и определяют возможные пути, по которым может перемещаться пользователь в процессе навигации в приложении.

Мы добавим действие для перехода от `WelcomeFragment` к `MessageFragment`, так как именно в этом направлении пользователь должен перемещаться в приложении. Наведите указатель мыши на `WelcomeFragment` в визуальном редакторе, затем щелкните на круге, появившемся у правого края, и перетащите его на `MessageFragment`. При этом два фрагмента соединяются стрелкой — действием:

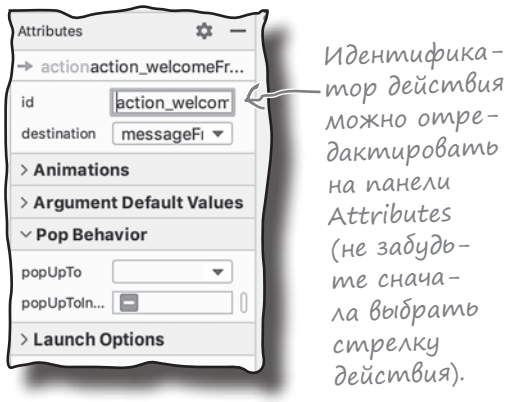


Каждому действию необходим уникальный идентификатор

Каждое действие должно иметь уникальный идентификатор. Android использует этот идентификатор для определения того, какая цель должна отображаться при переходе пользователя в приложении.

Каждый раз, когда вы создаете действие, Android Studio присваивает ему идентификатор по умолчанию. Этот идентификатор, как и все остальные свойства действия, можно отредактировать на панели Attributes справа от графа навигации.

Действию, которое вы только что создали, должен быть присвоен идентификатор «action_welcomeFragment_to_messageFragment», используемый в коде этой главы. Чтобы убедиться в этом, выделите действие (стрелку) в визуальном редакторе и щелкните на значении его атрибута `id` на панели Attributes. Этот идентификатор будет использоваться через несколько страниц.



Графы навигации как ресурсы XML

Граф навигации, как и макет, в действительности состоит из кода разметки XML. Чтобы просмотреть его, щелкните на кнопке Code в верхней части визуального редактора.

Вот как выглядит разметка XML для графа навигации приложения Secret Message *nav_graph.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>

    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" />
</navigation>
```

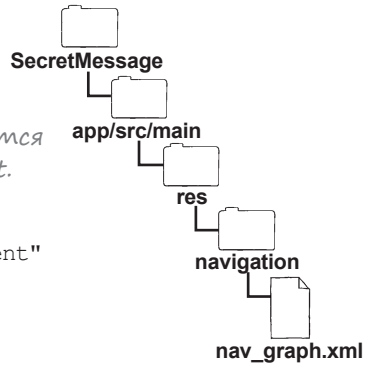
Граф навигации начинается с корневого элемента <navigation>.

Навигация начинается с WelcomeFragment.

Этот раздел описывает WelcomeFragment.

WelcomeFragment содержит действие перехода к MessageFragment.

Этот раздел описывает MessageFragment.

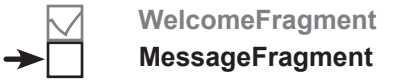


Как видите, *nav_graph.xml* содержит корневой элемент `<navigation>` и два элемента `<fragment>`: для `WelcomeFragment` и для `MessageFragment`. Элемент `<fragment>` для `WelcomeFragment` включает дополнительный элемент `<action>`, который описывает только что добавленное действие.

Итак, граф навигации создан, и мы можем перейти к следующей части компонента Navigation.

Графы навигации обычно редактируются в визуальном редакторе, но иногда бывает полезно обратиться к разметке XML.

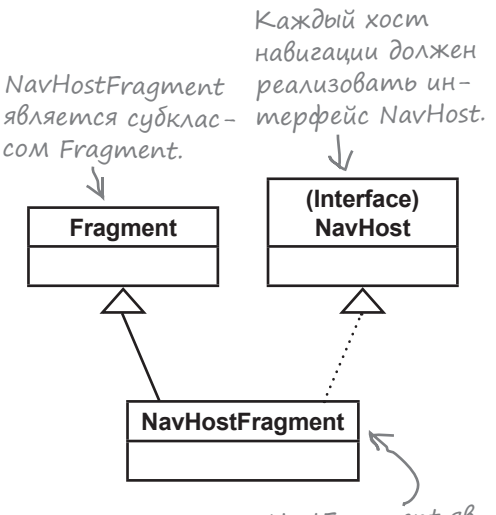
Добавление хоста навигации в макет при помощи FragmentContainerView



Как упоминалось ранее, компонент Navigation состоит из трех основных частей: графа навигации, определяющего возможные пути навигации; хоста навигации, отображающего цели; и контроллера навигации, управляющего тем, какая цель должна отображаться. Мы только что создали граф навигации, пора сделать следующий шаг — добавить хост навигации.

Чтобы добавить хост навигации, следует включить его в макет активности. К счастью, компонент Navigation включает встроенный хост с именем **`NavHostFragment`**, так что вам не придется писать его самостоятельно. Он является субклассом `Fragment`, который реализует интерфейс `NavHost`.

Так как `NavHostFragment` является разновидностью фрагмента, для его добавления в файл макета используется `FragmentContainerView`. Код выглядит примерно так:



Представление FragmentContainerView.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true" />
```

FragmentContainerView содержит NavHostFragment.

Два дополнительных атрибута.

Приведенный выше код напоминает код `FragmentContainerView`, который вы уже видели, но он включает два дополнительных атрибута: **`app:navGraph`** и **`app:defaultNavHost`**.

Атрибут `app:navGraph` сообщает хосту навигации, какой граф навигации тот должен использовать, — в данном случае *nav_graph.xml*. Граф навигации определяет фрагмент, который должен отображаться первым (начальную цель), и обеспечивает возможность перемещения пользователям между целями.

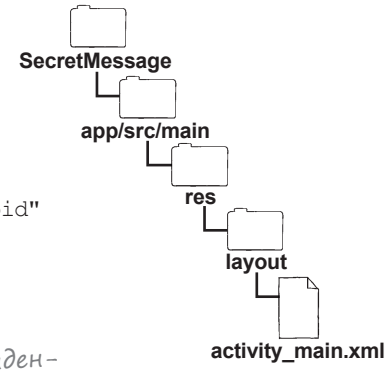
Атрибут `app:defaultNavHost` обеспечивает взаимодействие хоста навигации с кнопкой возврата на устройстве: эта тема рассматривается в следующей главе.

NavHostFragment — хост навигации. Он отображает фрагмент, к которому переходит пользователь, начиная с исходной цели графа навигации.

WelcomeFragment
MessageFragment

Добавление NavHostFragment в activity_main.xml

Мы добавим хост навигации в макет `MainActivity`, использующий созданный нами граф навигации. Обновите код разметки *activity_main.xml* и включите в него изменения, выделенные жирным шрифтом.




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    ~~android:id="@+id/fragment_container_view"~~
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    ~~android:name="com.hfad.secretmessage.WelcomeFragment"~~
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true"
    tools:context=".MainActivity" />
```

Мы используем дополнительные атрибуты из этого пространства имен.

Измените идентификатор.

Обновите значение name, чтобы использовался фрагмент NavHostFragment.

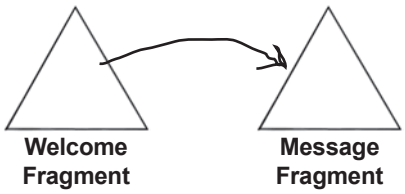
Укажите файл ресурса, содержащий граф навигации.

Обеспечивает взаимодействие хоста навигации с кнопкой возврата на устройстве. Эта тема более подробно рассматривается в главе 7.

Приложение должно переходить между фрагментами

Мы создали граф навигации и связали его с хостом навигации, который содержится в элементе `FragmentContainerView` в макете `MainActivity`. При запуске приложения будет отображаться `WelcomeFragment` — начальная цель в графе навигации.

Последнее, что осталось сделать в этой главе, — реализовать переход от `WelcomeFragment` к `MessageFragment`, когда пользователь щелкал на кнопке Start в макете `WelcomeFragment`. Давайте разберемся, как это делается.



Добавление OnClickListener для кнопки

WelcomeFragment
MessageFragment

Чтобы переходить от `WelcomeFragment` к `MessageFragment`, сначала необходимо позаботиться о том, чтобы кнопка Start фрагмента `WelcomeFragment` реагировала на щелчки. Для этого мы добавим к ней слушатель `OnClickListener`.

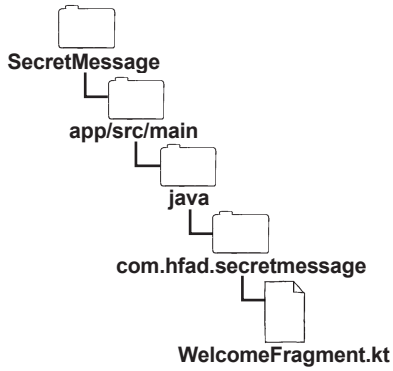
Ранее мы добавили `OnClickListener` для кнопки *активности*. Для этого мы сначала получали ссылку на кнопку вызовом `findViewById()`, а затем вызывали ее метод `setOnClickListener`. Этот код содержался в методе `onCreate()` активности, так как именно он первым получает доступ к представлениям в своем макете.

Но когда требуется добавить `OnClickListener` к кнопке *фрагмента*, ситуация слегка меняется.

Код OnClickListener для фрагмента несколько отличается

Первое отличие заключается в том, что слушатель `OnClickListener` добавляется к кнопке *фрагмента* в методе **`onCreateView()`** фрагмента, а не в `onCreate()`. Дело в том, что фрагмент впервые получает доступ к своим представлениям в `onCreateView()`, поэтому этот метод лучше подходит для назначения `OnClickListener`.

Второе отличие заключается в том, что **класс `Fragment` не содержит метод `findViewById()`**, поэтому вы не сможете воспользоваться этим методом для получения ссылок на любые представления. Впрочем, вместо этого можно вызвать `findViewById()` для корневого представления фрагмента.



А вот как выглядит код добавления `OnClickListener` для представления в коде фрагмента: мы добавим его в `WelcomeFragment` через пару страниц:

```
...
class WelcomeFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_welcome, container, false)
        val startButton = view.findViewById<Button>(R.id.start)

        startButton.setOnClickListener {
            //Код, выполняемый по щелчку на кнопке
        }
        return view
    }
}
```

Получает ссылку на корневое представление фрагмента.

Использует корневое представление для получения ссылки на кнопку Start фрагмента.

Возвращает корневое представление.

Итак, вы знаете, как добавить `OnClickListener` к кнопке фрагмента. Теперь посмотрим, как реализовать переход при щелчке на кнопке.

WelcomeFragment
MessageFragment

Класс Fragment не является субклассом Activity.

И хотя у фрагментов много общего с активностями, класс `Fragment` не расширяет `Activity` и поэтому не наследует никакие из его методов.

Вместо этого класс `Fragment` определяет собственный набор методов. Многие из них похожи на методы, наследуемые активностями, но методов вроде `findViewById()` в нем нет.

В этой книге вы больше узнаете о фрагментах и их методах. А пока посмотрим, какой код необходимо добавить в `WelcomeFragment`, чтобы при щелчке на кнопке приложение переходило к `MessageFragment`.

Получение контроллера навигации

Каждый раз, когда вы хотите перейти к новому фрагменту, сначала необходимо получить ссылку на контроллер навигации. Для этого следует вызвать метод **findNavController()** для корневого объекта `View`. Например, следующий код получает ссылку на контроллер навигации, связанный с корневым объектом представления с именем `view`:

```
val navController = view.findNavController()
```

Контроллер навигации

Выбор цели перехода при помощи действия

Получив контроллер навигации, вы отдаете ему команду перейти к новой цели, вызывая его метод **navigate()**. Этот метод получает один параметр: идентификатор действия навигации.

Как вы, возможно, помните, при создании графа навигации мы включили действие для перехода от `WelcomeFragment` к `MessageFragment`. Этому действию был присвоен идентификатор "action_welcomeFragment_to_messageFragment".

Если передать этот идентификатор методу `navigate()` контроллера навигации, контроллер увидит, что действие переходит от `WelcomeFragment` к `MessageFragment` и использует его для перехода к новому фрагменту.

Код выглядит так:

```
view.findNavController().navigate(R.id.action_welcomeFragment_to_messageFragment)
```

После получения ссылки на контроллер навигации вы отдаете ему команду выполнить переход по идентификатору действия.

Когда пользователь щелкает на кнопке Start фрагмента `WelcomeFragment`, приложение должно переходить к `MessageFragment`. Соответственно, мы включаем следующий код в слушатель `OnClickListener` кнопки Start:

```
val view = inflater.inflate(R.layout.fragment_welcome, container, false)
val startButton = view.findViewById<Button>(R.id.start)
startButton.setOnClickListener {
    view.findNavController()
        .navigate(R.id.action_welcomeFragment_to_messageFragment)
}
```

Получаем контроллер навигации.

Переход к MessageFragment.

Полный код `WelcomeFragment` приведен на следующей странице.

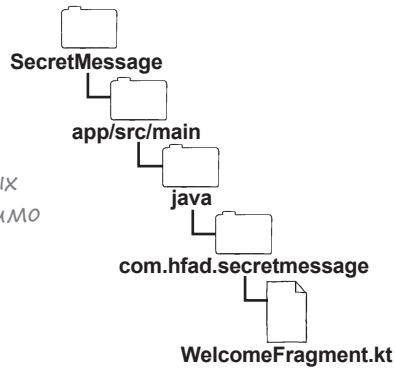
WelcomeFragment
MessageFragment

Полный код WelcomeFragment.kt

Ниже приведен полный код `WelcomeFragment`; обновите файл *WelcomeFragment.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import androidx.navigation.findNavController

class WelcomeFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        return val view = inflater.inflate(R.layout.fragment_welcome, container, false)
        val startButton = view.findViewById<Button>(R.id.start)

        startButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_welcomeFragment_to_messageFragment)
        }
        return view
    }
}
```

В приложении используются два дополнительных класса, которые необходимо импортировать.



Необходимо получить ссылку на корневое представление фрагмента, чтобы вызвать его метод findViewById().

При щелчке на кнопке происходит переход к MessageFragment.

Возвращает корневое представление фрагмента.

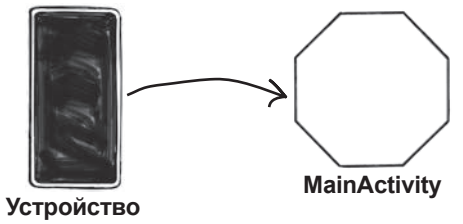
Вот и весь код, необходимый для того, чтобы фрагмент `WelcomeFragment` переходил к `MessageFragment`. Рассмотрим, что происходит при выполнении этого кода, а затем проведем тест-драйв приложения.

WelcomeFragment
MessageFragment

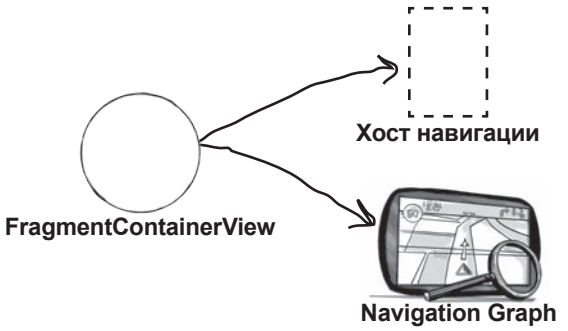
Что происходит при выполнении приложения

Ниже описана последовательность событий, происходящих при выполнении приложения:

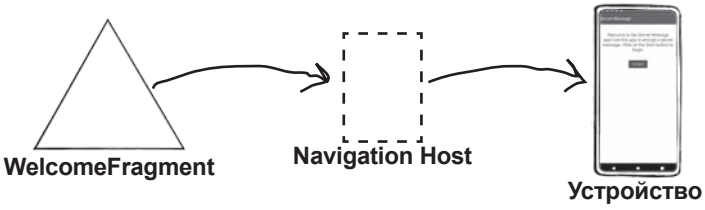
1. **Приложение запускается, и создается активность MainActivity.**



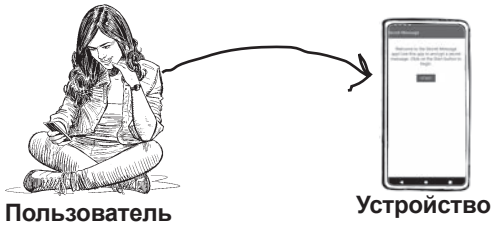
2. **Макет MainActivity activity_main.xml включает представление FragmentContainerView, которое задает хост и граф навигации.**



3. **Начальной целью в графе навигации указан WelcomeFragment, поэтому этот фрагмент добавляется в хост навигации и отображается на экране устройства.**



4. **Пользователь щелкает на кнопке Start в макете WelcomeFragment.**

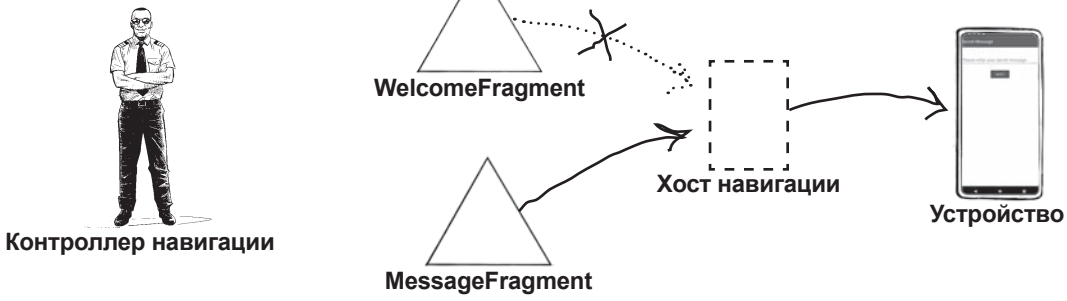


Продолжение истории

5 **Код слушателя OnClickListener кнопки Start находит контроллер навигации и вызывает его метод navigate().**
При вызове передается действие, используемое для перехода от `WelcomeFragment` к `MessageFragment`.

6 **Контроллер навигации ищет действие с заданным идентификатором в графе навигации.**
Он видит, что это действие переходит от `WelcomeFragment` к `MessageFragment`.

7 **Контроллер навигации заменяет WelcomeFragment на MessageFragment в хосте навигации, и на экране устройства отображается MessageFragment.**



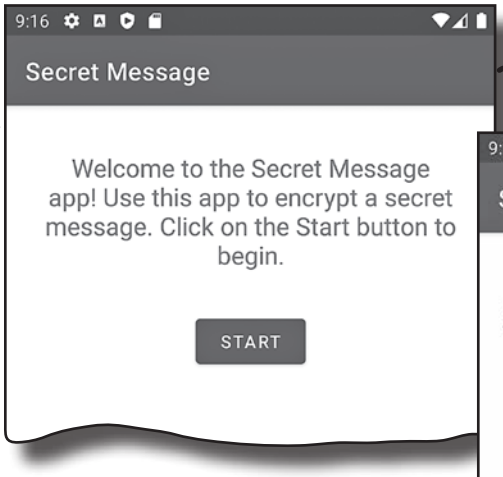
Тест-драйв

WelcomeFragment
MessageFragment

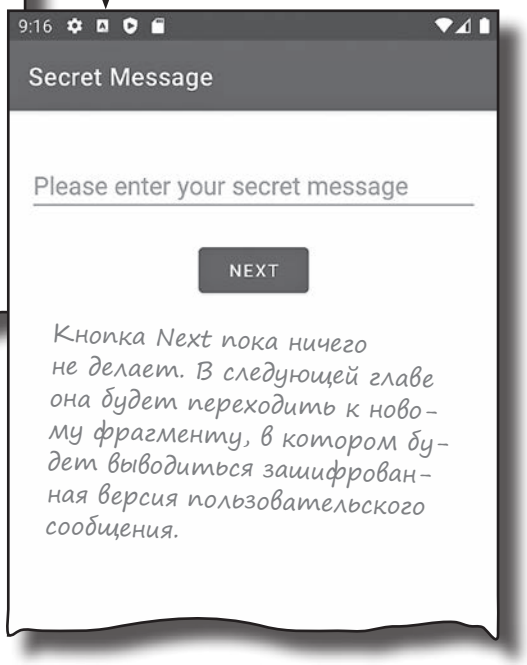
При запуске приложения начинает работать активность `MainActivity` и отображается фрагмент `WelcomeFragment` — все как прежде.

При щелчке на кнопке Start приложение переходит к `MessageFragment`, и этот фрагмент отображается на устройстве.

WelcomeFragment отображается при запуске приложения.



MessageFragment отображается при щелчке на кнопке Start фрагмента WelcomeFragment.



Поздравляем! Вы научились строить многоэкранные приложения с возможностью перехода между экранами.

Эта информация будет взята за основу в следующей главе, в которой мы завершим построение приложения Secret Message.

Часто Задаваемые Вопросы

В: Я попытался создать граф навигации перед редактированием файлов *build.gradle*, и среда Android Studio предложила добавить библиотеку компонента Navigation. Почему мы добавляем библиотеки вручную?

О: Чтобы вы видели, что библиотеки необходимы, и умели включать их без помощи Android Studio.

Есть и другая причина: мы знаем, что код в книге работает с версией библиотеки, указанной в файлах *build.gradle*. Android Studio может включать другую версию, которая нарушит работоспособность кода.

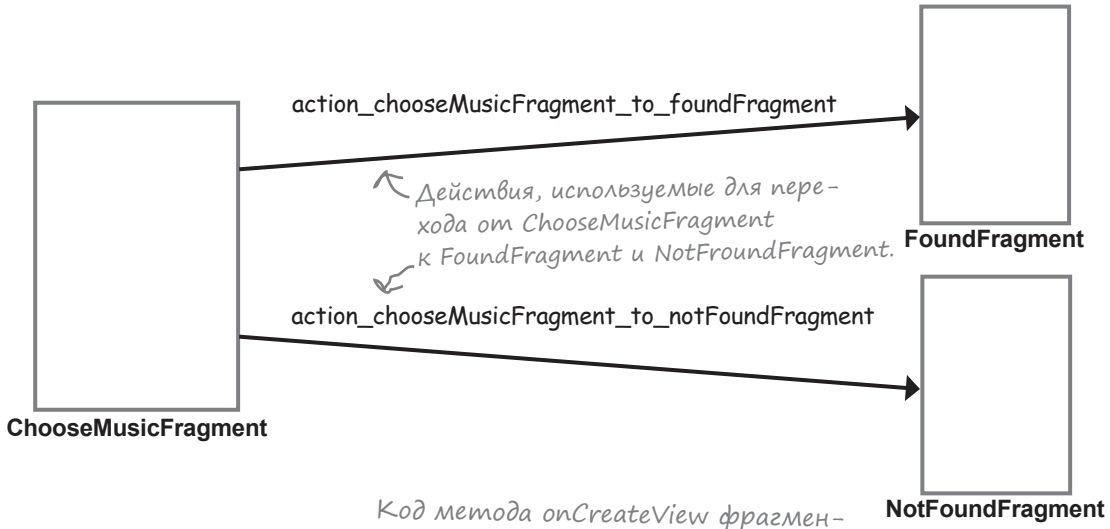
В: Обязательно ли использовать компонент Navigation для реализации навигации?

О: Строго говоря, нет, но без него все намного сложнее. До появления компонента Navigation навигация между экранами требовала *намного большего* объема кода.

Возьмите в руку карандаш

На следующей диаграмме показана часть графа навигации для музыкального приложения. В нем описаны два пути навигации: один переходит от `ChooseMusicFragment` к `FoundFragment`, а другой от `ChooseMusicFragment` к `NotFoundFragment`.

Завершите метод `onCreateView()`: если метод `musicFound()` возвращает *true*, приложение отображает `FoundFragment`, а если метод возвращает *false*, приложение вместо этого отображает `NotFoundFragment`.



Код метода onCreateView фрагмента ChooseMusicFragment.

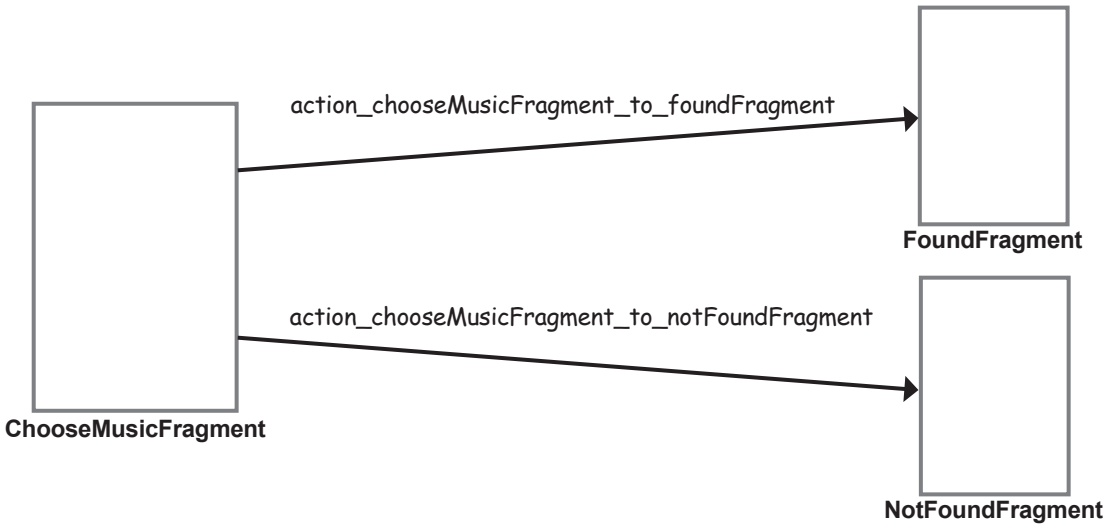
```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(R.layout.fragment_choose_music, container, false)
    val searchButton = view.findViewById<Button>(R.id.search)
    searchButton.setOnClickListener {
        if (musicFound()) {
            ..........
            ..........
        } else {
            ..........
            ..........
        }
    }
    return view
}
```

Возьмите в руку карандаш

Решение

На следующей диаграмме показана часть графа навигации для музыкального приложения. В нем описаны два пути навигации: один переходит от `ChooseMusicFragment` к `FoundFragment`, а другой от `ChooseMusicFragment` к `NotFoundFragment`.

Завершите метод `onCreateView()`: если метод `musicFound()` возвращает *true*, приложение отображает `FoundFragment`, а если метод возвращает *false*, приложение вместо этого отображает `NotFoundFragment`.

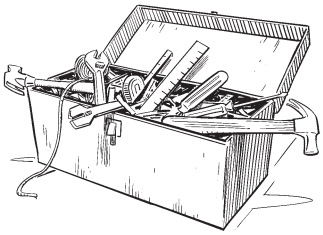



```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(R.layout.fragment_choose_music, container, false)
    val searchButton = view.findViewById<Button>(R.id.search)
    searchButton.setOnClickListener {
        if (musicFound()) {
            view.findNavController()
                .navigate(R.id.action_chooseMusicFragment_to_foundFragment)
        } else {
            view.findNavController()
                .navigate(R.id.action_chooseMusicFragment_to_notFoundFragment)
        }
    }
    return view
}
```

Получение ссылки на контроллер навигации.

Вызываем метод navigate с передачей идентификатора действия, описывающего путь навигации.

Ваш инструментарий Android



Глава 6 осталась позади, а ваш инструментарий пополнился фрагментами и навигацией.

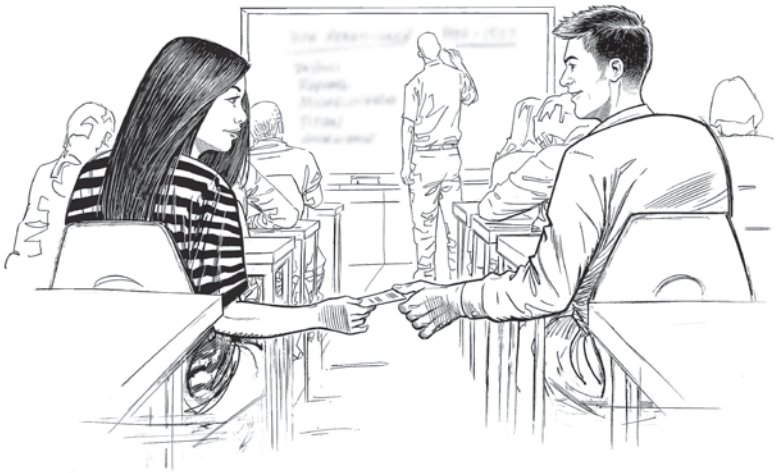
Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Фрагмент содержит код Kotlin и макет.
- Каждый фрагмент расширяет класс `Fragment` или один из его субклассов.
- Для добавления фрагментов в макет активности используется представление `FragmentContainerView`.
- Метод `onCreateView()` вызывается каждый раз, когда Android требуется макет фрагмента.
- Класс class не расширяет `Activity`.
- Фрагменты не содержат метод `findViewById()`.
- Компонент Navigation входит в семейство библиотек, плагинов и модулей, которые добавляются в проект при помощи Gradle.
- Граф навигации описывает возможные точки перехода и пути навигации. Для описания этих путей используются действия.
- Хост навигации представляет собой пустой контейнер, используемый для отображения фрагмента, к которому вы переходите. Компонент Navigation включает хост навигации по умолчанию с именем `NavHostFragment`, который расширяет класс `Fragment` и реализует интерфейс `NavHost`.
- Контроллер навигации использует действия для управления тем, какой фрагмент должен отображаться в хосте навигации.

7. Плагин safe args

Передача информации



Иногда фрагментам для нормальной работы требуется дополнительная информация. Например, если во фрагменте выводятся контактные данные, этот фрагмент должен знать, какой контакт он должен выбрать. Но что, если эта информация **должна поступать из другого фрагмента**? В этой главе мы **воспользуемся вашими знаниями о навигации** и применим их **для передачи данных между фрагментами**. Вы узнаете, **как добавить аргументы** к целям навигации, чтобы они могли получать необходимую информацию. Вы познакомитесь с плагином **Safe Args** и научитесь использовать его **для написания кода, безопасного по отношению к типам**. Наконец, вы научитесь **работать со стеком возврата** и управлять поведением кнопки Back. Читайте дальше — отступать уже поздно.

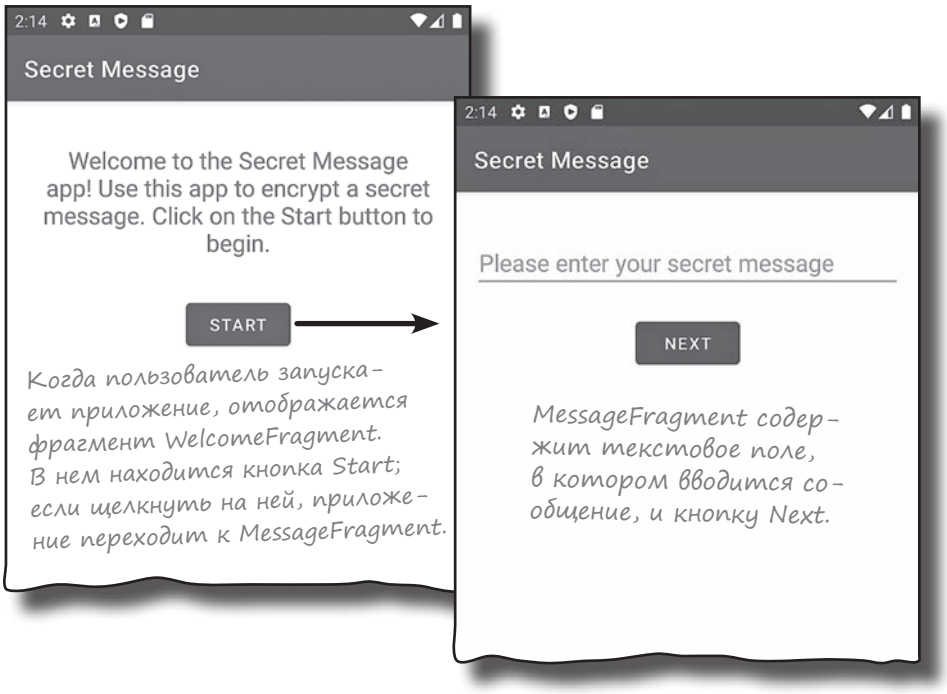
Приложение Secret Message переходит между фрагментами

В предыдущей главе вы научились использовать компонент Navigation для перемещения между двумя фрагментами. Эта информация использовалась для построения первой половины приложения Secret Message, которое должно получать от пользователя сообщение и шифровать его.

Текущая версия приложения использует фрагмент `WelcomeFragment` для вывода пояснительного текста. Когда пользователь щелкает на кнопке Start, фрагмент переходит ко второму фрагменту с именем `MessageFragment`.

`MessageFragment` включает текстовое поле, в котором пользователь вводит свое сообщение. Также в нем присутствует кнопка Next, на которой щелкает пользователь, чтобы зашифровать сообщение.

Напомним, как выглядит приложение в настоящее время:

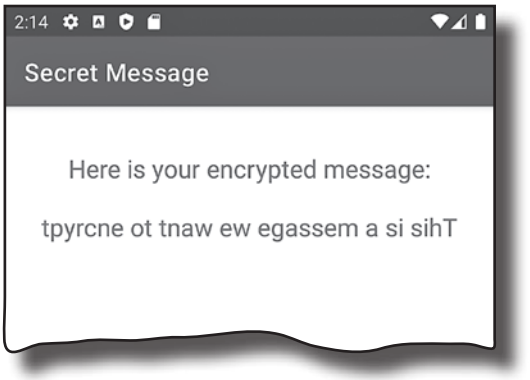


Когда пользователь щелкает на кнопке Next во фрагменте `MessageFragment`, в текущей версии приложения ничего не происходит. В этой главе мы завершим построение приложения, чтобы зашифрованное сообщение выводилось в новом фрагменте.

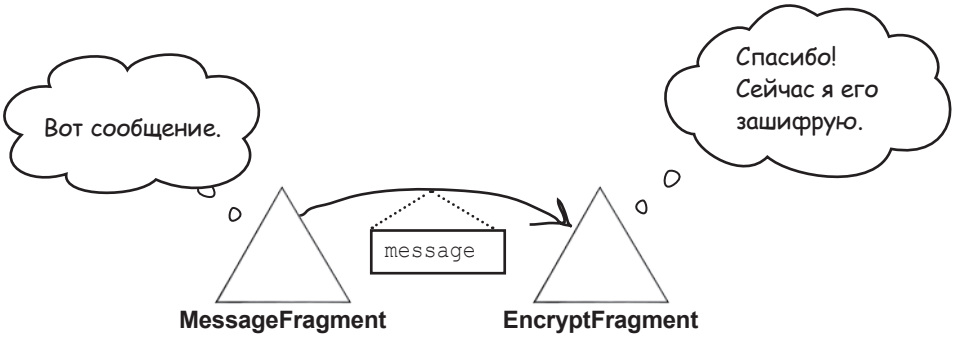
Фрагмент MessageFragment должен передать сообщение новому фрагменту

Мы добавим в приложение новый фрагмент с именем `EncryptFragment`. Этот фрагмент будет выводить зашифрованное сообщение, которое будет выглядеть примерно так:

Новый фрагмент (еще не созданный) с именем EncryptFragment. В нем выводится зашифрованная версия сообщения пользователя.



`EncryptFragment` получает текст сообщения от `MessageFragment`. Когда пользователь щелкает на кнопке Next фрагмента `MessageFragment`, приложение будет переходить к фрагменту `EncryptFragment` и передавать ему текст. Затем этот фрагмент шифрует текст и выводит результат:



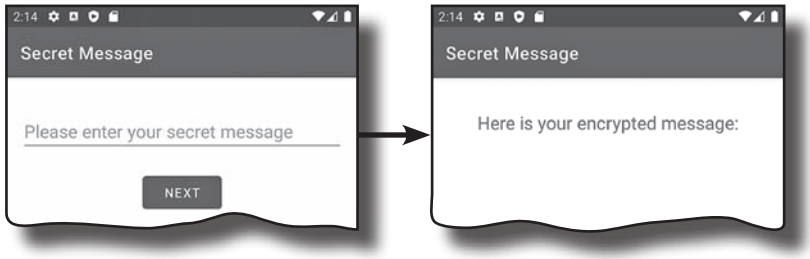
Чтобы выполнить эту работу, необходимо иметь возможность передавать данные между фрагментами. Для этого лучше всего воспользоваться плагином Gradle, который называется **Safe Args**, — дополнительной составляющей компонента Navigation. Он предоставляет механизм передачи данных между фрагментами способом, безопасным по отношению к типам. Тем самым предотвращается случайная передача данных неправильного типа, что может привести к ошибкам времени выполнения.

Вы больше узнаете об использовании Safe Args в ходе построения Secret Message. А пока рассмотрим действия, которые нам предстоит выполнить.

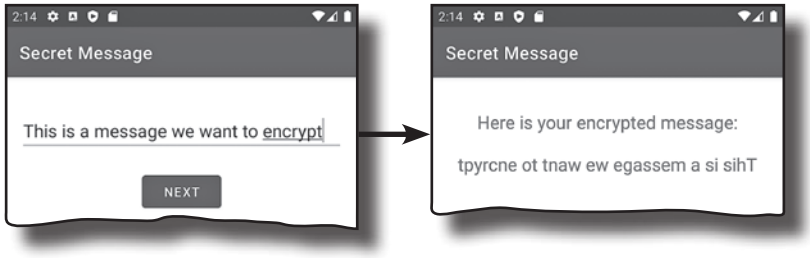
Что нам предстоит сделать

Действия, которые мы выполним для построения второй половины приложения Secret Message:

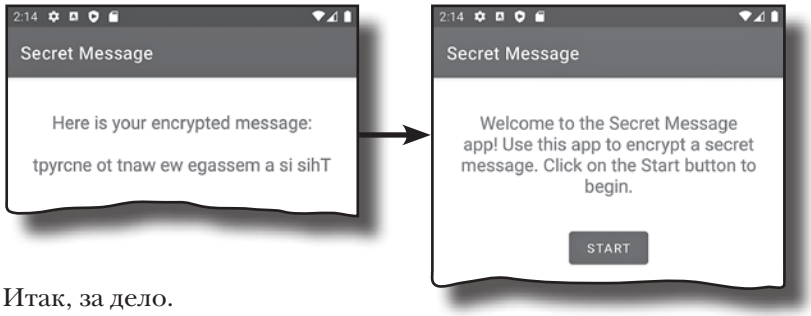
1. **Создание и отображение EncryptFragment.**
Мы создадим новый фрагмент с именем `EncryptFragment`, чтобы приложение переходило к нему при щелчке на кнопке Next в макете `MessageFragment`.



2. **Передача сообщения фрагменту EncryptFragment.**
Мы используем Safe Args для передачи пользовательского сообщения от `MessageFragment` к `EncryptFragment`. В `EncryptFragment` выводится зашифрованное сообщение.



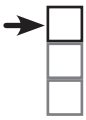
3. **Изменение поведения кнопки Back.**
Наконец, мы обновим приложение, чтобы при нажатии кнопки Back во время отображения фрагмента `EncryptFragment` приложение переходило к `WelcomeFragment`.



Не забудьте!

В этой главе мы изменим приложение Secret Message, созданное в главе 6. Не забудьте открыть проект этого приложения.

Итак, за дело.



EncryptFragment
Safe Args
Кнопка Back

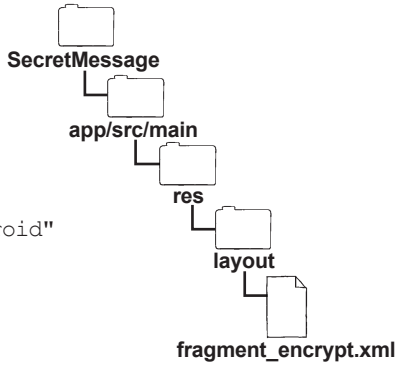
Создание фрагмента EncryptFragment...

Фрагмент `EncryptFragment` будет использоваться для вывода зашифрованной версии сообщения. Чтобы создать этот фрагмент, выделите пакет *com.hfad.secretmessage* в папке *app/src/main/java*, перейдите в меню File и выберите команду New→Fragment→Fragment (Blank). Введите имя фрагмента «EncryptFragment» и имя макета «fragment_encrypt». Убедитесь в том, что выбран язык Kotlin, а затем щелкните на кнопке Finish.

...и обновление его макета

Мы обновим макет `EncryptFragment` и добавим в него два текстовых представления. В первом будет выводиться строковый ресурс с именем `encrypt_text` (мы добавили его в файл *strings.xml* в предыдущей главе), а во втором — зашифрованное сообщение.

Откройте файл макета *fragment_encrypt.xml* и обновите его содержимое:




```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center_horizontal"
    tools:context=".EncryptFragment">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:textSize="20sp"
        android:text="@string/encrypt_text" />

    <TextView
        android:id="@+id/encrypted_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:textSize="20sp"/>
</LinearLayout>
```

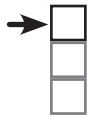
Определяем вертикальный макет LinearLayout.

Вывести описание...

Here is your encrypted message:

tpyrcne ot tnaw ew egassem a si sihT

....и сверхсекретное зашифрованное сообщение.

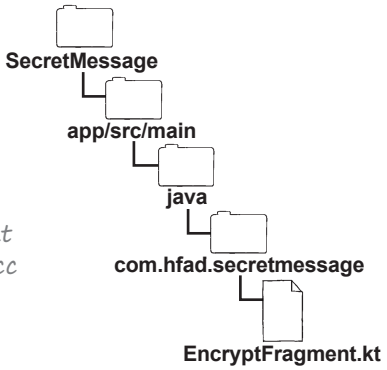


EncryptFragment
Safe Args
Кнопка Back

Обновление EncryptFragment.kt

Также необходимо обновить код Kotlin фрагмента `EncryptFragment` и убедиться в том, что среда Android Studio не добавила лишний код, из-за которого он не сможет делать то, что вам нужно.

Перейдите к пакету *com.hfad.secretmessage* в папке *app/src/main/java* и откройте файл *EncryptFragment.kt*. Замените код, сгенерированный Android Studio, следующим кодом:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class EncryptFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        // Заполнение макета для фрагмента
        return inflater.inflate(R.layout.fragment_encrypt, container, false)
    }
}
```

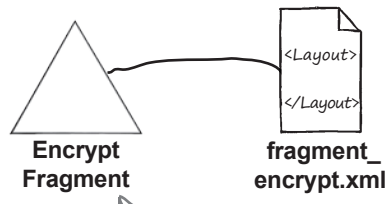
EncryptFragment расширяет класс Fragment.

Фрагмент переопределяет onCreateView().

Файл fragment_encrypt.xml используется для макета фрагмента.

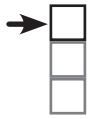
Приведенный выше код — все, что необходимо *EncryptFragment.kt* для определения базового фрагмента. Как и код остальных фрагментов, который вы видели, он расширяет класс `Fragment` и переопределяет его метод `onCreateView()`. Метод заполняет макет фрагмента и возвращает его корневое представление.

Мы написали всю разметку и код Kotlin, необходимые для `EncryptFragment`. Затем нужно позаботиться о том, чтобы фрагмент `MessageFragment` мог перейти к нему, — для этого его следует включить в граф навигации.



EncryptFragment использует разметку fragment_encrypt.xml в качестве макета.

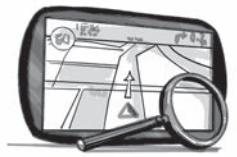
Включение EncryptFragment в граф навигации



EncryptFragment
Safe Args
Кнопка Back

Как вы уже знаете, граф навигации содержит подробную информацию о целях приложения и возможных путях, по которым можно перемещаться к ним.

Чтобы добавить `EncryptFragment` в граф навигации, откройте *nav_graph.xml*, щелкните на кнопке New Destination в визуальном редакторе и выберите вариант «fragment_encrypt». В результате фрагмент будет добавлен в приложение.



Граф навигации

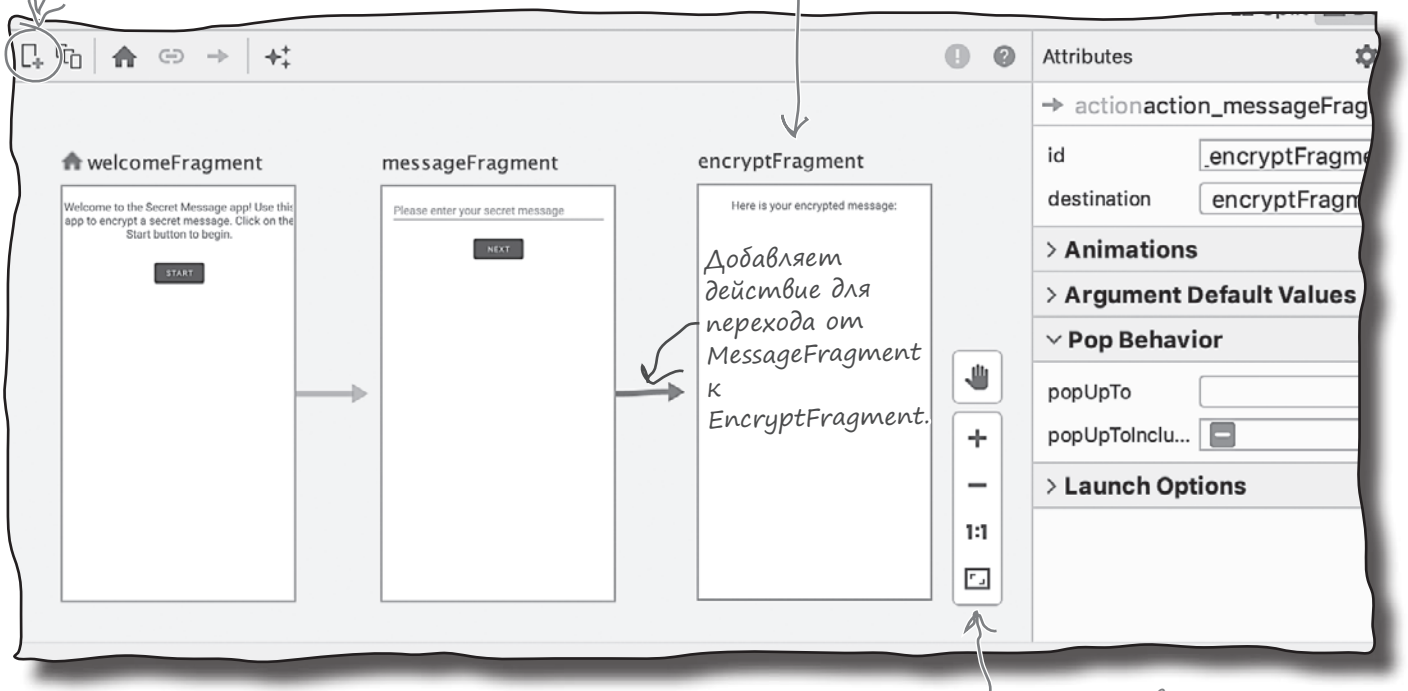
Также необходимо добавить новое действие, чтобы фрагмент `MessageFragment` мог перейти к `EncryptFragment`. Для этого наведите указатель мыши на `MessageFragment` и перетащите стрелку действия от правого края к `EncryptFragment`. Убедитесь, что действие имеет идентификатор «action_messageFragment_to_encryptFragment», что соответствует коду в этой книге.

Не беспокойтесь, если все это покажется вам слишком хлопотным. Полный код приведен на следующей странице, так что при желании вы сможете просто обновить его.

После внесения этих изменений граф навигации должен выглядеть примерно так:

Щелкните на этой кнопке, чтобы добавить новую цель.

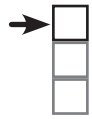
Добавьте EncryptFragment как новую цель.



Добавляет действие для перехода от MessageFragment к EncryptFragment.

При помощи этих инструментов можно изменять размеры или перемещать цели.

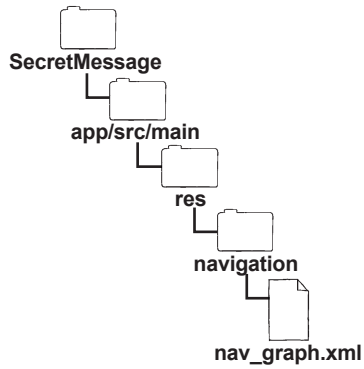
Давайте посмотрим, как выглядит разметка XML.



EncryptFragment
Safe Args
Кнопка Back

Обновленный код nav_graph.xml

Каждый раз, когда вы обновляете граф навигации, все изменения добавляются в разметку XML. Новый код выглядит так (изменения выделены жирным шрифтом):

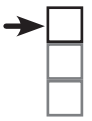



```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>

    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" >
        <action
            android:id="@+id/action_messageFragment_to_encryptFragment"
            app:destination="@id/encryptFragment" />
    </fragment>

    <fragment
        android:id="@+id/encryptFragment"
        android:name="com.hfad.secretmessage.EncryptFragment"
        android:label="fragment_encrypt"
        tools:layout="@layout/fragment_encrypt" />
</navigation>
```

Новое действие для перехода от MessageFragment к EncryptFragment.

Добавляем EncryptFragment.

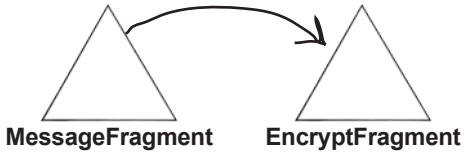
Новое действие будет использоваться для перехода к `EncryptFragment`.



EncryptFragment
Safe Args
Кнопка Back

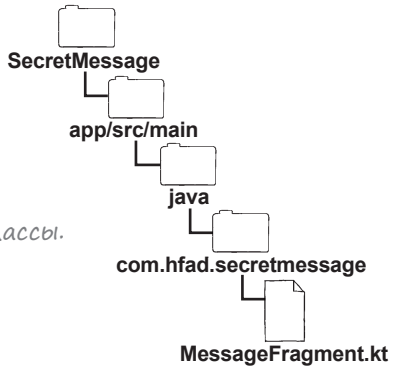
Фрагмент MessageFragment должен переходить к EncryptFragment

Необходимо сделать так, чтобы при щелчке на кнопке Next фрагмент `MessageFragment` переходил к `EncryptFragment`. Для этого мы добавим к кнопке слушатель `OnClickListener`, который будет включать код навигации.



Как вы узнали в предыдущей главе, чтобы перейти от одного фрагмента к другому, следует получить контроллер навигации и передать ему действие навигации. Контроллер навигации использует это действие для отображения правильного фрагмента.

Необходимый для этого код вам уже знаком. Обновите файл *MessageFragment.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import androidx.navigation.findNavController

class MessageFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        return val view = inflater.inflate(R.layout.fragment_message, container, false)
        val nextButton = view.findViewById<Button>(R.id.next)

        nextButton.setOnClickListener {
            view.findNavController()
                    .navigate(R.id.action_messageFragment_to_encryptFragment)
        }
        return view
    }
}
```

Импортируйте эти классы.

Получение ссылки на корневое представление.

По кнопке Next должен происходить переход к EncryptFragment.

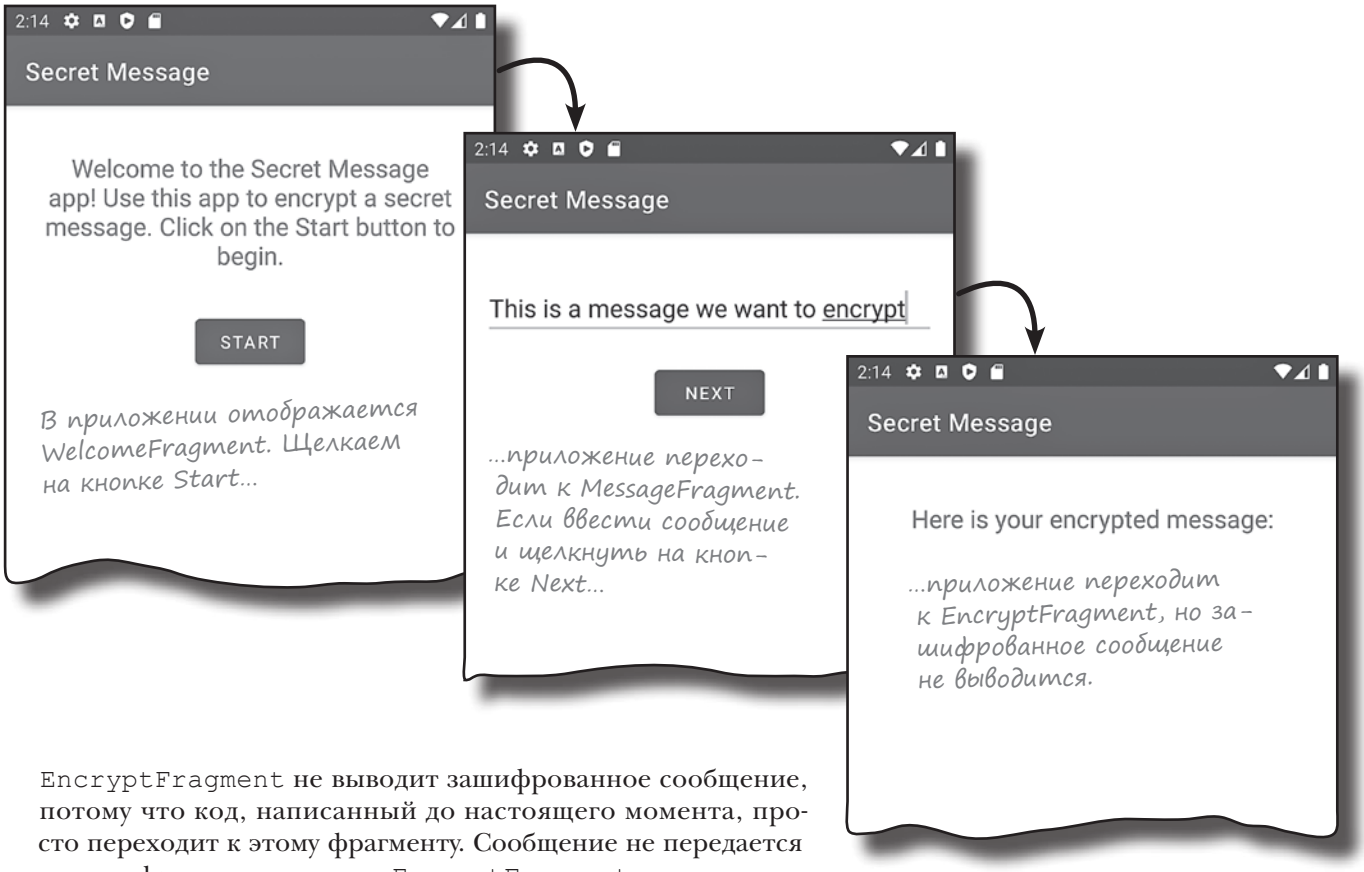
Давайте проведем тест-драйв приложения и убедимся в том, что оно работает.

Тест-драйв

- [x] **EncryptFragment**
- [ ] **Safe Args**
- [ ] **Кнопка Back**

При запуске приложения запускается активность `MainActivity` и отображается фрагмент `WelcomeFragment`. При щелчке на кнопке Start приложение, как и прежде, переходит к `MessageFragment`.

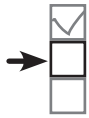
Когда вы вводите сообщение в текстовом поле `MessageFragment` и щелкаете на кнопке Next, приложение переходит к `EncryptFragment`. С введенным сообщением при этом ничего не происходит.



`EncryptFragment` не выводит зашифрованное сообщение, потому что код, написанный до настоящего момента, просто переходит к этому фрагменту. Сообщение не передается новому фрагменту, поэтому `EncryptFragment` с ним ничего сделать не может.

Для передачи сообщения от `MessageFragment` к `EncryptFragment` можно воспользоваться Safe Args. Как упоминалось ранее, это дополнительная часть компонента Navigation, которая позволяет передавать аргументы целям способом, безопасным по отношению к типам.

Давайте выясним, как это происходит.



EncryptFragment
Safe Args
Кнопка Back

Добавление Safe Args в файлы build.gradle

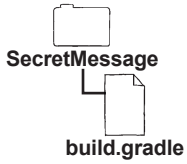
Прежде чем начинать использовать Safe Args, необходимо обновить файлы *build.gradle* проекта и приложения. Давайте сделаем это.

Добавление пути к классам в файл build.gradle проекта

Сначала необходимо добавить новый путь к классам в файл *build.gradle* проекта, указав, что вы хотите использовать плагин Safe Args. Путь к классам включает номер версии, который должен соответствовать используемому в библиотеке главного компонента Navigation.

В приложении Secret Message для проверки соответствия номеров версий будет использоваться переменная `nav_version`, поэтому ее необходимо включить в путь к классам.

Откройте файл *SecretMessage/build.gradle* и добавьте путь к классам (выделен жирным шрифтом) в раздел `dependencies`:

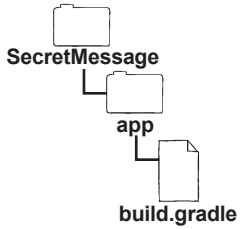



```
dependencies {
    ...
    classpath "androidx.navigation:navigation-safe-args-gradle-plug-in:$nav_version"
}
```

В предыдущей главе мы включили переменную nav_version для проверки того, что во всех частях компонента Navigation используется один номер версии.

Добавление плагина в файл build.gradle приложения

Затем необходимо сообщить Gradle, что вы используете плагин Safe Args; для этого добавьте строку в файл *build.gradle*. Откройте файл *SecretMessage/app/build.gradle* и добавьте строку, выделенную жирным шрифтом, в раздел `plug-ins`:




```
plug-ins {
    ...
    id 'androidx.navigation.safeargs.kotlin'
}
```

Добавляет плагин Safe Args.

После внесения этих изменений щелкните на ссылке Sync Now, появившейся в верхней части редактора кода. Тем самым вы синхронизируете внесенные изменения с оставшейся частью проекта.

Помните, что изменения должны синхронизироваться при каждом изменении файла build.gradle.

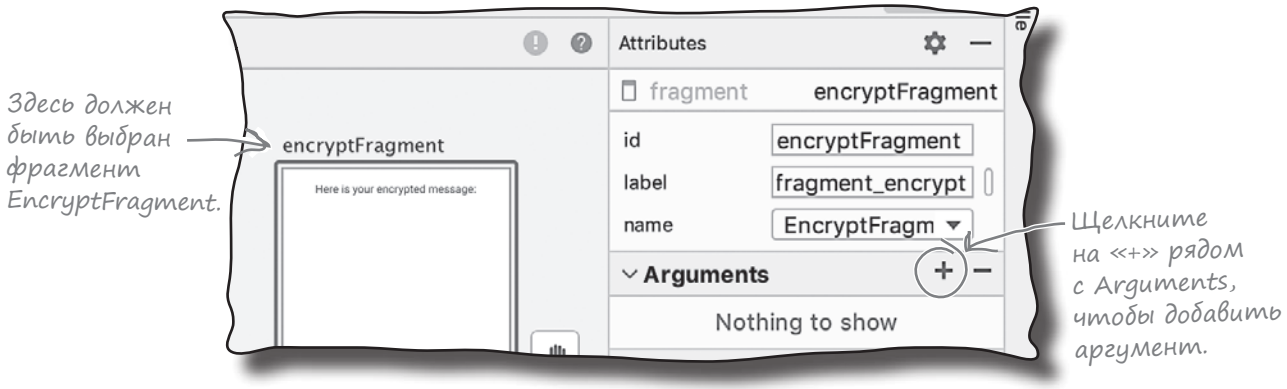
После включения плагина Safe Args в приложение можно воспользоваться им для передачи сообщения от `MessageFragment` к `EncryptFragment`. Мы сделаем это на следующем шаге.

Фрагмент EncryptFragment должен получать строковый аргумент

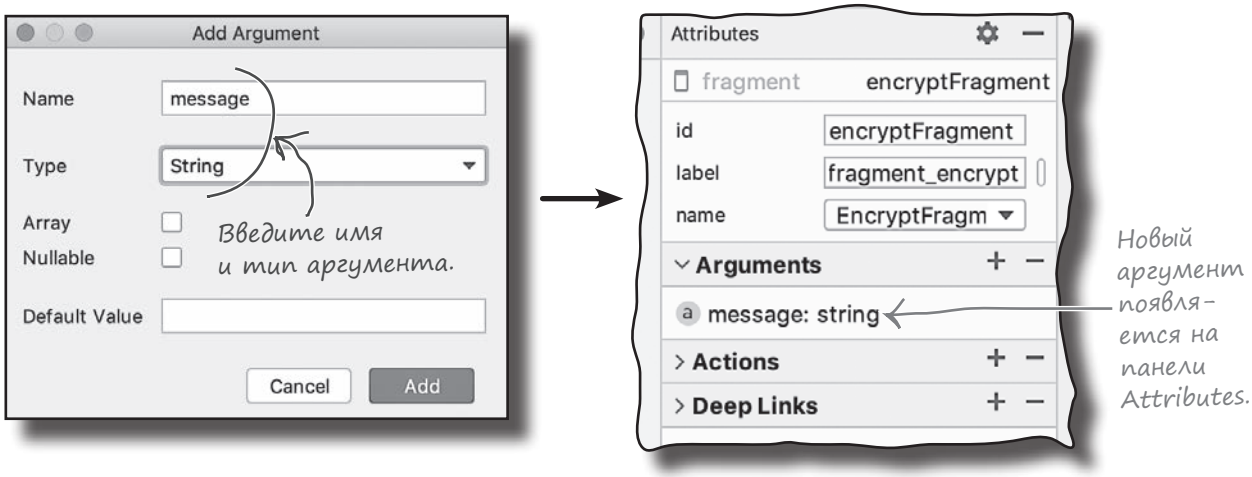
Прежде всего необходимо указать, что фрагмент `EncryptFragment` может получать сообщения пользователя. Для этого мы добавим строковый аргумент в этот фрагмент на графе навигации; `MessageFragment` использует эти аргументы для передачи сообщения (строки) фрагменту `EncryptFragment`.

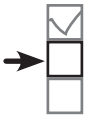

Граф навигации

Чтобы добавить аргумент, откройте файл *nav_graph.xml* из *app/src/main/res/nav_graph.xml*. Выберите `EncryptFragment` в визуальном редакторе графа навигации, перейдите на панель Attributes и щелкните на кнопке «+» рядом с разделом Arguments:



Когда вы щелкаете на кнопке «+», появляется окно Add Argument, в котором вводится информация об аргументе. В данном случае фрагмент `EncryptFragment` должен получать аргумент `String` с сообщением, поэтому присвойте аргументу имя «message», выберите тип «String» и щелкните на кнопке Add. Новый фрагмент создается и добавляется в раздел Arguments панели Attributes:

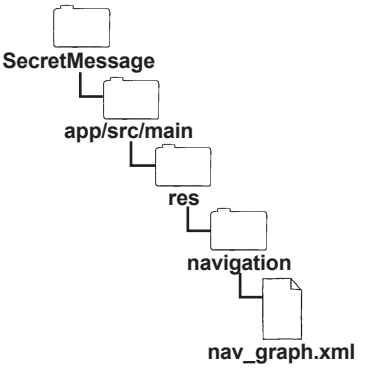




EncryptFragment
Safe Args
Кнопка Back

Обновленный код nav_graph.xml

Когда вы добавляете аргумент в графе навигации, в его разметке XML появляется новый элемент <argument>. Ниже приведен обновленный код *nav_graph.xml* (изменения выделены жирным шрифтом):




```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>
    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" >
        <action
            android:id="@+id/action_messageFragment_to_encryptFragment"
            app:destination="@id/encryptFragment" />
    </fragment>
    <fragment
        android:id="@+id/encryptFragment"
        android:name="com.hfad.secretmessage.EncryptFragment"
        android:label="fragment_encrypt"
        tools:layout="@layout/fragment_encrypt" >
        <argument
            android:name="message"
            app:argType="string" />
    </fragment>
</navigation>
```

Новый аргумент: строка с именем «message».

EncryptFragment
Safe Args
Кнопка Back

Фрагмент MessageFragment должен передать сообщение EncryptFragment

После того как мы добавили аргумент `String` в `EncryptFragment`, `MessageFragment` может использовать его для передачи пользовательского сообщения при переходе к этому фрагменту.

Как вы уже знаете, для перехода между целями контроллеру навигации передается действие навигации. Контроллер навигации использует это действие для вывода правильного фрагмента. Например, `MessageFragment` использует следующий код для перехода к `EncryptFragment` при щелчке на кнопке:

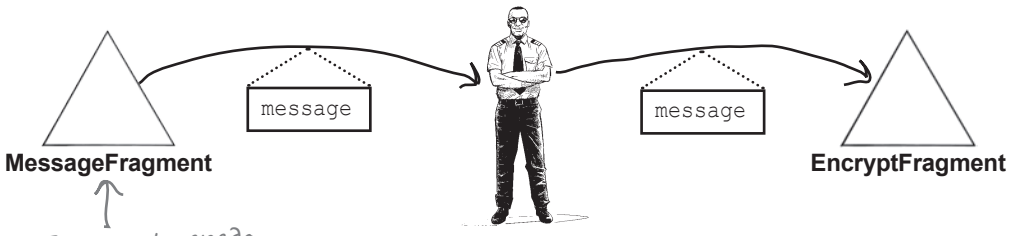
```
nextButton.setOnClickListener {
    view.findNavController()
            .navigate(R.id.action_messageFragment_to_encryptFragment)
}
```

Передает действие контроллеру навигации, который использует его для отображения правильного фрагмента.

Действиям навигации можно передавать аргументы

Если вы хотите передать аргумент цели, это можно сделать, передав его значение действию навигации.

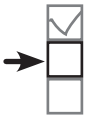
Когда контроллер навигации получает действие, включающее аргумент, он переходит к соответствующему фрагменту и передает значение аргумента. Например, в приложении Secret Message можно заставить `MessageFragment` передать сообщение пользователя `EncryptMessage` (через контроллер навигации), включив его в действие навигации. Это происходит примерно так:



MessageFragment передает сообщение контроллеру навигации...

...а контроллер навигации передает сообщение EncryptFragment.

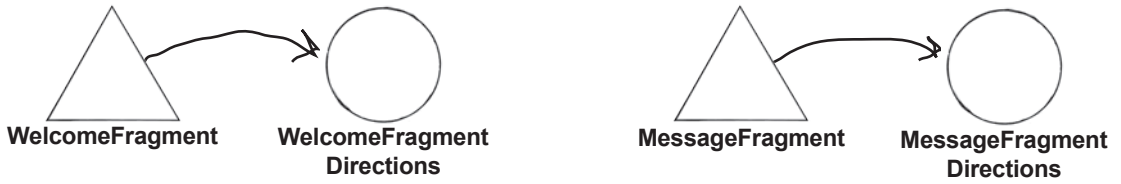
Для добавления аргументов к сообщениям навигации можно воспользоваться классом **`Directions`**. Давайте разберемся, что он собой представляет.



EncryptFragment
Safe Args
Кнопка Back

Safe Args генерирует классы Directions

Классы `Directions` используются для передачи аргументов целям. Когда вы подключаете плагин Safe Args, среда Android Studio использует его для генерирования класса `Directions` **для каждого фрагмента, от которого может выполняться переход**. Так, в приложении Secret Message можно переходить от фрагментов `WelcomeFragment` и `MessageFragment` к другим целям, поэтому плагин Safe Args генерирует класс `Directions` для каждого из этих фрагментов; для `WelcomeFragment` генерируется класс с именем `WelcomeFragmentDirections`, а для `MessageFragment` — класс с именем `MessageFragmentDirections`:

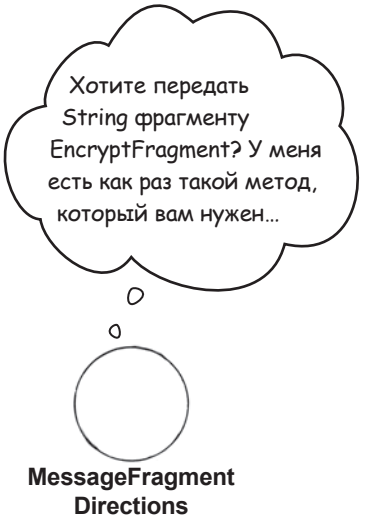


Каждый фрагмент использует собственный класс `Directions` для навигации. Например, фрагмент `MessageFragment` должен использовать класс `MessageFragmentDirections`, если он хочет перейти к другой цели с передачей аргумента.

Использование класса Directions для добавления аргументов в действия

Каждый класс `Directions` включает сгенерированный метод для каждого из действий фрагмента. Например, у `MessageFragment` имеется действие с идентификатором «action_messageFragment_to_encryptFragment», поэтому класс `MessageFragmentDirections` включает соответствующий метод с именем `actionMessageFragmentToEncryptFragment()`. А поскольку `EncryptFragment` получает `String`, сгенерированный метод включает аргумент `String`.

Генерируемые методы используются для перехода к целям. Например, чтобы перейти от `MessageFragment` к `EncryptFragment` и передать строковое сообщение, следует добавить в `MessageFragment` следующий код:




```
val action = MessageFragmentDirections
                .actionMessageFragmentToEncryptFragment(message)
view.findNavController().navigate(action)
```

MessageFragmentDirections используется для создания действия.

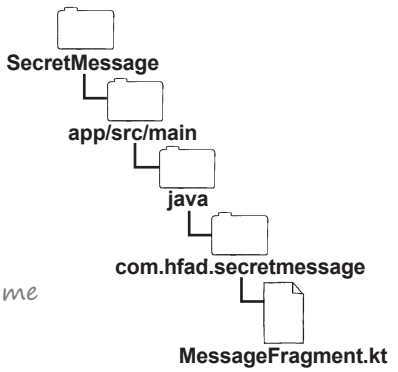
К действию добавляется строка сообщения message.

Давайте добавим этот код.

Обновление кода MessageFragment.kt

EncryptFragment
Safe Args
Кнопка Back

Ниже приведен обновленный код `MessageFragment`; не забудьте включить изменения (выделенные жирным шрифтом) в *MessageFragment.kt*:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import android.widget.EditText
import androidx.navigation.findNavController

class MessageFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_message, container, false)
        val nextButton = view.findViewById<Button>(R.id.next)
        val messageView = view.findViewById<EditText>(R.id.message)

        nextButton.setOnClickListener {
            val message = messageView.text.toString()
            val action = MessageFragmentDirections
                    .actionMessageFragmentToEncryptFragment(message)
            view.findNavController().navigate(action)
                .navigate(R.id.action_messageFragment_to_encryptFragment)
        }
        return view
    }
}
```

Импортируйте этот класс.

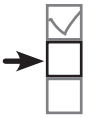
Получение ссылки на текстовое поле с сообщением.

Получение строки сообщения message.

Переход к EncryptMessage с передачей сообщения.

Удалите эту строку.

Это все, что необходимо `MessageFragment` для передачи пользовательского сообщения `EncryptFragment`. Затем фрагмент `EncryptFragment` должен получить сообщение и использовать его.



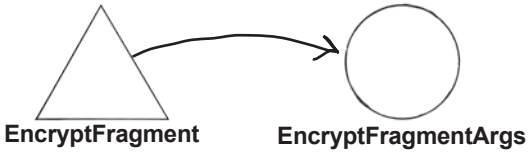
EncryptFragment
Safe Args
Кнопка Back

Фрагмент EncryptFragment должен получить значение аргумента

Как вы знаете, `MessageFragment` использует аргумент `String` для передачи сообщения фрагменту `EncryptFragment`. `EncryptFragment` должен получить это значение, чтобы вывести его зашифрованную версию.

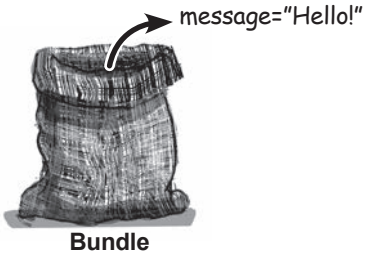
Для получения аргументов фрагменты могут использовать класс **`Args`**. Когда вы подключаете плагин Safe Args, Android Studio использует его для генерирования класса `Args` **для каждого фрагмента, получающего аргументы.** Например, в приложении Secret Message `EncryptFragment` получает аргумент `String`, поэтому плагин Safe Args генерирует для него класс `Args` с именем `EncryptFragmentArgs`:

Плагин Safe Args генерирует классы Directions и Args. Класс Directions используется для передачи аргументов цели, а класс Args — для их получения.



Использование класса Args для получения аргументов

Каждый класс `Args` включает метод **`fromBundle()`**, который используется для получения аргументов, переданных фрагменту. Например, в приложении Secret Message фрагмент `EncryptFragment` получает аргумент `String` с именем `message`, поэтому значение этого аргумента может быть получено следующим образом:




```
val message = EncryptFragmentArgs.fromBundle(requireArguments()).message
```

Получает значение аргумента message.

Эта команда присваивает объект `String` (значение аргумента) переменной `message`.

Сообщение необходимо зашифровать

Теперь вы знаете, как `EncryptFragment` получает переданное сообщение, и мы можем зашифровать его и вывести результат.

Для шифрования будет использован метод Kotlin `reversed()`, который просто переставляет буквы `String` в обратном порядке. Код выглядит так:

В реальном мире применяются намного более мощные средства шифрования. Это всего лишь пример.

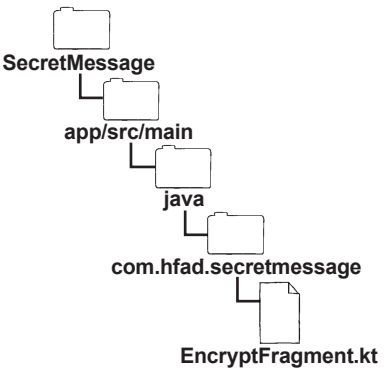
```
val encryptedView = view.findViewById<TextView>(R.id.encrypted_message)
encryptedView.text = message.reversed()
```

Полный код `EncryptFragment` приведен на следующей странице.

Полный код EncryptFragment.kt

EncryptFragment
Safe Args
Кнопка Back

Ниже приведен полный код файла *EncryptFragment.kt*; обновите его и включите изменения, выделенные жирным шрифтом:




```
package com.hfad.secretmessage

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView

class EncryptFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        return val view = inflater.inflate(R.layout.fragment_encrypt, container, false)
        val message = EncryptFragmentArgs.fromBundle(requireArguments()).message
        val encryptedView = view.findViewById<TextView>(R.id.encrypted_message)
        encryptedView.text = message.reversed()
        return view
    }
}
```

Импортируйте этот класс.

Получение корневого представления.

Получение сообщения.

Буквы переставляются в обратном порядке, а полученное сообщение выводится.

Вот и все, что необходимо сделать для вывода зашифрованного сообщения. На следующей странице мы разберемся, что же происходит при выполнении приложения.

Часто Задаваемые Вопросы

В: Мой проект не распознает классы `Directions` и `Args`. Почему?

О: Похоже, плагин Safe Args их не сгенерировал.

Сначала убедитесь в том, что вы добавили плагин Safe Args в файл build.gradle приложения и синхронизировали изменения. Если плагин отсутствует, Android Studio не сможет сгенерировать классы `Directions` и `Args`.

Затем убедитесь в том, что аргумент `EncryptFragment` был добавлен в граф навигации. Safe Args генерирует класс `Args` для каждой цели, получающей аргументы, и если аргумент отсутствует, то класс `Args` генерироваться не будет.

Если проблемы все еще остаются, попробуйте перестроить проект — откройте меню Build и выберите команду Rebuild Project.

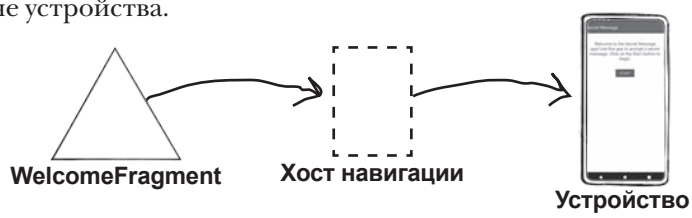
В: Вы сказали, что Safe Args добавляется в файлы *build.gradle* как плагин, тогда как остальные части компонента Navigation включаются как зависимость. Почему?

О: Safe Args добавляется как плагин, потому что он должен иметь возможность генерировать классы `Directions` и `Args`. Плагины способны генерировать код, а зависимости этого делать не могут.

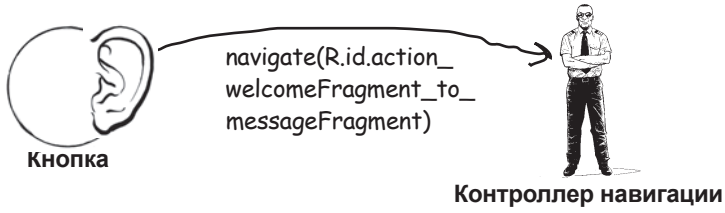
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

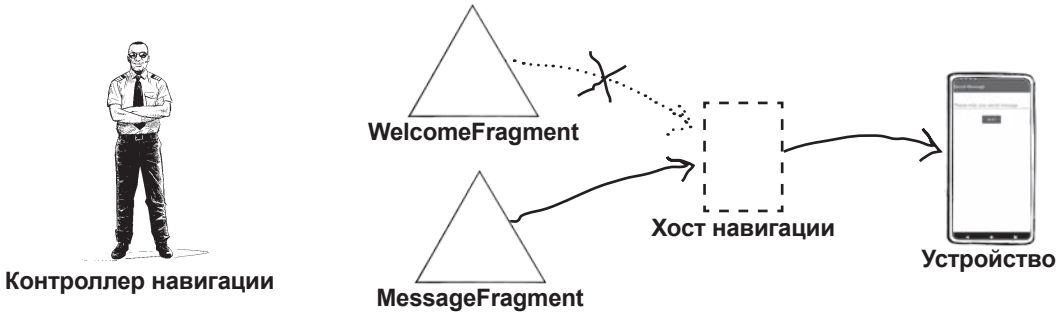
1. **Приложение запускается, и создается активность MainActivity.**
Фрагмент `WelcomeFragment` добавляется в хост навигации и отображается на экране устройства.



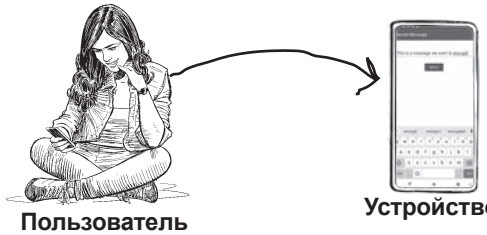
2. **Когда пользователь щелкает на кнопке Start, ее код OnClickListener передает действие методу navigate() контроллера навигации.**
Действие описывает путь перехода от `WelcomeFragment` к `MessageFragment`.

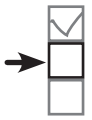


3. **Контроллер навигации помещает MessageFragment в хост навигации, чтобы он отображался на экране устройства.**



4. **Пользователь вводит сообщение и щелкает на кнопке Next.**

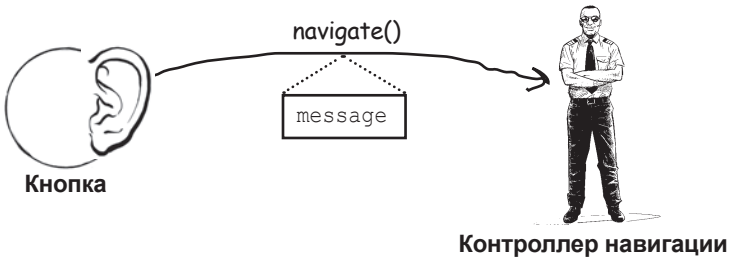




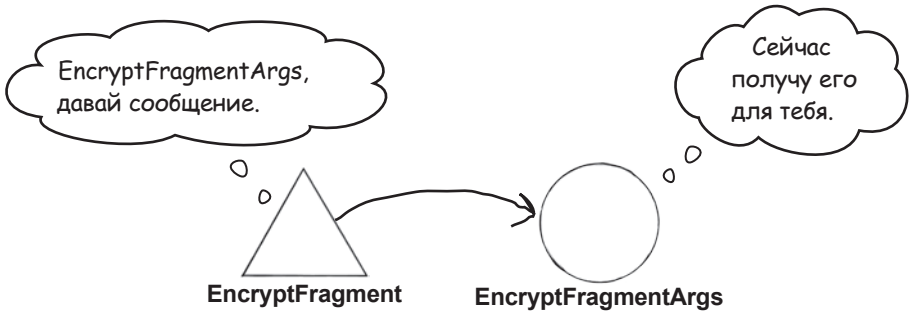
EncryptFragment
Safe Args
Кнопка Back

Продолжение истории

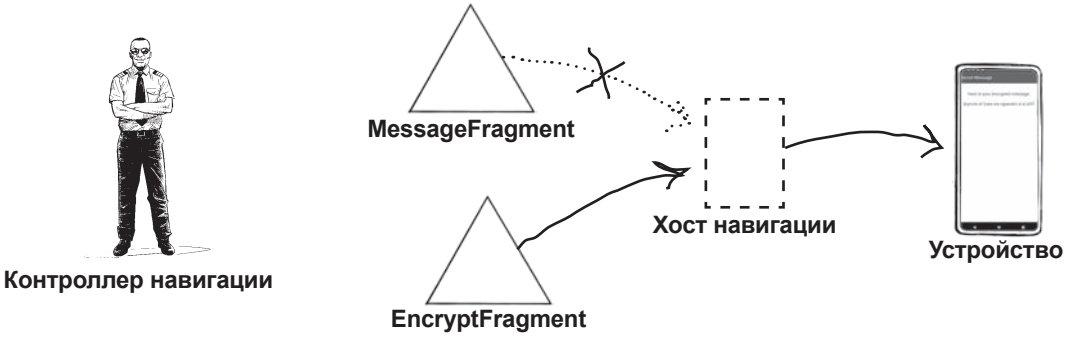
5 **Слушатель OnClickListener кнопки Next использует класс MessageFragmentDirections для присоединения сообщения к действию.**
Действие (вместе с сообщением) передается контроллеру навигации.



6 **Контроллер навигации передает сообщение EncryptFragment.**
`EncryptFragment` использует класс `EncryptFragmentArgs` для чтения его значения.



7 **Контроллер навигации заменяет MessageFragment фрагментом EncryptFragment в хосте навигации MainActivity.**
`EncryptFragment` отображается на экране устройства и выводит зашифрованное сообщение.



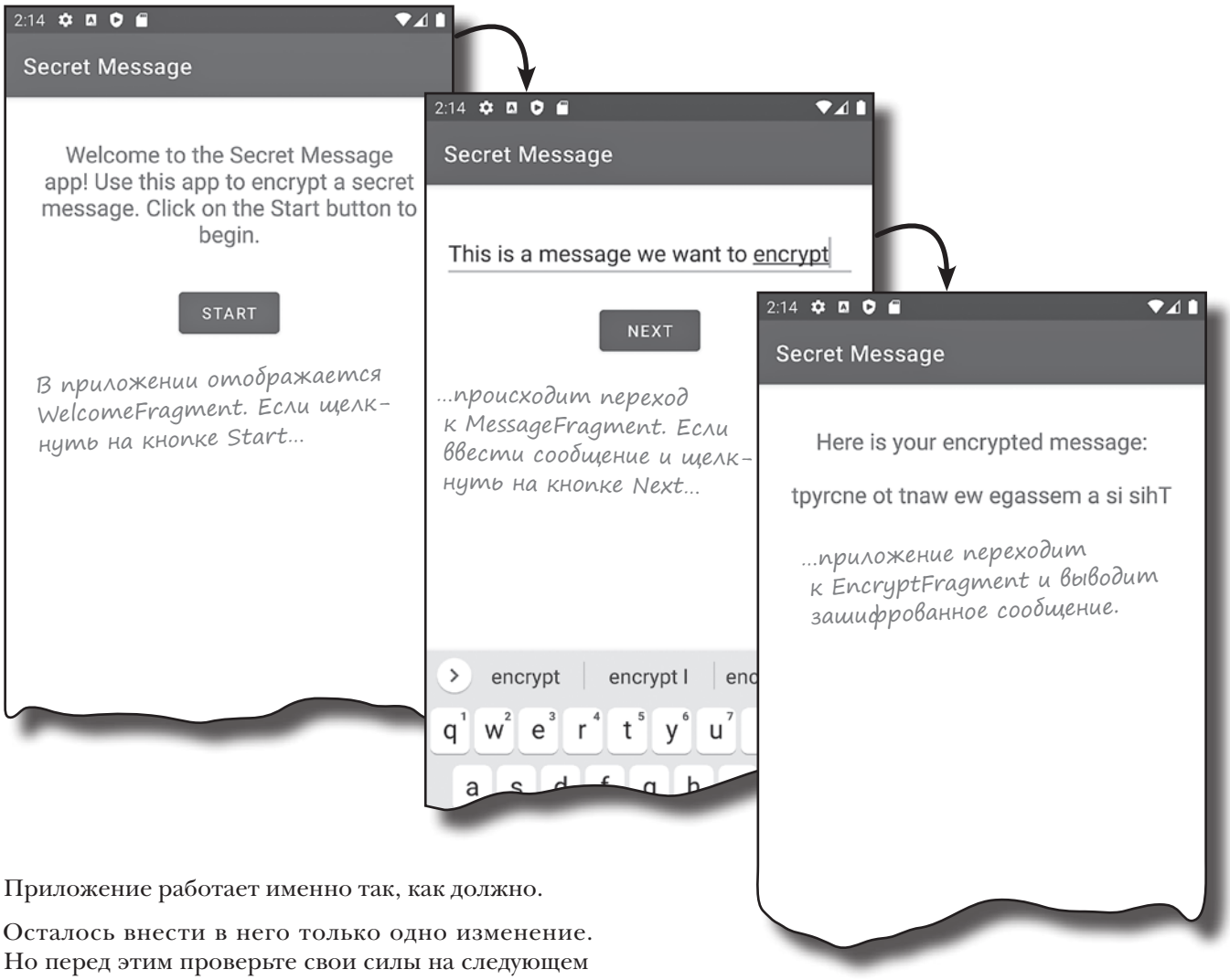
А теперь проведем тест-драйв приложения.

Тест-драйв

EncryptFragment
Safe Args
Кнопка Back

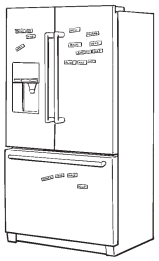
При запуске приложения отображается фрагмент `WelcomeFragment`, и вы можете перейти к `MessageFragment`, щелкнув на кнопке Start.

Когда вы вводите сообщение и щелкаете на кнопке Next в `MessageFragment`, отображается фрагмент `EncryptFragment` с зашифрованным сообщением.



Приложение работает именно так, как должно.

Осталось внести в него только одно изменение. Но перед этим проверьте свои силы на следующем упражнении.



Развлечения с магнитами

Приведенный ниже код графа навигации определяет путь навигации между двумя фрагментами: `ChooseTypeFragment` и `DrinksFragment`.

`ChooseTypeFragment` содержит поле выбора с идентификатором `choose` и кнопку с идентификатором `next`. Когда пользователь щелкает на кнопке, фрагмент `ChooseTypeFragment` должен перейти к `DrinksFragment` и передать значение, выбранное пользователем в поле выбора.

Фрагмент `DrinksFragment` должен вывести выбранное значение в текстовом представлении с идентификатором `choice` своего макета.

Кто-то попытался выложить код `ChooseTypeFragment` и `DrinksFragment` из магнитов, но от порыва ветра часть магнитов упала на пол. Удастся ли вам вернуть упавшие магниты на свои места?

Navigation graph:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/chooseTypeFragment">

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment" />
    </fragment>
    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" >
        <argument
            android:name="drinkType"
            app:argType="string" />
    </fragment>
</navigation>
```

Код фрагмента приведен на следующей странице.

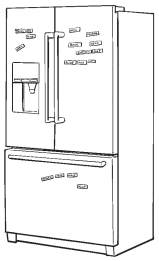
ChooseTypeFragment:

```
class ChooseTypeFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_choose_type, container, false)
        val choice = view.findViewById<Spinner>(R.id.choose).selectedItem.toString()
        val nextButton = view.findViewById<Button>(R.id.next)
        nextButton.setOnClickListener {

            val action = ..........................................................................

                         ..........................................................................(choice)
            view.findNavController().navigate(action)
        }
        return view
    }
}
```

DrinksFragment:

```
class DrinksFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_drinks, container, false)

        val choice = ..........................................................................
        val choiceView = view.findViewById<TextView>(R.id.choice)
        choiceView.text = choice
        return view
    }
}
```

Использовать все магниты необязательно.



Развлечения с магнитами. Решение

Приведенный ниже код графа навигации определяет путь навигации между двумя фрагментами: `ChooseTypeFragment` и `DrinksFragment`.

`ChooseTypeFragment` содержит поле выбора с идентификатором `choose` и кнопку с идентификатором `next`. Когда пользователь щелкает на кнопке, фрагмент `ChooseTypeFragment` должен перейти к `DrinksFragment` и передать значение, выбранное пользователем в поле выбора.

Фрагмент `DrinksFragment` должен вывести выбранное значение в текстовом представлении с идентификатором `choice` своего макета.

Кто-то попытался выложить код `ChooseTypeFragment` и `DrinksFragment` из магнитов, но от порыва ветра часть магнитов упала на пол. Удастся ли вам вернуть упавшие магниты на свои места?

Граф навигации:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/chooseTypeFragment">

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment" />
    </fragment>
    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" >
        <argument
            android:name="drinkType"
            app:argType="string" />
    </fragment>
</navigation>
```

Код фрагмента приведен на следующей странице.

ChooseTypeFragment:

```
class ChooseTypeFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_choose_type, container, false)
        val choice = view.findViewById<Spinner>(R.id.choose).selectedItem.toString()
        val nextButton = view.findViewById<Button>(R.id.next)
        nextButton.setOnClickListener {
            val action = ChooseTypeFragmentDirections
                         .actionChooseTypeFragmentToDrinksFragment(choice)
            view.findNavController().navigate(action)
        }
        return view
    }
}
```

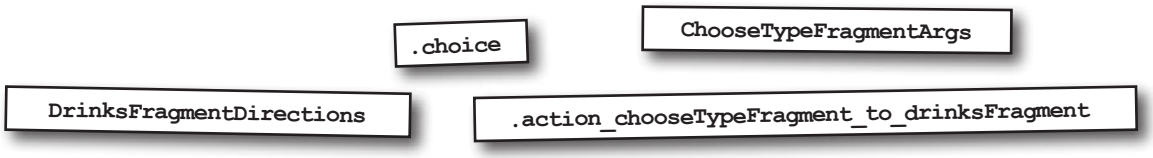
Этот класс используется для перехода от ChooseTypeFragment.

DrinksFragment:

```
class DrinksFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_drinks, container, false)
        val choice = DrinksFragmentArgs.fromBundle(requireArguments()).drinkType
        val choiceView = view.findViewById<TextView>(R.id.choice)
        choiceView.text = choice
        return view
    }
}
```

Имя аргумента DrinksFragment в графе навигации.

Эти магниты не понадобились.



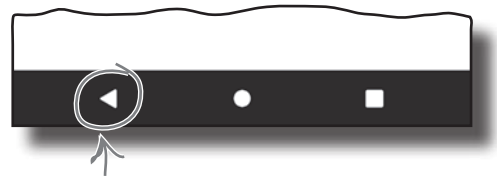
А если пользователь захочет вернуться?

EncryptFragment
Safe Args
Кнопка Back

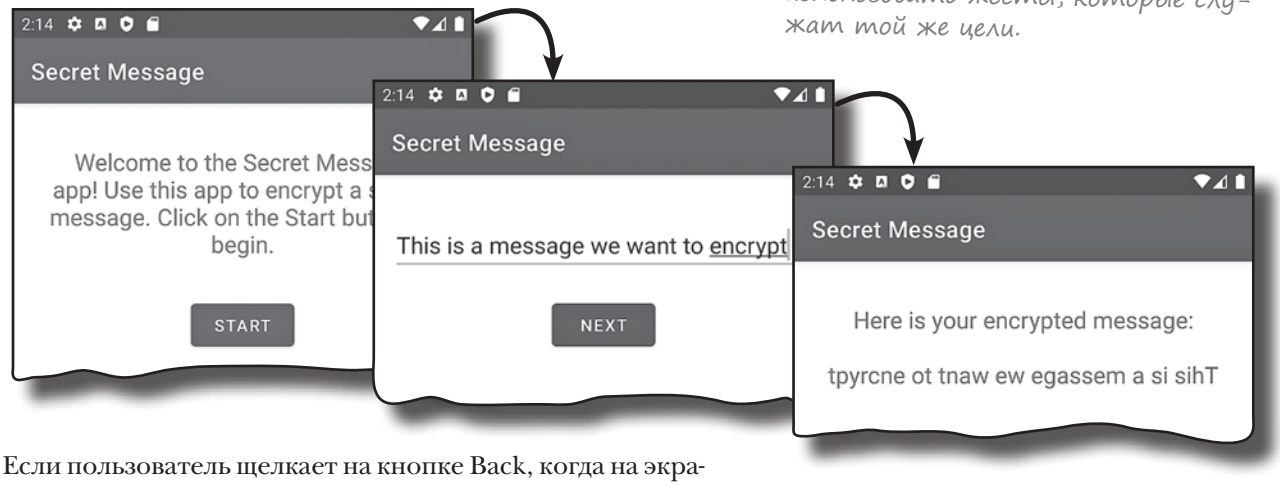
Осталось последнее, на что необходимо обратить внимание в приложении Secret Message: что происходит, когда пользователь пытается вернуться обратно по экранам приложения?

Как вам наверняка известно, в мире Android возврат выполняется нажатием кнопки возврата на устройстве или жестом возврата, которые возвращают вас к предыдущим экранам.

Допустим, пользователь запускает приложение Secret Message и переходит сначала от `WelcomeFragment` к `MessageFragment`, а затем к `EncryptFragment`:

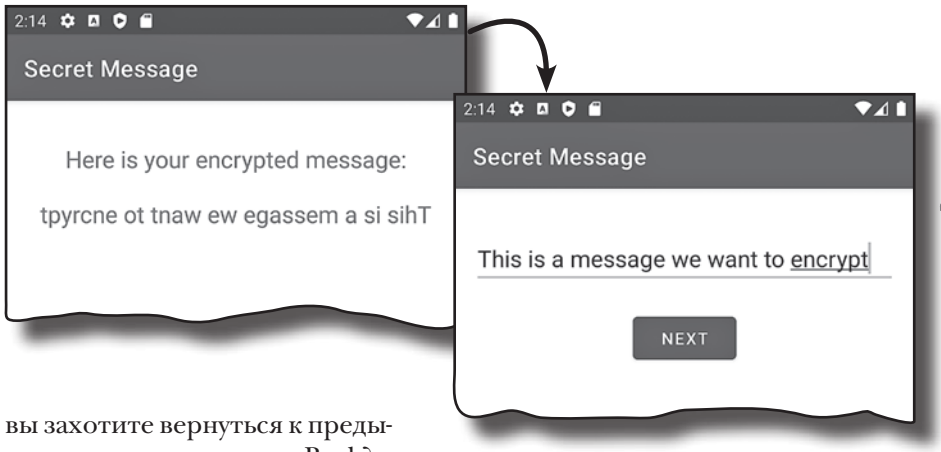


Кнопка Back появляется в нижней части экрана устройства. Некоторые пользователи или устройства могут использовать жесты, которые служат той же цели.



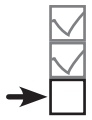
Если пользователь щелкает на кнопке Back, когда на экране отображается `EncryptFragment`, приложение возвращается к предыдущему фрагменту — `MessageFragment`:

Если на экране отображается Encrypt Fragment, а пользователь щелкает на кнопке Back...



...приложение возвращается к Message Fragment.

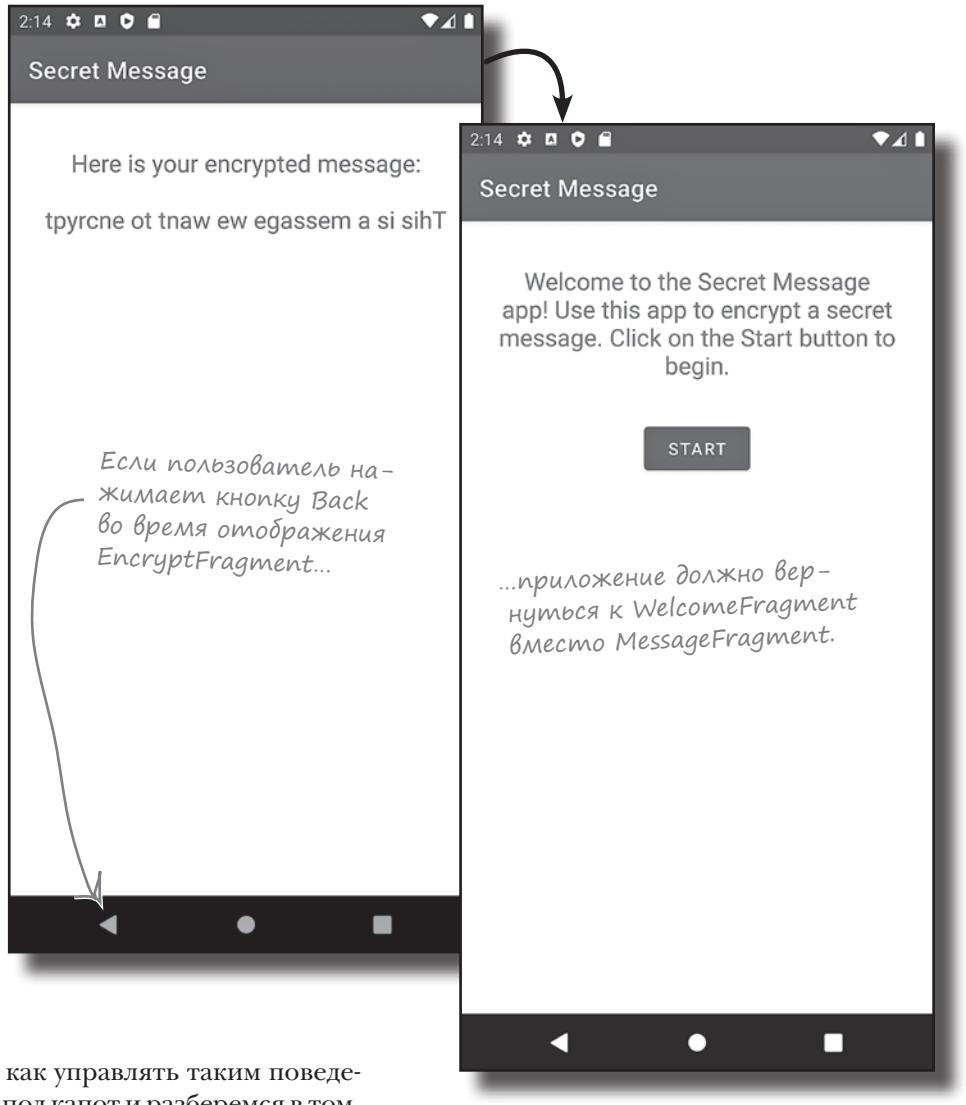
Но что, если вы захотите вернуться к предыдущему фрагменту нажатием кнопки Back?



EncryptFragment
Safe Args
Кнопка Back

Поведение возврата можно изменить

Вместо того чтобы возвращаться к `MessageFragment` при нажатии кнопки Back, возможно, будет лучше перейти прямо к `WelcomeFragment`. Это позволит пользователю снова начать с первого фрагмента:



Чтобы понять, как управлять таким поведением, заглянем под капот и разберемся в том, как работает стек возврата Android.

Знакомство со стеком возврата

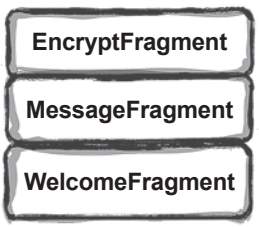
Когда вы переходите между целями в вашем приложении, Android отслеживает все посещенные точки, добавляя их в **стек возврата**. Стек возврата содержит журнал всех посещенных точек приложения. Каждый раз, когда вы переходите к какой-нибудь точке, Android добавляет ее на вершину стека, а при нажатии кнопки Back из стека извлекается последняя цель и отображается цель, находящаяся под ней.

Сценарий использования стека возврата:

1. **Когда вы запускаете приложение Secret Message, отображается фрагмент WelcomeFragment.**
   Android добавляет `WelcomeFragment` в стек возврата.

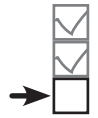
2. **Вы переходите к MessageFragment.**
   Эта цель помещается на вершину стека возврата, над `WelcomeFragment`.

3. **Затем вы переходите к EncryptFragment.**
   `EncryptFragment` помещается на вершину стека возврата.



4. **Вы щелкаете на кнопке Back, EncryptFragment извлекается из стека возврата.**
   Отображается фрагмент `MessageFragment`, так как он в данный момент находится на вершине стека.

5. **Вы снова щелкаете на кнопке Back.**
   `MessageFragment` извлекается с вершины стека возврата, отображается фрагмент `WelcomeFragment`.

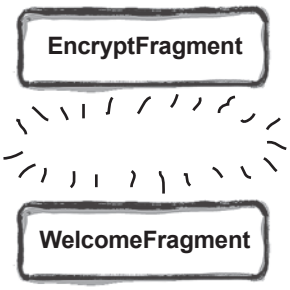


EncryptFragment
Safe Args
Кнопка Back

Использование графа навигации для извлечения фрагментов из стека возврата

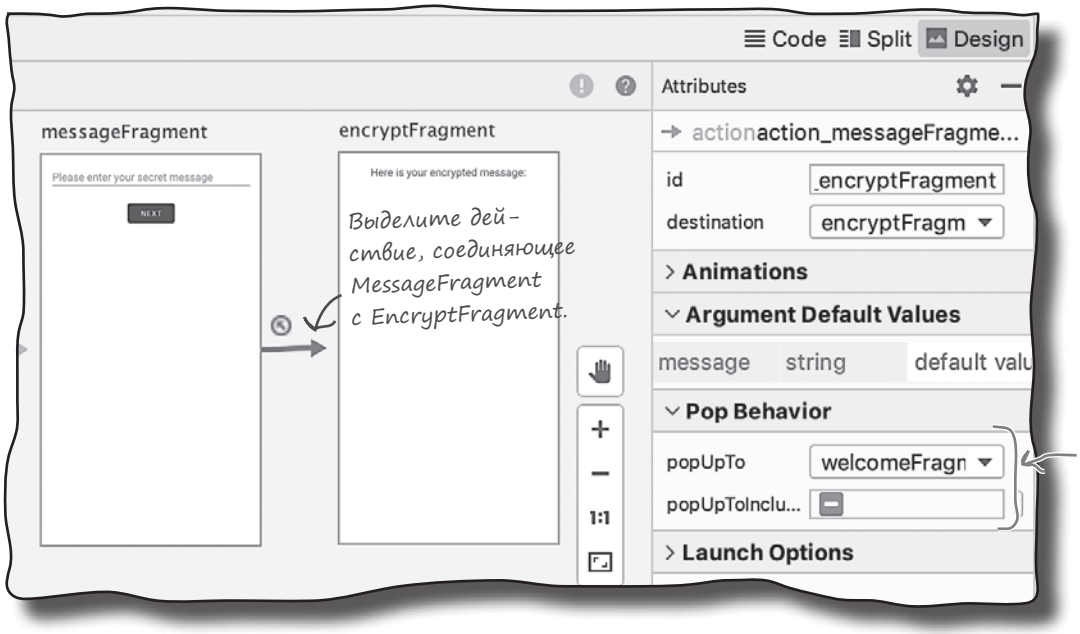
Вы только что узнали, как стек возврата и кнопка Back работают по умолчанию. Однако при желании вы можете извлекать цели из стека возврата в ходе перемещений пользователя в приложении. Для этого следует задать в графе навигации **поведение извлечения**.

Чтобы понять, как работает этот механизм, обновим граф навигации в приложении Secret Message, чтобы при переходе приложения от `MessageFragment` к `EncryptFragment` фрагмент `MessageFragment` извлекался из стека возврата. Это означает, что когда пользователь щелкает на кнопке Back при отображаемом фрагменте `EncryptFragment`, вместо `MessageFragment` отображается фрагмент `WelcomeFragment`.



Удаление MessageFragment из стека возврата означает, что при щелчке на кнопке Back из EncryptFragment вместо MessageFragment отображается WelcomeFragment.

Откройте граф навигации *nav_graph.xml* (если он не был открыт ранее) и переключитесь в визуальный редактор. Выделите действие, соединяющее `MessageFragment` с `EncryptFragment`. Затем в разделе Pop Behavior панели Attributes измените значение свойства `popUpTo` на «welcomeFragment». Тем самым вы даете команду Android извлекать фрагменты из стека возврата, пока не будет достигнут фрагмент `WelcomeFragment`:



Обновите поведение извлечения из стека, чтобы при использовании этого действия контроллер навигации извлекал из стека возврата все фрагменты до WelcomeFragment. Если бы popUpInclusive было присвоено значение true, то из стека также был бы извлечен фрагмент WelcomeFragment.

Прежде чем проводить тест-драйв приложения, посмотрим, как выглядит разметка XML.

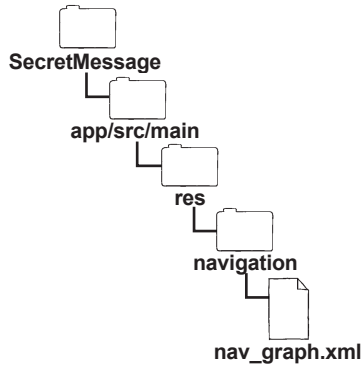
EncryptFragment
Safe Args
Кнопка Back

Обновленный код nav_graph.xml

Когда вы определяете для действия поведение извлечения, в разметку XML добавляется элемент <popUpTo>. Он указывает, насколько далеко будут извлекаться фрагменты из стека возврата.

Ниже приведен обновленный код *nav_graph.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/welcomeFragment">

    <fragment
        android:id="@+id/welcomeFragment"
        android:name="com.hfad.secretmessage.WelcomeFragment"
        android:label="fragment_welcome"
        tools:layout="@layout/fragment_welcome" >
        <action
            android:id="@+id/action_welcomeFragment_to_messageFragment"
            app:destination="@id/messageFragment" />
    </fragment>
    <fragment
        android:id="@+id/messageFragment"
        android:name="com.hfad.secretmessage.MessageFragment"
        android:label="fragment_message"
        tools:layout="@layout/fragment_message" >
        <action
            android:id="@+id/action_messageFragment_to_encryptFragment"
            app:destination="@id/encryptFragment"
            app:popUpTo="@id/welcomeFragment" />
    </fragment>
    <fragment
        android:id="@+id/encryptFragment"
        android:name="com.hfad.secretmessage.EncryptFragment"
        android:label="fragment_encrypt"
        tools:layout="@layout/fragment_encrypt" >
        <argument
            android:name="message"
            app:argType="string" />
    </fragment>
</navigation>
```

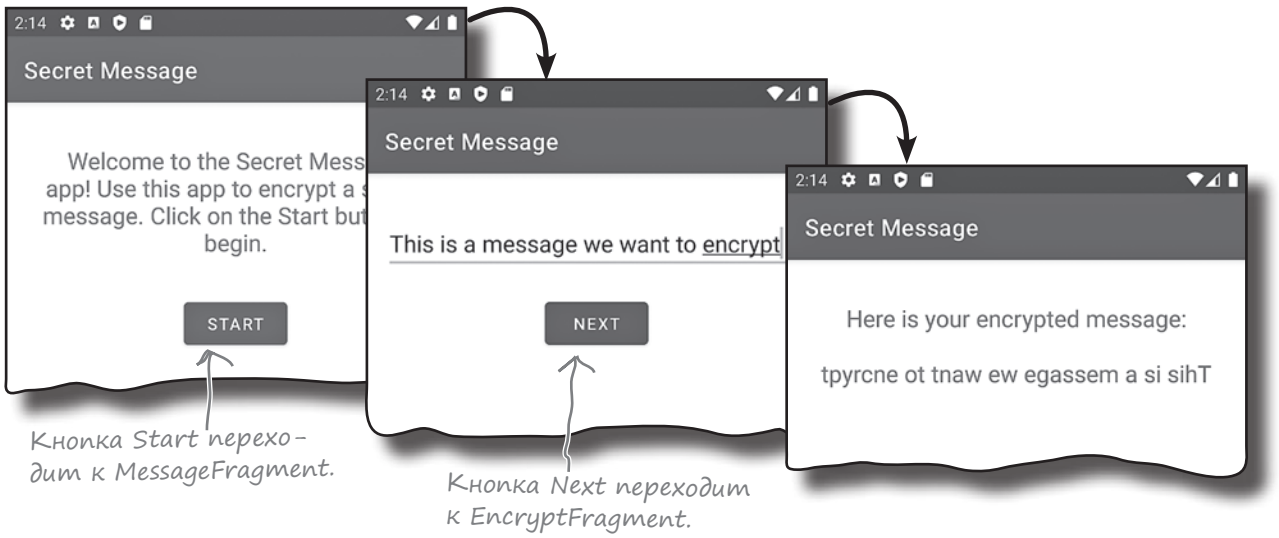


Этот новый атрибут извлекает фрагменты из стека возврата до WelcomeFragment.

Тест-драйв

- EncryptFragment
- Safe Args
- **Кнопка Back**

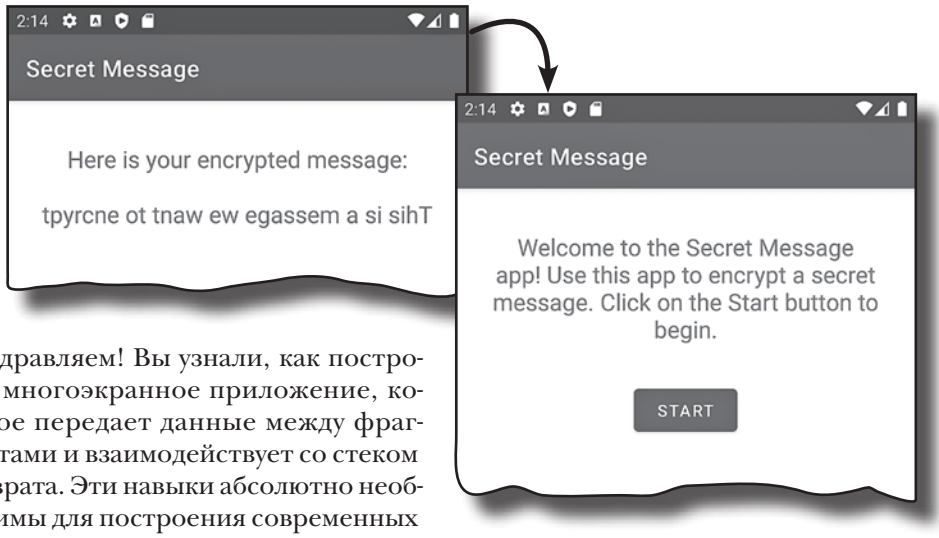
При запуске приложения отображается фрагмент `WelcomeFragment`. Как и прежде, вы можете перейти к `MessageFragment` и `EncryptFragment`:



Кнопка Start переходит к MessageFragment.

Кнопка Next переходит к EncryptFragment.

Когда вы щелкаете на кнопке, приложение возвращается к `WelcomeFragment` в обход `MessageFragment`:



Щелчок на кнопке Back из EncryptFragment возвращает к WelcomeFragment.

Поздравляем! Вы узнали, как построить многоэкранное приложение, которое передает данные между фрагментами и взаимодействует со стеком возврата. Эти навыки абсолютно необходимы для построения современных приложений Android.

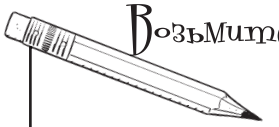
Другие возможности использования компонента Navigation рассматриваются в следующей главе.

СТАНЬ плагином Safe Args

Ниже приведен код графа навигации для приложения StarBuzz. Представьте себя на месте плагина Safe Args и скажите, какие классы Directions и Args будут сгенерированы для этого кода.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/selectCoffeeFragment">

    <fragment
        android:id="@+id/selectCoffeeFragment"
        android:name="com.hfad.starbuzz.SelectCoffeeFragment"
        android:label="fragment_select_coffee"
        tools:layout="@layout/fragment_select_coffee" >
        <action
            android:id="@+id/action_selectCoffeeFragment_to_coffeeFragment"
            app:destination="@id/coffeeFragment" />
    </fragment>

    <fragment
        android:id="@+id/coffeeFragment"
        android:name="com.hfad.starbuzz.CoffeeFragment"
        android:label="fragment_coffee"
        tools:layout="@layout/fragment_coffee" >
        <argument
            android:name="coffee"
            app:argType="string" />
    </fragment>
</navigation>
```

Ответы на с. 328.



Возьмите в руку карандаш

Приведенный ниже граф навигации обеспечивает переход от `TitleFragment` к `ChooseTypeFragment` и от `ChooseTypeFragment` к `DrinksFragment`.

Как бы вы обновили код, чтобы при нажатии кнопки Back при отображаемом фрагменте `DrinksFragment` приложение возвращалось сразу к `TitleFragment`?

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/titleFragment">

    <fragment
        android:id="@+id/titleFragment"
        android:name="com.hfad.drinksapp.TitleFragment"
        android:label="fragment_title"
        tools:layout="@layout/fragment_title" >
        <action
            android:id="@+id/action_titleFragment_to_chooseTypeFragment"
            app:destination="@id/chooseTypeFragment" />
    </fragment>

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment" />
    </fragment>

    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" />
</navigation>
```

Ответы на с. 329.

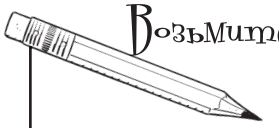
СТАНЬ плагином Safe Args. Решение

Ниже приведен код графа навигации для приложения StarBuzz. Представьте себя на месте плагина Safe Args и скажите, какие классы Directions и Args будут сгенерированы для этого кода.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/selectCoffeeFragment">

    <fragment
        android:id="@+id/selectCoffeeFragment"
        android:name="com.hfad.starbuzz.SelectCoffeeFragment"
        android:label="fragment_select_coffee"
        tools:layout="@layout/fragment_select_coffee" >
        <action
            android:id="@+id/action_selectCoffeeFragment_to_coffeeFragment"
            app:destination="@id/coffeeFragment" />
    </fragment>

    <fragment
        android:id="@+id/coffeeFragment"
        android:name="com.hfad.starbuzz.CoffeeFragment"
        android:label="fragment_coffee"
        tools:layout="@layout/fragment_coffee" >
        <argument
            android:name="coffee"
            app:argType="string" />
    </fragment>
</navigation>
```

Возможен переход от SelectCoffeeFragment, поэтому Safe Args генерирует класс SelectCoffeeFragmentDirections.

CoffeeFragment получает аргумент, поэтому Safe Args генерирует класс CoffeeFragmentArgs.



Возьмите в руку карандаш

Решение

Приведенный ниже граф навигации обеспечивает переход от `TitleFragment` к `ChooseTypeFragment` и от `ChooseTypeFragment` к `DrinksFragment`.

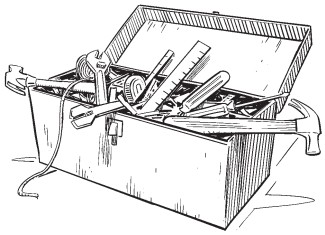
Как бы вы обновили код, чтобы при нажатии кнопки Back при отображаемом фрагменте `DrinksFragment` приложение возвращалось сразу к `TitleFragment`?

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/titleFragment">

    <fragment
        android:id="@+id/titleFragment"
        android:name="com.hfad.drinksapp.TitleFragment"
        android:label="fragment_title"
        tools:layout="@layout/fragment_title" >
        <action
            android:id="@+id/action_titleFragment_to_chooseTypeFragment"
            app:destination="@id/chooseTypeFragment" />
    </fragment>

    <fragment
        android:id="@+id/chooseTypeFragment"
        android:name="com.hfad.drinksapp.ChooseTypeFragment"
        android:label="fragment_choose_type"
        tools:layout="@layout/fragment_choose_type" >
        <action
            android:id="@+id/action_chooseTypeFragment_to_drinksFragment"
            app:destination="@id/drinksFragment"
            app:popUpTo="@id/titleFragment" />
    </fragment>

    <fragment
        android:id="@+id/drinksFragment"
        android:name="com.hfad.drinksapp.DrinksFragment"
        android:label="fragment_drinks"
        tools:layout="@layout/fragment_drinks" />
</navigation>
```

Необходимо добавить эту строку, чтобы при переходе к DrinksFragment фрагмент ChooseTypeFragment извлекался из стека возврата.

Ваш инструментарий Android

ГЛАВА 7



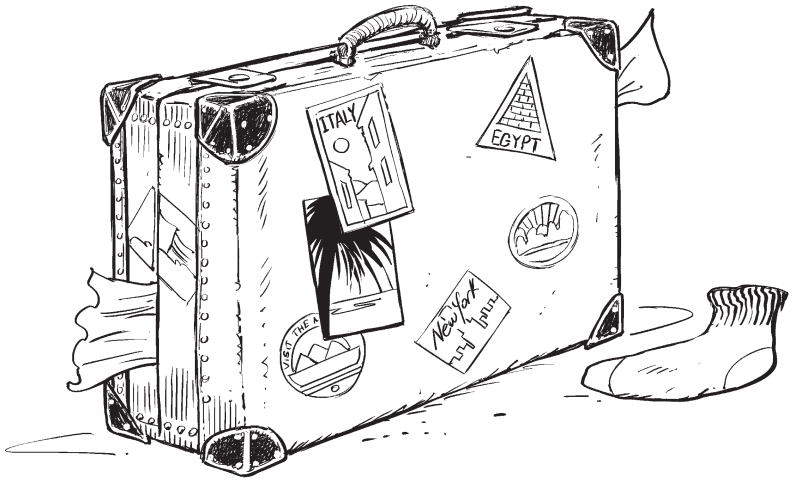
Глава 7 осталась позади, а ваш инструментарий пополнился плагином Safe Args и операциями со стеком возврата.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Плагин Safe Args — дополнительная часть компонента Navigation, обеспечивающая передачу аргументов способом, безопасным по отношению к типам. Он генерирует классы `Directions` и `Args`.
- Классы `Directions` используются для передачи аргументов целям.
- Safe Args генерирует класс `Directions` для каждой цели, от которой может выполняться переход.
- Классы `Args` используются для получения аргументов, передаваемых целям.
- Safe Args генерирует класс `Args` для каждой цели, получающей аргументы.
- Поведение извлечения в графе навигации используется для вывода целей из стека возврата.

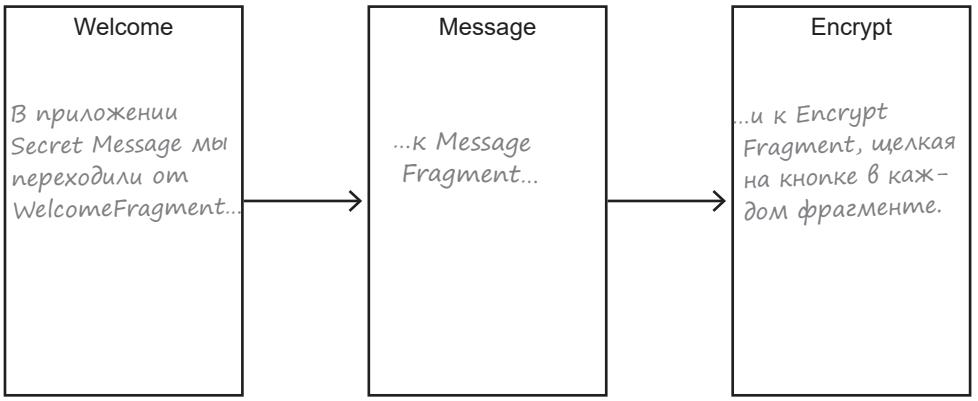
Туда и обратно



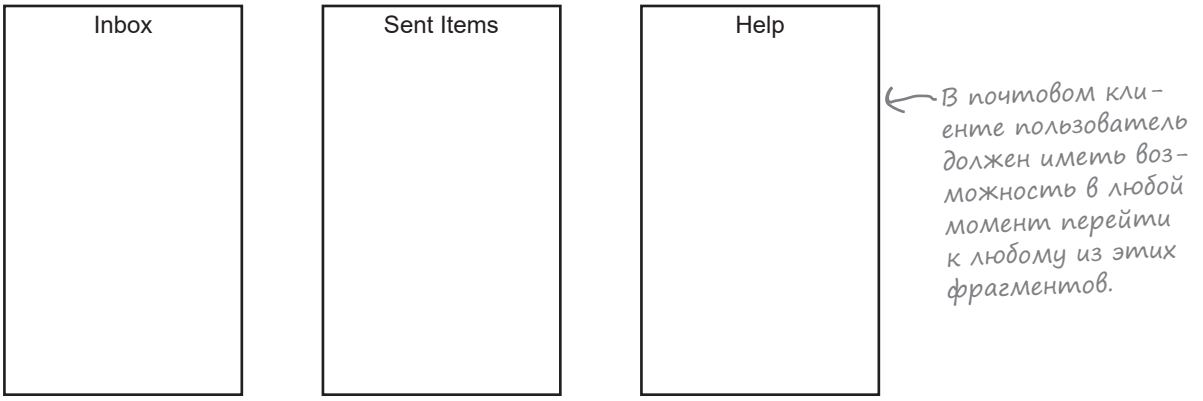
Во многих приложениях возникает необходимость в переходе между разными целями. Компонент Navigation сильно упрощает построение таких пользовательских интерфейсов. В этой главе вы научитесь пользоваться некоторыми навигационными UI-компонентами для Android, **упрощающими перемещение пользователя по приложению**. Вы узнаете, как пользоваться **темами** и как заменить панель приложения **панелью инструментов**. Вы научитесь добавлять **элементы меню**, которые могут использоваться для **навигации**, и освоите **реализацию навигации с нижней панели**. Наконец, мы создадим эффектную выдвижную панель, которая появляется на экране сбоку от вашей активности

Разные приложения, разные структуры

В двух предыдущих главах вы научились использовать Android-компонент Navigation для перехода между фрагментами путем щелчка на кнопке. Такой подход хорошо работал в приложении Secret Message, потому что переходы между целями были линейными:



Однако не все приложения используют такую структуру. Во многих приложениях существуют экраны, к которым пользователь должен переходить независимо от того, в какой точке приложения он находится. Например, в почтовом клиенте могут быть папки Входящие, Отправленные и Справка, которые должны открываться с минимальными усилиями:



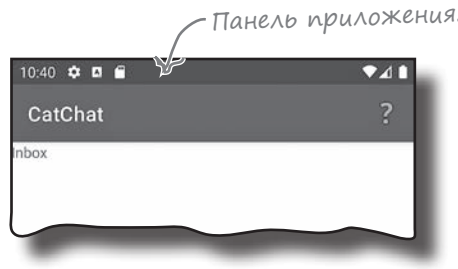
Итак, в вашем приложении используется нелинейная схема навигации. Как же сделать так, чтобы все экраны были постоянно доступными?

В Android существуют навигационные UI-компоненты

Если у вас есть экраны, которые должны быть доступны из любой точки приложения, возможно, стоит воспользоваться одним из навигационных UI-компонентов Android:

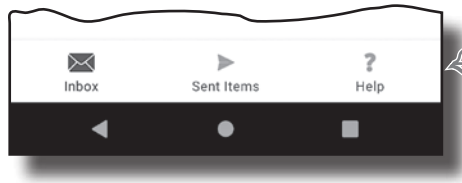
Панель приложения

Это панель, которая отображается у верхнего края экрана. Android обычно включает такую панель по умолчанию. На нее можно добавлять элементы, по щелчку на которых происходит переход к целям.



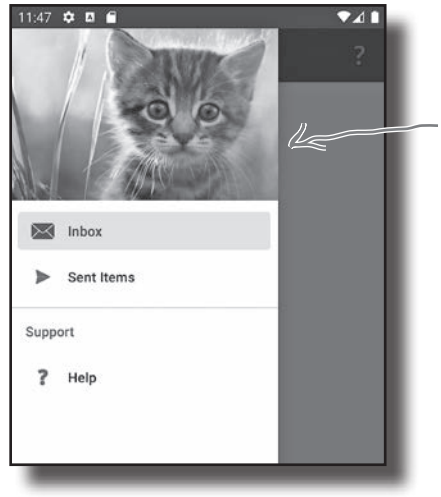
Нижняя панель навигации

Отображается у нижнего края экрана. На ней размещаются элементы, которые могут использоваться для навигации.



Выдвижная панель навигации

Панель, которая выдвигается от стороны экрана. Такие панели используются во многих приложениях, так как они обладают очень гибкими возможностями.



В этой главе вы узнаете, как реализовать все три типа навигационных UI-компонентов. Для этого мы построим прототип почтового клиента CatChat.

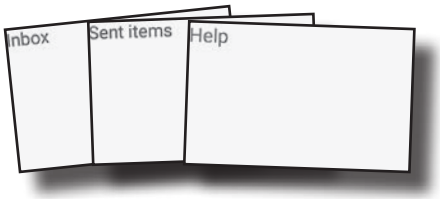
Мозговой штурм

Подумайте, в каких известных вам Android-приложениях используются этим компоненты. Как вы думаете, для каких ситуаций лучше подходит каждый из них? Почему?

Как работает приложение CatChat

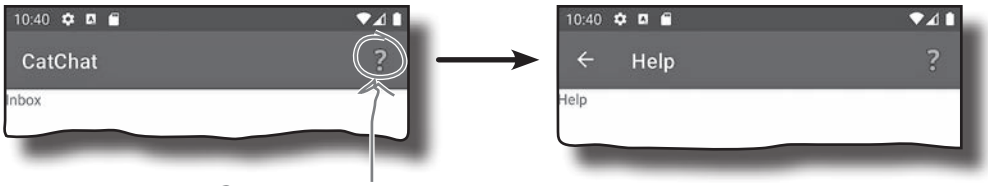
В приложении CatChat будет использоваться одна активность с именем `MainActivity` и три фрагмента: `InboxFragment`, `SentItemsFragment`, `HelpFragment`. Каждый фрагмент будет отображаться в `MainActivity` при переходе к нему.

Это фрагменты.



В реальном почтовом клиенте в этих фрагментах будут отображаться списки входящих и отправленных сообщений, а также справка. Так как нас сейчас интересует только навигация, в каждом фрагменте просто выводится текст.

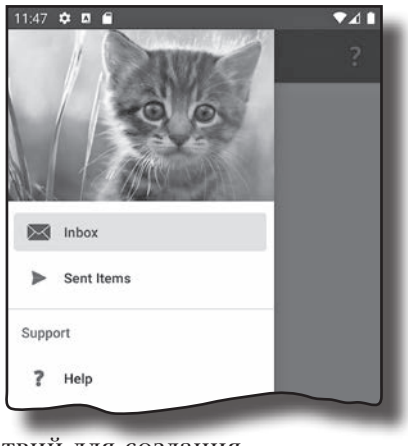
Приложение будет включать панель приложения, на которой размещается элемент для открытия меню справки. Когда пользователь щелкает на нем, приложение переходит к фрагменту `HelpFragment` и отображает его в `MainActivity`.



Элемент, который будет использоваться для перехода к HelpFragment.

В приложении также будет использоваться выдвижная панель для перехода между всеми тремя фрагментами. На выдвижной панели размещается элемент для каждого фрагмента, и при щелчке на каждом элементе будет отображаться соответствующий фрагмент

Выдвижная панель навигации будет открывать меню с элементами для всех трех фрагментов. При щелчке на каждом элементе открывается соответствующий фрагмент



Рассмотрим последовательность действий для создания приложения CatChat

Что мы собираемся сделать

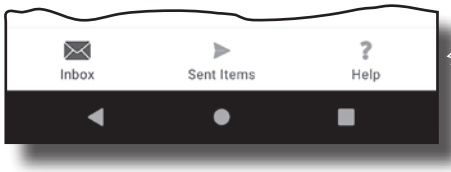
Приложение CatChat будет создано за четыре этапа.

1. **Замена стандартной панели приложения панелью инструментов.**
Панель инструментов внешне похожа на стандартную панель приложения, но она обладает большей гибкостью и включает новейшую функциональность панелей приложений. Заодно вы научитесь применять темы и стили.

2. **Добавление элемента справки на панель инструментов.**
По щелчку на элементе меню Help будет происходить переход к `HelpFragment`.

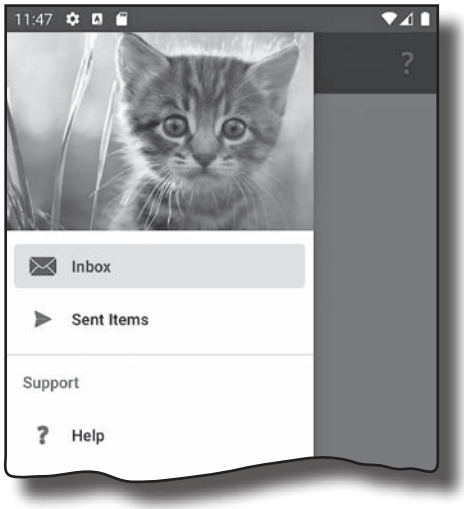
Этот элемент будет добавлен на панель инструментов.

3. **Реализация нижней панели навигации.**
Мы добавим в приложение нижнюю панель, с которой можно перейти к каждому из фрагментов приложения.



В окончательной версии приложения нижней панели не будет, но мы хотим показать вам, как она создается.

4. **Создание выдвижной панели.**
Наконец, нижняя панель навигации будет заменена выдвижной панелью. На выдвижной панели будет отображаться графический заголовок и элементы, представляющие все фрагменты приложения, разделенные на группы.



Начнем с создания нового проекта для приложения CatChat.

- [x] **Панель инструментов**
- [ ] **Панель инструментов навигации**
- [ ] **Нижняя панель**
- [ ] **Выдвижная панель**

Создание нового проекта

Для приложения CatChat понадобится новый проект; создайте его в Android Studio по схеме, уже знакомой вам по предыдущим главам. Выберите вариант Empty Activity, введите имя «CatChat» и имя пакета «com.hfad.catchat», подтвердите папку для сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin с минимальным уровнем SDK API 21, чтобы приложение работало на большинстве устройств Android.

Создается стандартная панель приложения...

Когда вы создаете новый проект с пустой активностью, Android Studio обычно добавляет панель приложения, на которой выводится имя приложения. Эта панель отображается в верхней части экрана при запуске приложения:

Стандартная панель приложения, которую Android Studio создает за вас. Если ваша версия Android Studio не добавила панель приложения, не беспокойтесь — через пару страниц вы узнаете, как добавить ее вручную.

Панель приложения полезна по нескольким причинам:

- **Она делает ключевые действия (например, публикацию контента и выполнение поиска) более заметными и при этом более предсказуемыми.**
- **Она помогает пользователю понять, в какой точке приложения он находится; для этого на панели выводится имя приложения или описание текущего экрана.**
- **Она может использоваться для перехода к разным целям.**

Но как добавляется панель приложения?

...путем применения темы

Стандартная панель приложения добавляется в приложение путем применения **темы**. Тема обеспечивает целостность оформления приложения на разных экранах. Она управляет внешним видом приложения и наличием у него панели приложения.

Android включает ряд тем, которые могут использоваться в ваших приложениях. По умолчанию Android Studio применяет тему, включающую панель приложения.

Использование темы Material Design

В приложении CatChat будет использоваться тема **Material Design**.

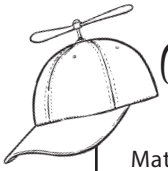
Система Material Design была разработана компанией Google. Она помогает строить качественные приложения и веб-сайты с целостным оформлением. Идея заключалась в том, чтобы пользователь, переключающийся с приложения Google (такого, как Play Store) на приложение, созданное сторонним разработчиком, мгновенно чувствовал себя в знакомой обстановке и знал, что делать.

В основу Material Design изначально были заложены принципы дизайна печатных материалов, отражающие внешний вид и поведение реальных объектов (например, карточек для записей и листков бумаги.) Новейшей стадией ее эволюции стала система Material You, предоставляющая пользователю более динамичный опыт взаимодействия с персонализированной цветовой палитрой.

Файл приложения build.gradle должен содержать зависимость от библиотеки Material

Темы Material хранятся в отдельной библиотеке, которая должна быть включена в приложение. Для этого следует добавить зависимость для библиотеки *com.google.android.material* в файл *build.gradle* приложения.

Так как мы собираемся использовать тему Material в приложении CatChat, необходимо проверить, что эта библиотека включена в приложение. Откройте файл *CatChat/app/build.gradle* и убедитесь в том, что он включает следующую строку (выделена жирным шрифтом):



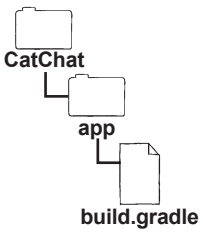
Серьезное программирование

Material Design включает целый ряд компонентов, тем, инструментов и рекомендаций. Некоторые из них будут применены на практике в следующих двух главах, а дополнительную информацию можно найти по адресу

https://material.io

```
dependencies {
    ...
    implementation 'com.google.android.material:material:1.4.0'
    ...
}
```

Используемая версия библиотеки Material.



Возможно, среда Android Studio уже включила эту зависимость за вас. Если это не было сделано, вам придется добавить ее самостоятельно и щелкнуть на ссылке Sync Now, появляющейся в верхней части редактора кода, чтобы синхронизировать изменения с оставшейся частью проекта.

Мы хотим применить тему Material в приложении CatChat для управления его панелью приложения. Давайте посмотрим, как это делается.

Применение темы в AndroidManifest.xml

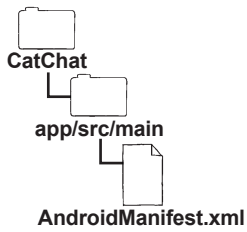
Тема применяется в файле *AndroidManifest.xml* приложения. Как вы узнали в главе 3, этот файл предоставляет информацию о конфигурации приложения. В нем содержатся атрибуты — включая тему, — непосредственно влияющие на панель приложения.

Ниже приведен код *AndroidManifest.xml*, созданный за нас средой Android Studio в проекте CatChat (ключевые части выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hfad.catchat">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.CatChat">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Имя приложения, удобное для пользователя.

Тема, назначаемая для приложения.



Атрибут **android:label** ссылается на текст, выводимый в панели приложения, — то самое имя, которое вы указываете при исходном создании приложения. В приведенном выше коде используется строковый ресурс с именем `app_name`, который был добавлен средой Android Studio в *strings.xml*.

Атрибут **android:theme** определяет тему. В приведенном выше коде он задается директивой:

```
android:theme="@style/Theme.CatChat">
```

Это значение сообщает, что тема определяется как стилевой ресурс (обозначаемый `@style`) с именем `Theme.CatChat`.

Определение стилей в файлах стилевых ресурсов

Стилевые ресурсы используются для описания любых тем и стилей, которые должны использоваться приложением, и хранятся в одном или в нескольких файлах стилевых ресурсов.

Когда мы создаем приложение CatChat, среда Android Studio создает два файла стилевых ресурсов за нас. Обоим файлам присваивается имя *themes.xml*, и они размещаются в папках *app/src/main/res/values* и *app/src/main/res/values-night*. Файл в папке *values* содержит файл стилевых ресурсов приложения, используемый по умолчанию, а файл в папке *values-night* используется ночью.

В некоторых версиях Android Studio этому файлу присваивается имя styles.xml.

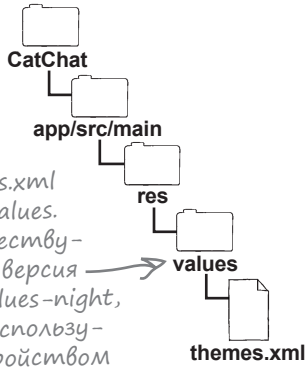
themes.xml в папке *values* определяет код стиля следующего вида:

Стиль базируется на этой теме.

Определяет стиль.

Эти строки переопределяют цвета родительской схемы.

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.CatChat" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        ...
    </style>
</resources>
```



Код themes.xml из папки values. Также существует другая версия в папке values-night, которая используется устройством в ночное время.

Элемент **<style>** сообщает Android, что определяется стилевой ресурс. Каждому стилю должно быть присвоено имя, по которому он идентифицируется. Имя определяется атрибутом **name**:

```
name="Theme.CatChat"
```

Стилю присваивается имя Theme.CatChat.

Атрибут `theme` из *AndroidManifest.xml* использует это имя для назначения темы приложения, например `"@style/Theme.CatChat"`.

Стиль также включает атрибут **parent**, который указывает, на основе какой темы должен определяться стиль. В приведенном выше коде используется значение:

```
parent="Theme.MaterialComponents.DayNight.DarkActionBar"
```

Эта тема отображает темную панель приложения и позволяет приложению переходить между дневными и ночными стилями.

соответствующее теме с темной панелью приложения, которая позволяет приложению переходить между дневной и ночной темой. В дневное время он используется стиль, определенный в файле стилевых ресурсов из папки *values*, а по ночам устройство переключается на стиль из папки *values-night*.

Приведенный выше стиль также включает элементы `<item>`, которые переопределяют некоторые цвета темы. Рассмотрим их более подробно.

Стили могут переопределять цвета темы

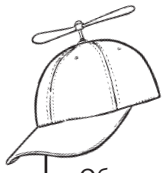
Если вы захотите переопределить любые свойства родительской темы (например, цветовую схему), для этого следует добавить в стиль элементы <item>. Например, в приложении CatChat включены элементы для переопределения основного и дополнительного цвета. Основной цвет используется приложением для прорисовки таких объектов, как панель приложения. Дополнительный цвет используется некоторыми представлениями для создания контраста.

Можно использовать несколько файлов стилевых ресурсов для применения разных цветовых схем в разных ситуациях. Так, приложение CatChat включает один файл стилевых ресурсов в папке *values* и другой файл в папке *values-night*. Такая схема — в сочетании с темой DayNight, на основе которой определяется каждый стиль, — позволяет применять разные цветовые схемы в дневное и ночное время.

Ниже приведен код, используемый приложением CatChat для переопределения основных цветов в файле *themes.xml* из папки *values*:

```
<item name="colorPrimary">@color/purple_500</item>
<item name="colorPrimaryVariant">@color/purple_700</item>
<item name="colorOnPrimary">@color/white</item>
```

Как видите, у каждого элемента имеется атрибут name, а обозначение @color ссылается на цвет из **файла цветовых ресурсов**.



Серьезное программирование

Об использованных в этом примере цветах можно больше узнать по адресу

https://material.io/develop/android/theming/color

Также существует приложение, которое поможет вам выбрать цветовую схему приложения:

https://material.io/tools/color

Файлы цветовых ресурсов определяют набор цветов

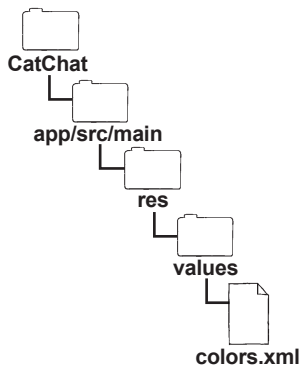
Когда вы создаете новый проект, среда Android Studio обычно включает стандартный файл цветовых ресурсов с именем *colors.xml*. Он находится в папке *app/src/main/res/values*, а содержащиеся в нем цвета могут использоваться в вашем приложении.

Типичный файл цветовых ресурсов выглядит примерно так:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    ...
</resources>
```

Для каждого цвета указывается имя и шестнадцатеричный код.

Остальные цвета опущены.



Чтобы изменить цветовую схему приложения, включите нужные цвета в файл цветовых ресурсов и обращайтесь к ним из файлов стилевых ресурсов.

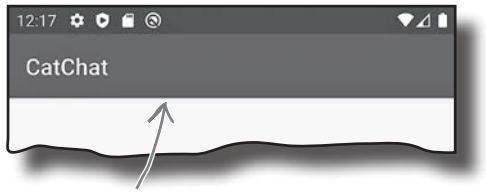
- [x] **Панель инструментов**
- [ ] Панель инструментов навигации
- [ ] Нижняя панель
- [ ] Выдвижная панель

Замена панели приложения по умолчанию панелью инструментов

Ранее вы видели, как Android Studio добавляет в приложение панель приложения по умолчанию, назначая ему тему. Добавить панель приложения таким способом несложно, но в более гибком варианте она **заменяется панелью инструментов**.

Простейшая панель инструментов не отличается от панели приложения по умолчанию, которую вы уже видели, но она выделяется намного большей гибкостью. Например, вы можете изменить ее высоту, а в следующей главе вы научитесь сворачивать и разворачивать ее при прокрутке экрана устройства. Также в нее включена новейшая функциональность панелей приложений, так что создание панели инструментов упрощает реализацию этой функциональности в ваших приложениях.

Чтобы заменить панель приложения по умолчанию, следует удалить исходную панель приложения, включить панель инструментов в ваш макет, а затем приказать активности использовать панель инструментов как его панель приложения. На ближайших страницах мы покажем, как это делается.



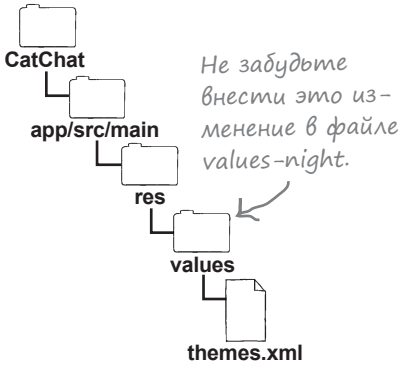
Панель инструментов выглядит так же, как уже известная вам панель приложения, но обладает большей гибкостью.

Удаление панели приложения по умолчанию с использованием темы

Чтобы удалить панель приложения по умолчанию, примените к приложению тему, не включающую панель приложения.

Например, текущая версия приложения CatChat использует тему `Theme.MaterialComponents.DayNight.DarkActionBar`. Мы удалим эту панель приложения и заменим ее на `Theme.MaterialComponents.DayNight.NoActionBar`. Эти две темы почти не отличаются, не считая того, что во второй нет панели приложения.

Давайте изменим тему: обновите код в файлах *themes.xml* из папок *values* и *values-night* и включите в них следующие изменения (выделены жирным шрифтом):



Не забудьте внести это изменение в файле values-night.

```
<resources>
    <style name="Theme.CatChat"
           parent="Theme.MaterialComponents.DayNight.~~DarkActionBar~~NoActionBar">
        ...
    </style>
</resources>
```

Измените тему, чтобы удалить панель приложения по умолчанию.

Вот и все, что необходимо знать для удаления панели приложения по умолчанию. На следующем шаге вы научитесь добавлять панели инструментов.

Панель инструментов является разновидностью View

В отличие от панели приложения по умолчанию, панель инструментов является разновидностью представления, которая добавляется в макет. А раз это представление, это означает, что вы можете в полной мере управлять его размером и позицией.

Доступны разные типы представлений, и в приложении CatChat мы будем использовать **панель инструментов Material**. Эта разновидность панели инструментов хорошо работает в сочетании с темами Material — такими, как использованная в нашем приложении.

Код добавления панели инструментов Material выглядит примерно так:

```
<com.google.android.material.appbar.MaterialToolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    style="@style/Widget.MaterialComponents.Toolbar.Primary" />
```

Определяет панель инструментов.

Назначает идентификатор для обращения к панели из кода активности.

Задает размер.

Применяет к панели стиль.

Сначала вы определяете панель инструментов:

```
<com.google.android.material.appbar.MaterialToolbar
    ... />
```

Полный путь к классу MaterialToolbar.

где `com.google.android.material.appbar.MaterialToolbar` — полный путь к классу `MaterialToolbar`. Затем другие атрибуты представлений используются для назначения идентификатора и определения внешнего вида панели. Например, чтобы ширина панели инструментов соответствовала ширине родителя, а высота — высоте панели приложения по умолчанию из соответствующей темы, можно использовать следующий код:

```
android:layout_width="match_parent"
android:layout_height="?attr/actionBarSize"
```

Префикс **`?attr`** в этом коде означает, что вы хотите использовать атрибут из текущей темы. В нашем конкретном случае запись `?attr/actionBarSize` обозначает высоту панели приложения по умолчанию для темы.

Также к панели инструментов можно применить стилевое оформление, чтобы она использовала основные цвета приложения. Это делается так:

```
style="@style/Widget.MaterialComponents.Toolbar.Primary"
```

Применяет стилевое оформление к панели инструментов, чтобы она своим внешним видом соответствовала панели приложения по умолчанию из темы.

Итак, теперь вы знаете, как выглядит код панели инструментов. Добавим панель инструментов в макет `MainActivity`.

Добавление панели инструментов в activity_main.xml

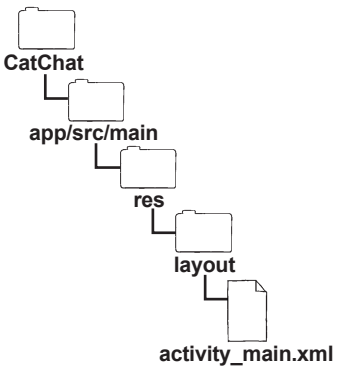
- **Панель инструментов**
- Панель инструментов навигации
- Нижняя панель
- Выдвижная панель

Мы добавим панель инструментов в макет `MainActivity`, чтобы она отображалась у верхнего края экрана. Обновите код в *activity_main.xml* и приведите его к следующему виду:

Используется линейный макет.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />
</LinearLayout>
```

Макет содержит панель инструментов Material без других представлений.



Где имя приложения?

Приведенный выше код добавляет панель инструментов в макет `MainActivity`, но панель еще не обладает никакой функциональностью панелей приложений. Например, если запустить приложение на этой стадии, вы увидите, что имя приложения не выводится на панели инструментов, как на стандартной панели приложения в предыдущей версии:

Если просто добавить панель инструментов в макет без внесения других изменений, она будет выглядеть как пустая полоса у верхнего края экрана. На ней не будет ни имени приложения, ни других аспектов функциональности панели.

Чтобы панель инструментов вела себя как нормальная панель приложения, необходимо внести изменения в код активности, написанный на Kotlin.

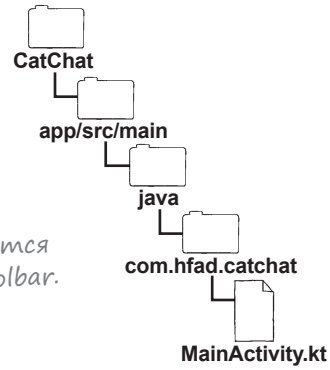
Замена панели инструментов панелью приложения MainActivity

Чтобы панель инструментов вела себя как привычная панель приложения, необходимо сообщить об этом `MainActivity`. Для этого следует вызвать в коде активности метод **`setSupportActionBar()`** и передать ему ссылку на панель инструментов:

```
val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
setSupportActionBar(toolbar)
```

Дает команду Android рассматривать панель инструментов так, словно она является стандартной панелью приложения, и вывести на ней имя приложения.

Этот код будет добавлен в метод `onCreate()` активности `MainActivity`, чтобы он выполнялся сразу же после создания активности. Обновите код *MainActivity.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.catchat

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.google.android.material.appbar.MaterialToolbar

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
    }
}
```

В коде используется класс MaterialToolbar.

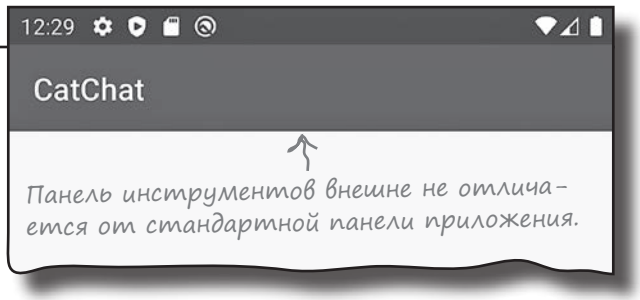
Используйте панель инструментов как панель приложений по умолчанию.

Вот и все, что необходимо сделать для замены панели приложения панелью инструментов. Давайте проведем тест-драйв приложения и убедимся в том, что имя выводится правильно.

Тест-драйв

Если запустить приложение, `MainActivity` отображает панель инструментов у верхнего края экрана. Панель инструментов включает имя приложения.

После того как стандартная панель приложения была заменена панелью инструментов, дополним ее новой функциональностью.



Использование панели инструментов для навигации

- [x] Панель инструментов
- [ ] **Панель инструментов навигации**
- [ ] Нижняя панель
- [ ] Выдвижная панель

В начале этой главы мы сказали, что наше приложение должно обеспечивать переходы к разным экранам приложения CatChat с использованием навигационных UI-компонентов Android. После добавления панели инструментов в приложение создадим пару экранов и воспользуемся панелью инструментов для перемещения между ними.

Как работает навигация с панели инструментов

Начнем с создания двух новых фрагментов — `InboxFragment` и `HelpFragment`, которые будут отображаться в `MainActivity` при переходе к ним пользователя. Фрагмент `InboxFragment` будет отображаться при запуске приложения, а `HelpFragment` — при переходе к нему.

Мы не будем добавлять функциональность справки или работы с электронной почтой в InboxFragment и HelpFragment. В каждом фрагменте будет выводиться блок текста, чтобы вы знали, какой фрагмент отображается в настоящее время.

Для перехода к `HelpFragment` мы добавим на панель инструментов элемент Help, чтобы панель выглядела так:

Элемент Help, который мы поместим на панель инструментов.

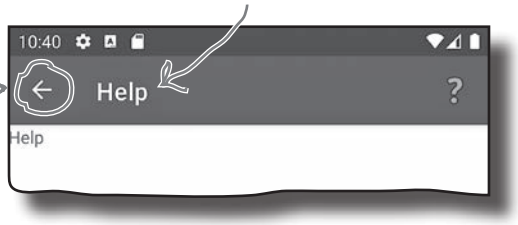
InboxFragment отображается при запуске приложения.

Для добавления элемента Help мы воспользуемся меню — как это делается, вы узнаете через несколько страниц.

При щелчке на элементе Help приложение будет переходить к `HelpFragment`. Мы также обновим заголовок панели инструментов, чтобы в ней выводилось название текущего экрана, и добавим кнопку Up для упрощения возврата к `InboxFragment`.

Мы обновим текст на панели инструментов, чтобы он показывал, что в настоящее время отображается фрагмент HelpFragment.

Эта кнопка (Up) будет возвращаться к InboxFragment.



Так работает навигация с панели инструментов. Начнем с ее реализации, для чего в приложение будут добавлены фрагменты `InboxFragment` и `HelpFragment`.

Часто Задаваемые Вопросы

В: Почему метод, используемый для назначения панели приложения, называется **`setSupportActionBar()`**? Разве он не должен называться **`setSupportAppBar()`**?

О: В ранних версиях Android панель приложения (app bar) называлась панелью действий (action bar), и под капотом стандартная панель приложения использует класс `ActionBar`. Именно поэтому метод называется `setSupportActionBar`.

В: **Стили могут использоваться только для переопределения стандартных цветов?**

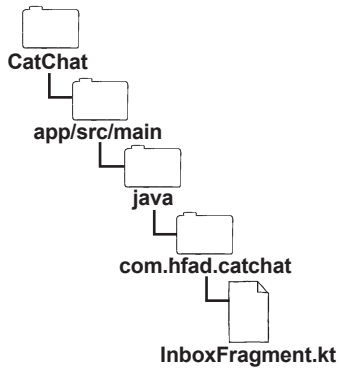
О: Нет, с их помощью можно переопределить любые свойства темы. Например, чтобы текст на всех кнопках не выводился в верхнем регистре, можно добавить элемент с именем `"android:textAllCaps"` со значением `false`.

- [x] Панель инструментов
- [ ] **Панель инструментов навигации**
- [ ] Нижняя панель
- [ ] Выдвижная панель

Создание InboxFragment

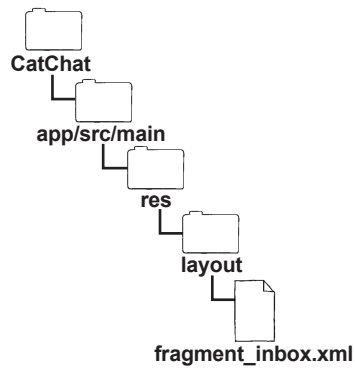
При запуске приложения в `MainActivity` должен отображаться фрагмент `InboxFragment`. Выделите пакет *com.hfad.catchat* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «InboxFragment» и имя макета «fragment_inbox»; убедитесь в том, что выбран язык Kotlin. Затем обновите код *InboxFragment.kt* и приведите его к следующему виду:

Так выглядит InboxFragment.




```
package com.hfad.catchat

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class InboxFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_inbox, container, false)
    }
}
```

А это код фрагмента *fragment_inbox.xml* (обновите код и в этом файле):

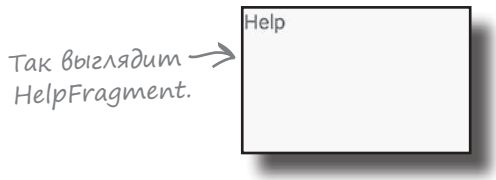


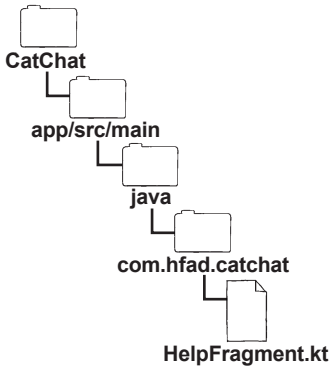

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:context=".InboxFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Inbox" />
</FrameLayout>
```

Макет InboxFragment содержит только TextView. Мы добавляем этот текст, чтобы можно было легко определить, какой фрагмент отображается на экране.

Создание HelpFragment

Фрагмент `HelpFragment` отображается в `MainActivity`, когда пользователь щелкает на элементе Help панели инструментов. Выделите пакет *com.hfad.catchat* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «HelpFragment» с именем макета «fragment_help» и убедитесь в том, что выбран язык Kotlin. Код *HelpFragment.kt* должен выглядеть так:



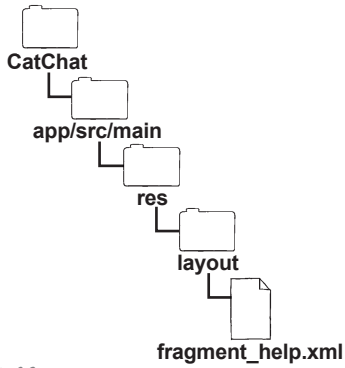



```
package com.hfad.catchat

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class HelpFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_help, container, false)
    }
}
```

Файл *fragment_help.xml* должен содержать следующую разметку:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".HelpFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Help" />
</FrameLayout>
```

В реальном приложении этот текст будет храниться в строковом ресурсе.



- [x] Панель инструментов
- [ ] Панель инструментов навигации
- [ ] Нижняя панель
- [ ] Выдвижная панель

Использование компонента Navigation для перехода к HelpFragment

В главе 6 вы научились перемещаться между фрагментами при помощи Android-компонента Navigation. И хотя для перехода к новой цели по-прежнему будет использоваться панель инструментов, для всех задач навигации все равно можно пользоваться компонентом Navigation.

Сначала необходимо добавить компонент Navigation в проект CatChat средствами Gradle.

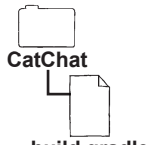
Добавление номера версии в файл проекта build.gradle

Как и прежде, необходимо добавить в файл *build.gradle* проекта новую переменную с версией компонента Navigation, который должен быть добавлен в приложение.

Откройте файл *CatChat/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `buildscript`:

```
buildscript {
    ext.nav_version = '2.3.5'
    ...
}
```

Будет использоваться эта версия компонента Navigation.



Добавление зависимостей в файл проекта build.gradle

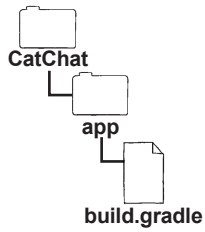
Затем необходимо добавить две зависимости компонента Navigation в файл *build.gradle* приложения.

Откройте файл *CatChat/app/build.gradle* и добавьте две строки (выделенные жирным шрифтом) в раздел `dependencies`:

Эти две строки добавляются в файл build.gradle приложения.

```
dependencies {
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    ...
}
```

Дополнительная зависимость для навигационных UI-компонентов.



После внесения этих изменений щелкните на ссылке Sync Now, появляющейся в верхней части редактора кода, чтобы синхронизировать изменения с оставшейся частью проекта.

После того как компонент Navigation будет добавлен в проект, мы воспользуемся им для создания графа навигации — после следующего упражнения.

Часто Задаваемые Вопросы

В: Мне придется добавлять компонент Navigation в каждое приложение, использующее навигацию? Неужели нет более удобного способа?

О: Компонент Navigation не включается в основные библиотеки Android, поэтому вам придется включить его в свой проект, прежде чем вы сможете им пользоваться.

Некоторые версии Android Studio автоматически добавляют компонент Navigation при создании графа навигации, но они могут использовать другую версию. Мы добавляем компонент вручную, чтобы вы знали, где он добавляется и какая версия используется в приложении.

Для чего я существую?

Мы собираемся воспользоваться компонентом Navigation для перехода между фрагментами приложения CatChat, но помните ли вы, что делает каждая часть компонента Navigation?

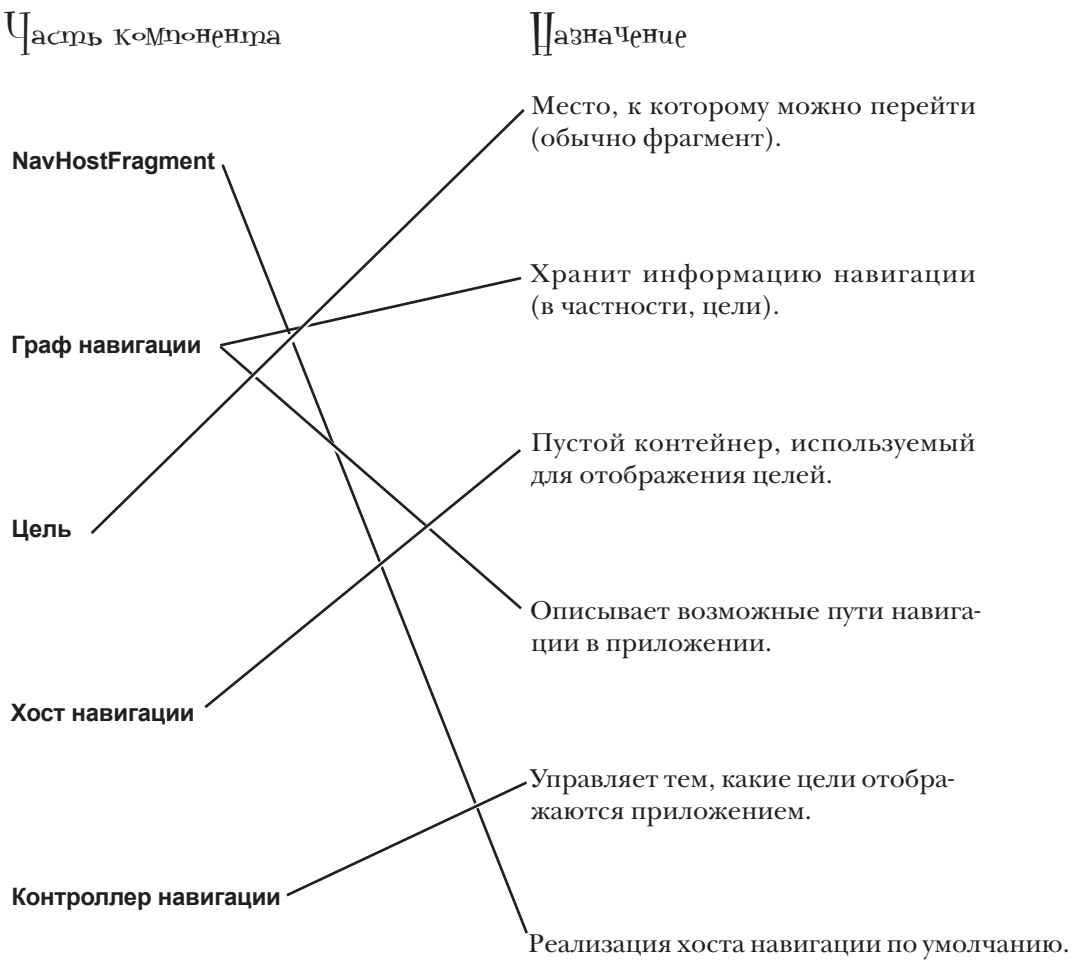
Соедините линией каждую часть с ее предназначением. Подсказка: у некоторых частей может быть несколько предназначений.

Часть компонента	Назначение
NavHostFragment	Место, к которому можно перейти (обычно фрагмент).
Граф навигации	Хранит информацию навигации (в частности, цели).
Цель	Пустой контейнер, используемый для отображения целей.
Хост навигации	Описывает возможные пути навигации в приложении.
Контроллер навигации	Управляет тем, какие цели отображаются приложением.
	Реализация хоста навигации по умолчанию.

Для чего я существую? Решение

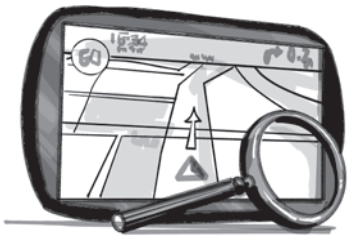
Мы собираемся воспользоваться компонентом Navigation для перехода между фрагментами приложения CatChat, но помните ли вы, что делает каждая часть компонента Navigation?

Соедините линией каждую часть с ее предназначением. Подсказка: у некоторых частей может быть несколько предназначений.



Добавление фрагментов в граф навигации

Как вы уже знаете, граф навигации вашего приложения содержит подробную информацию о целях вашего приложения и возможных путях, по которым к ним можно перейти. В приложении CatChat фрагменты `InboxFragment` и `HelpFragment` являются возможными целями, поэтому мы создадим новый граф навигации и добавим в него два фрагмента.



Граф навигации

Чтобы создать граф навигации, выделите папку *CatChat/app/src/main/res* на панели проекта, а затем выберите команду File→New→Android Resource File. Введите имя файла «nav_graph», выберите тип ресурса «Navigation» и щелкните на кнопке OK.

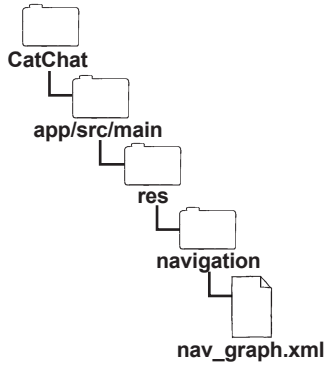
Затем откройте граф навигации (файл *nav_graph.xml*), переключитесь в режим Code и обновите файл, чтобы он соответствовал приведенному ниже коду:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/inboxFragment">

    <fragment
        android:id="@+id/inboxFragment"
        android:name="com.hfad.catchat.InboxFragment"
        android:label="Inbox"
        tools:layout="@layout/fragment_inbox" />
    <fragment
        android:id="@+id/helpFragment"
        android:name="com.hfad.catchat.HelpFragment"
        android:label="Help"
        tools:layout="@layout/fragment_help" />
</navigation>
```

Эта строка означает, что InboxFragment – первая отображаемая цель.

InboxFragment присваивается идентификатор inboxFragment и метка Inbox.

HelpFragment присваивается идентификатор helpFragment и метка Help.



Приведенный выше код добавляет фрагменты `InboxFragment` и `HelpFragment` в граф навигации и назначает каждому из них метку, удобную для пользователя.

И это все изменения, которые необходимо внести в граф навигации. Перейдем к обновлению макета `MainActivity`, чтобы он мог отображать каждый фрагмент, к которому вы переходите.

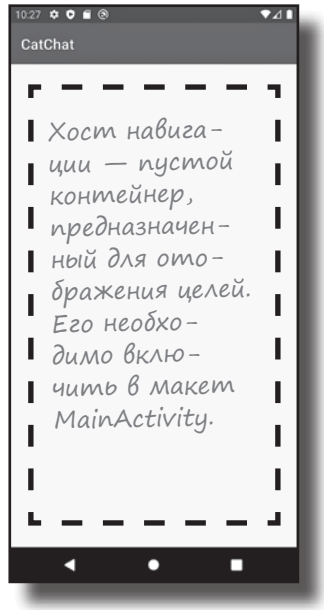
Добавление хоста навигации в activity_main.xml

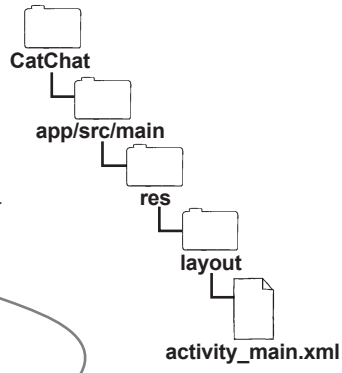
- [x] Панель инструментов
- [ ] **Панель инструментов навигации**
- [ ] Нижняя панель
- [ ] Выдвижная панель

Как вы узнали в главе 6, чтобы в приложении отображались цели, к которым вы переходите, следует включить в макет активности хост навигации. Для этого используется элемент `FragmentContainerView`, определяющий тип применяемого хоста навигации и имя графа навигации.

Код для решения этой задачи в приложении CatChat почти полностью совпадает с кодом, добавленным в приложение Secret Message в главе 6. Обновите код activity_main.xml и включите в него изменения, выделенные жирным шрифтом:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />
</LinearLayout>
```





Этот код добавляется в хост навигации. В качестве графа навигации он использует nav_graph.xml.

После добавления хоста навигации мы пополним панель инструментов элементом для перехода к `HelpFragment`.

Определение элементов панели инструментов в файле ресурсов меню

Чтобы указать Android, какие элементы должны размещаться на панели инструментов, следует определить **меню**. Каждое меню определяется в XML-файле ресурсов меню.

Мы создадим новый файл ресурсов меню с именем *menu_toolbar.xml*, который будет использоваться для добавления элемента Help на панель инструментов. Выделите папку *app/src/main/res*, откройте меню File, выберите команду New, а затем — вариант создания нового файла ресурсов Android. Введите имя «menu_toolbar», укажите тип ресурса «Menu» и убедитесь в том, что выбрано имя каталога menu. При нажатии кнопки OK среда Android Studio создает файл и добавляет его в папку *app/src/main/res/menu*.

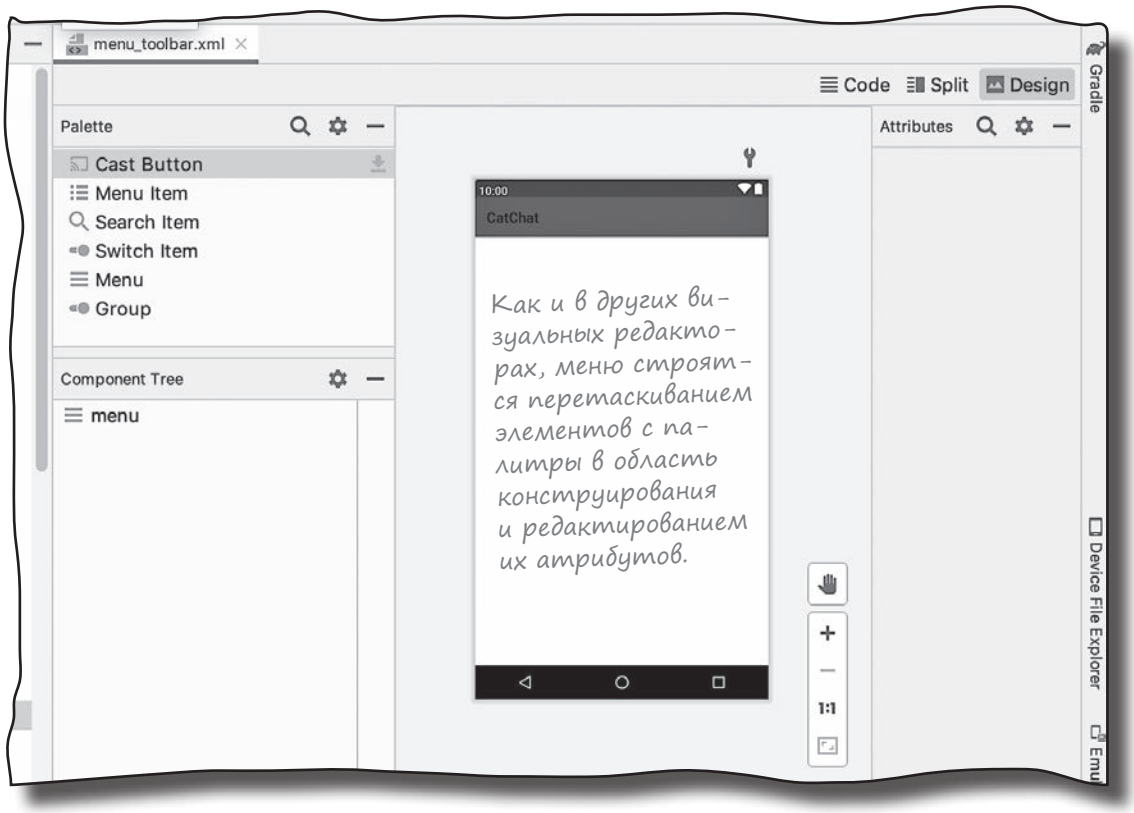
Как и в случае с графами навигации и файлами макетов, вы можете редактировать файлы ресурсов меню путем изменения кода XML или с помощью встроенного визуального редактора. Визуальный редактор выглядит так:

Часто Задаваемые Вопросы

В: Файлы ресурсов меню используются только панелями инструментов?

О: Как вы узнаете позднее в этой главе, файлы ресурсов меню также используются нижними панелями навигации и выдвижными панелями. Также они могут быть задействованы при создании всплывающих и контекстных меню. За дополнительной информацией обращайтесь по адресу

https://developer.android.com/guide/topics/ui/menus



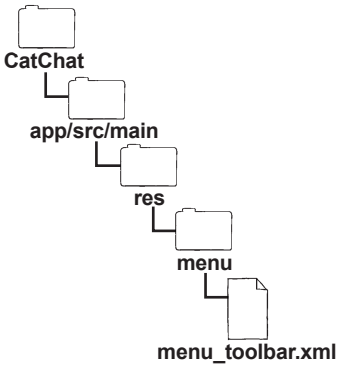
Добавление элемента Help в меню

Мы хотим иметь возможность перейти к фрагменту `HelpFragment` с панели инструментов, поэтому в файл ресурсов меню будет добавлен элемент Help. Это будет сделано прямым редактированием кода XML.

Переключитесь в режим Code для файла *menu_toolbar.xml* и обновите код (строки, которые нужно добавить, выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/helpFragment"
        android:icon="@android:drawable/ic_menu_help"
        android:title="Help"
        app:showAsAction="always" />
</menu>
```

Этот элемент определяет меню.



Этот раздел определяет элемент Help и отображает для него встроенный значок справки в Android.

Каждый файл ресурсов меню, включая приведенный выше, содержит корневой элемент **<menu>**. Он сообщает Android, что в файле определяется меню.

Внутри элемента `<menu>` обычно находится группа элементов `<item>`, каждый из которых описывает отдельный элемент меню. В нашем конкретном примере определяется один элемент Help с заголовком «Help».

Элемент `<item>` содержит ряд атрибутов, управляющих внешним видом элемента меню.

Атрибут android:id назначает идентификатор элемента. Идентификатор используется компонентом Navigation для перехода к цели; **он должен совпадать с идентификатором цели, к которой происходит переход на графе навигации**. Через несколько страниц вы увидите, как работает эта схема.

Атрибут `android:icon` указывает, какой значок должен отображаться для элемента (и должен ли). В Android есть много встроенных значков, и IDE выводит список доступных вариантов, когда вы начинаете вводить имя значка.

Атрибут `android:title` определяет текст элемента.

Атрибут `app:showAsAction` указывает, когда элемент должен появляться на панели инструментов. Значение «always» означает, что он всегда должен отображаться в основной области панели инструментов.

Идентификатор элемента меню должен совпадать с идентификатором цели на графе навигации.

UI-компоненты навигации, такие, как панель инструментов, зависят от совпадения идентификаторов. Если они не совпадают, навигация с панели инструментов будет работать некорректно.

- [x] Панель инструментов
- [ ] **Панель инструментов навигации**
- [ ] Нижняя панель
- [ ] Выдвижная панель

onCreateOptionsMenu() добавляет элементы меню на панель инструментов

После определения файла ресурсов меню необходимо добавить элементы на панель инструментов. Для этого в коде активности реализуется метод **`onCreateOptionsMenu()`**. Этот метод вызывается в тот момент, когда активность готова к добавлению элементов на панель инструментов. Он заполняет файл ресурсов меню и добавляет каждый описанный в нем элемент на панель инструментов.

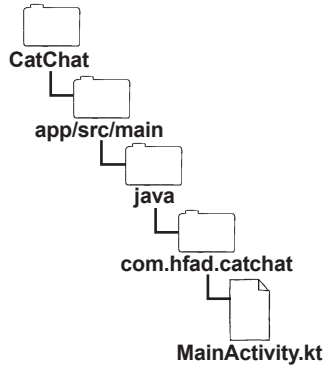
В приложении CatChat элемент, определенный в *menu_toolbar.xml*, должен быть добавлен на панель инструментов `MainActivity`. Код добавления элемента выделен ниже жирным шрифтом (мы обновим его через несколько страниц):

```
package com.hfad.catchat

...
import android.view.Menu

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }
}
```

Метод onCreateOptionsMenu() использует класс Menu.

Реализация этого метода добавляет на панель инструментов элементы из файла ресурсов меню.

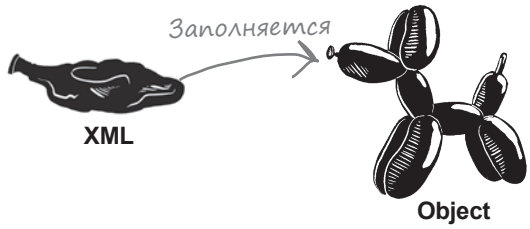
Имя файла ресурсов меню.



Строка

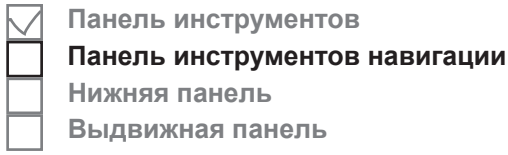
```
menuInflater.inflate(R.menu.menu_toolbar, menu)
```

заполняет файл ресурсов меню. Во внутренней реализации она создает объект `Menu`, представляющий файл ресурсов меню, а все элементы, содержащиеся в файле ресурсов меню, преобразуются в объекты `MenuItem`, которые затем добавляются на панель инструментов.



Вот и все, что необходимо сделать для добавления меню на панель инструментов. Теперь нужно сделать так, чтобы активность `MainActivity` переходила к `HelpFragment`, когда пользователь щелкает на элементе Help.

Реакция на щелчки на командах в onOptionsItemSelected()



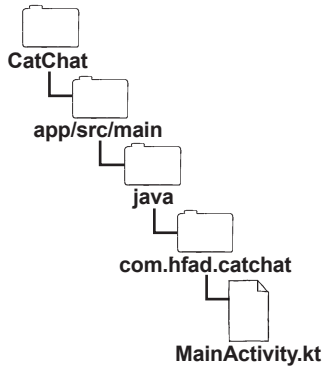
После того как элементы меню будут добавлены на панель инструментов методом `onCreateOptionsMenu()`, необходимо обеспечить их реакцию на щелчки. Для этого следует реализовать метод **`onOptionsItemSelected()`**. Этот метод выполняется каждый раз, когда на элементе панели инструментов делается щелчок:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    //Код, выполняемый при щелчке на элементе
}
```

Ссылка на элемент, на котором был сделан щелчок.

Приложение CatChat должно переходить к `HelpFragment` каждый раз, когда пользователь щелкает на элементе меню Help. Компонент Navigation поможет нам с решением этой задачи.

Ниже показано, как выглядит код; эти изменения будут добавлены в *MainActivity.kt* через несколько страниц:




```
package com.hfad.catchat

...
import android.view.MenuItem
import androidx.navigation.findNavController
import androidx.navigation.ui.onNavDestinationSelected

class MainActivity : AppCompatActivity() {
    ...

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

Эти дополнительные классы используются в коде.

Когда пользователь щелкает на элементе панели инструментов, контроллер навигации переходит к правильной цели.

Так выглядят многие реализации onOptionsItemSelected.

Каждый раз, когда пользователь щелкает на элементе меню Help, контроллер навигации получает его идентификатор и ищет совпадающий идентификатор в графе навигации. Затем цель с таким идентификатором передается хосту навигации, чтобы он был отображен на экране устройства.

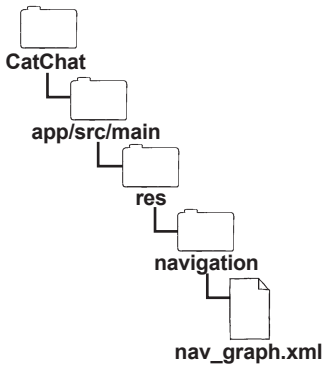
Настройка панели инструментов

Мы рассмотрели все, что необходимо знать для добавления элемента меню Help на панель инструментов и настройки перехода от него к `HelpFragment`. Но прежде чем опробовать приложение на практике, нужно внести еще одно изменение.

Когда мы определяли граф навигации *nav_graph.xml* ранее в этой главе, для каждой цели была добавлена метка:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
    <fragment
        android:id="@+id/inboxFragment"
        android:name="com.hfad.catchat.InboxFragment"
        android:label="Inbox"
        tools:layout="@layout/fragment_inbox" />
    <fragment
        android:id="@+id/helpFragment"
        android:name="com.hfad.catchat.HelpFragment"
        android:label="Help"
        tools:layout="@layout/fragment_help" />
</navigation>
```

Этот граф навигации был создан ранее, здесь ничего нового.



Для понимания, какой экран отображается в настоящий момент, мы настроим панель инструментов, чтобы при переходе к новой цели ее метка отображалась на панели инструментов. Также на панель инструментов будет добавлена кнопка Up, чтобы после перехода к `HelpFragment` пользователь мог легко вернуться к `InboxFragment`:

Так выглядит панель при запуске приложения.

Когда вы переходите к HelpFragment, на панели приложения должен отображаться текст «Help» и кнопка Up.

Кнопка Up.

Может показаться, что кнопка Up делает то же, что и кнопка Back на устройстве, но это не совсем так. Кнопка *Back* позволяет пользователю пройти весь путь по стеку возврата, то есть историю посещенных им экранов. С другой стороны, кнопка *Up* работает на основании иерархии графа навигации. Она предоставляет средства для быстрого перехода вверх по иерархии.

Панель инструментов можно настроить, чтобы текст на ней обновлялся при смене фрагмента, и добавить кнопку Up при помощи компонента Navigation. Давайте посмотрим, как это делается.

Настройка панели инструментов с использованием AppBarConfiguration

Панель инструментов требуется настроить так, чтобы текст на ней содержал метку текущей цели в графе навигации и чтобы на ней выводилась кнопка Up. Для этого можно построить объект **`AppBarConfiguration`**, основанный на графе навигации, и связать его с панелью инструментов. Класс `AppBarConfiguration` является частью компонента Navigation; с его помощью можно организовать взаимодействие панелей приложений и панелей инструментов с контроллером навигации.

Ниже приведен код построения `AppBarConfiguration` и связывания объекта с панелью инструментов (выделено жирным шрифтом):

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
    setSupportActionBar(toolbar)
    val navHostFragment = supportFragmentManager
        .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
    val navController = navHostFragment.navController
    val builder = AppBarConfiguration.Builder(navController.graph)
    val appBarConfiguration = builder.build()
    toolbar.setupWithNavController(navController, appBarConfiguration)
}
```

Эти строки получают ссылку на контроллер навигации из хоста навигации.

Строит конфигурацию, связывающую панель инструментов с графом навигации.

Эта строка применяет конфигурацию к панели инструментов.

Приведенный выше код сначала получает ссылку на контроллер навигации из хоста навигации, для чего используется следующий код:

```
val navHostFragment = supportFragmentManager
    .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val navController = navHostFragment.navController
```

Используем этот код, потому что вызов findNavController() из метода onCreate() активности может завершиться неудачей.

Такой код приходится использовать везде, где вам нужно получить ссылку на контроллер навигации из метода `onCreate()` активности.

Затем в коде строится объект конфигурации, связывающий панель инструментов с графом навигации, и этот объект применяется к панели инструментов.

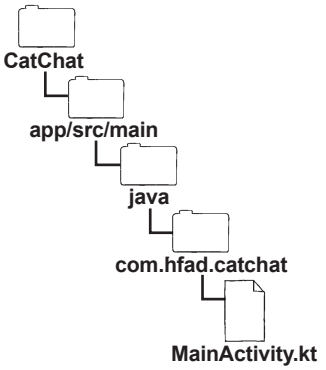
При выполнении кода информация из графа навигации используется для отображения метки текущей цели. Также на панель инструментов добавляется кнопка Up для всех целей, кроме стартовой цели графа навигации, которой в нашем случае является `InboxFragment`.

Вот и все, что необходимо знать для реализации навигации с панели инструментов. Полный код `MainActivity` приведен на следующей странице.

- [x] Панель инструментов
- [ ] **Панель инструментов навигации**
- [ ] Нижняя панель
- [ ] Выдвижная панель

Полный код MainActivity.kt

Ниже приведен полный код *MainActivity.kt*; обновите свой код (изменения выделены жирным шрифтом):




```
package com.hfad.catchat

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.navigation.findNavController
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.onNavDestinationSelected
import androidx.navigation.ui.setupWithNavController
import com.google.android.material.appbar.MaterialToolbar

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val navHostFragment = supportFragmentManager
            .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController
        val builder = AppBarConfiguration.Builder(navController.graph)
        val appBarConfiguration = builder.build()
        toolbar.setupWithNavController(navController, appBarConfiguration)
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

Импортируем все дополнительные классы.

Настраиваем панель инструментов, чтобы на ней размещалась кнопка Up и метка фрагмента, к которому вы перешли.

Все элементы меню добавляются на панель инструментов (в данном случае элемент Help).

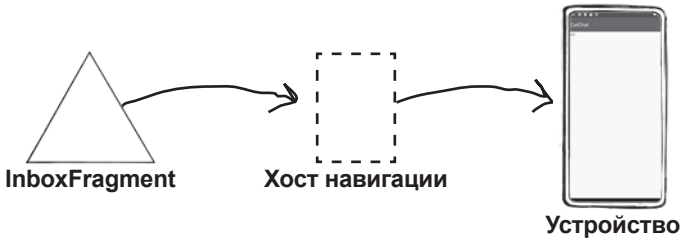
При щелчке на элементе происходит переход к цели.

- [x] Панель инструментов
- [ ] Панель инструментов навигации
- [ ] Нижняя панель
- [ ] Выдвижная панель

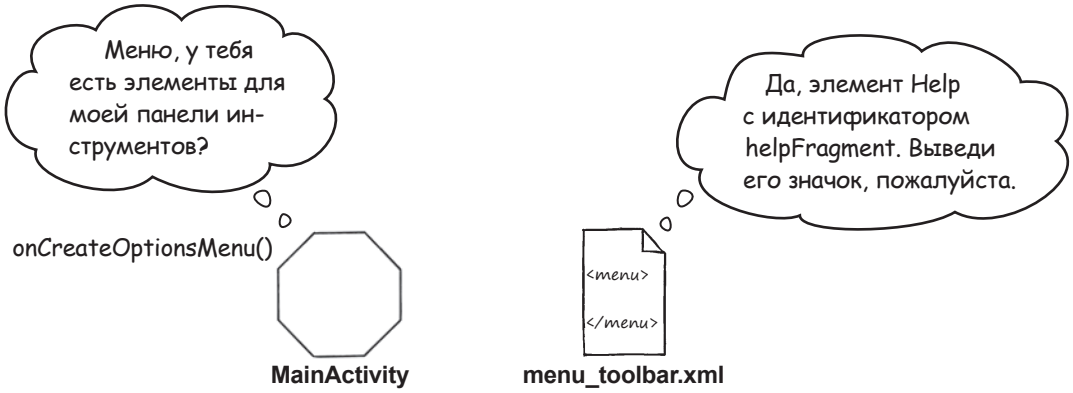
Что происходит во время запуска приложения

Происходит следующее:

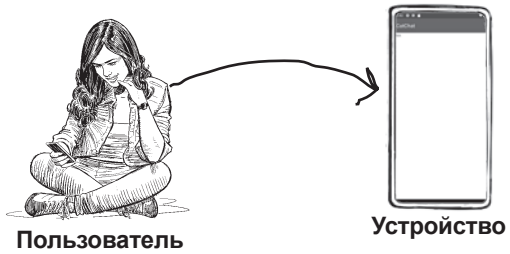
1 **Приложение запускается, создается активность MainActivity.**
`InboxFragment` добавляется в хост навигации и отображается на экране устройства.



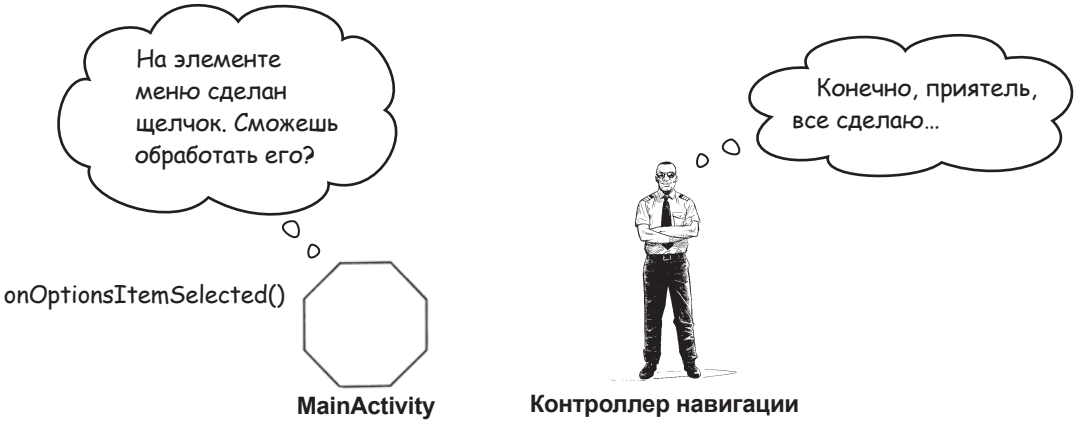
2 **Выполняется метод onCreateOptionsMenu активности MainActivity.**
На панель инструментов добавляется элемент меню Help, определенный в *menu_toolbar.xml*.



3 **Пользователь щелкает на элементе Item панели инструментов.**

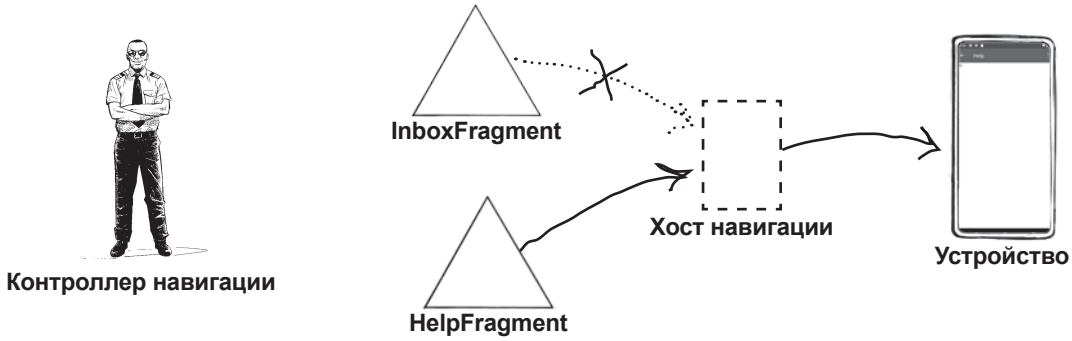


4 **Выполняется метод onOptionsItemSelected() активности MainActivity.**
Он передает навигацию элемента Help контроллеру навигации.



5 **Контроллер навигации обращается к идентификатору элемента Help в графе навигации.**

6 **Контроллер навигации заменяет InboxFragment фрагментом HelpFragment в хосте навигации.**



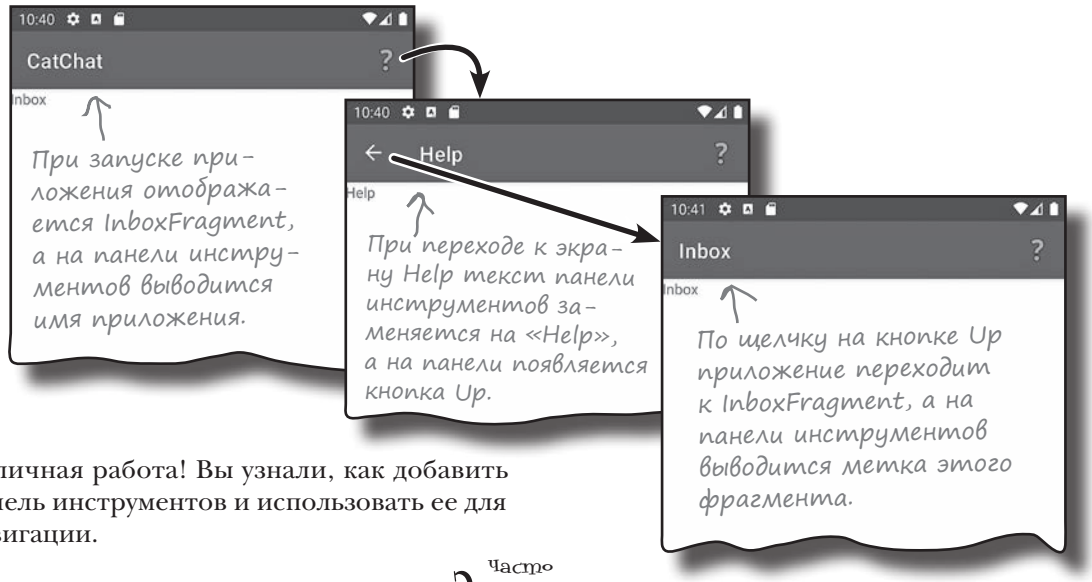
Тест-драйв

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [ ] Нижняя панель
- [ ] Выдвижная панель

При запуске приложения CatChat запускается активность `MainActivity` и в макете `MainActivity` отображается `InboxFragment`.

Элемент Help отображается на панели инструментов `MainActivity`. Если щелкнуть на нем, приложение переходит к `HelpFragment`. Текст панели инструментов заменяется текстом «Help», и появляется кнопка Up.

По щелчку на кнопке Up приложение переходит к `InboxFragment`. Кнопка Up исчезает, и на панели инструментов появляется текст «Inbox».



Отличная работа! Вы узнали, как добавить панель инструментов и использовать ее для навигации.

Часто Задаваемые Вопросы

В: Можно ли добавить элементы меню и кнопку Up со стандартной панелью приложения?

О: Да, но это потребует большего объема кода Kotlin. Например, для реализации кнопки Up необходимо переопределить отдельный метод с именем `onSupportNavigateUp()`.

В: Когда появляется кнопка Up?

О: Когда в хосте навигации отображается любой фрагмент, который не является стартовой целью в графе навигации. В нашем примере она появляется для любого фрагмента, кроме `InboxFragment`.

В: Можно ли добавить панель инструментов к фрагменту?

О: Да. Панель инструментов является разновидностью `View`, поэтому ее можно добавлять к активностям и фрагментам.

В: Как определить, добавлять ли панель инструментов к активности или фрагменту?

О: Если вы хотите, чтобы макет панели инструментов оставался неизменным для всех фрагментов, добавьте его к активности, отображающей эти фрагменты. Если макет панели инструментов должен заметно изменяться в зависимости от фрагмента, стоит подумать о добавлении ее к фрагменту.

СТАНЬ меню

Ниже приведен файл ресурсов меню и навигационный граф. Меню добавлено на панель инструментов. Представьте себя на месте меню и скажите, к какому фрагменту будет происходить переход при щелчке на каждом элементе меню.

Это файл ресурсов меню.

```
<menu ...>
    A <item
          android:id="@+id/homeFragment"
          android:title="Home" />
    B <item
          android:id="@+id/ordersFragment"
          android:title="Orders" />
    C <item
          android:id="@+id/accountFragment"
          android:title="Account" />
</menu>
```

К какому фрагменту происходит переход при щелчке на каждом из этих элементов?

Это граф навигации.

```
<navigation ...>
    <fragment
        android:id="@+id/startFragment"
        android:name="com.hfad.store.HomeFragment"
        android:label="Home" />
    <fragment
        android:id="@+id/homeFragment"
        android:name="com.hfad.store.StartFragment"
        android:label="Hello" />
    <fragment
        android:id="@+id/orderFragment"
        android:name="com.hfad.store.OrdersFragment"
        android:label="Orders" />
    <fragment
        android:id="@+id/accountFragment"
        android:name="com.hfad.store.YourAccountFragment"
        android:label="Your Account" />
</navigation>
```

СТАНЬ меню. Решение

Ниже приведен файл ресурсов меню и навигационный граф. Меню добавлено на панель инструментов. Представьте себя на месте меню и скажите, к какому фрагменту будет происходить переход при щелчке на каждом элементе меню.

Это файл ресурсов меню.

```
<menu ...>
    A <item
          android:id="@+id/homeFragment"
          android:title="Home" />
    B <item
          android:id="@+id/ordersFragment"
          android:title="Orders" />
    C <item
          android:id="@+id/accountFragment"
          android:title="Account" />
</menu>
```

Переходит к StartFragment, так как этой цели назначен идентификатор homeFragment.

Цели с совпадающим идентификатором не существует, поэтому этот элемент никуда не переходит.

Переходит к YourAccountFragment, так как этой цели назначен идентификатор accountFragment.

Это граф навигации.

```
<navigation ...>
    <fragment
        android:id="@+id/startFragment"
        android:name="com.hfad.store.HomeFragment"
        android:label="Home" />
    <fragment
        android:id="@+id/homeFragment"
        android:name="com.hfad.store.StartFragment"
        android:label="Hello" />
    <fragment
        android:id="@+id/orderFragment"
        android:name="com.hfad.store.OrdersFragment"
        android:label="Orders" />
    <fragment
        android:id="@+id/accountFragment"
        android:name="com.hfad.store.YourAccountFragment"
        android:label="Your Account" />
</navigation>
```

Многие разновидности UI-навигации работают с компонентом Navigation

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [ ] Нижняя панель
- [ ] Выдвижная панель

К текущему моменту вы научились включать навигацию с панели инструментов, определяя файл ресурсов меню, использовать компонент Navigation для перехода между фрагментами и настраивать панель инструментов для изменения ее внешнего вида при переходах в приложении.

К счастью, другие разновидности UI-навигации, такие как нижние панели и выдвижные панели, работают аналогичным образом. И хотя внешне они отличаются друг от друга, вы можете брать приемы, освоенные вами для навигации с панели инструментов, и применять их к другим типам навигации.

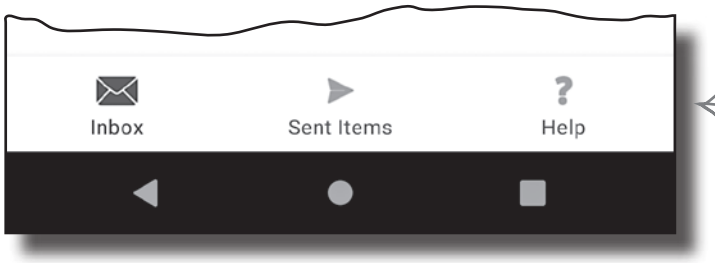
Чтобы вы поняли, как работают эти виды навигации, мы добавим нижнюю панель навигации в активность `MainActivity` приложения CatChat.

Нижняя панель навигации вмещает до пяти элементов. Она отображается у нижнего края экрана.

Как работает нижняя панель навигации

Как следует из названия, нижняя панель навигации — разновидность панели навигации, которая располагается в нижней части экрана устройства. На ней можно разместить элементы для переходов до пяти целей.

Нижняя панель навигации приложения CatChat выглядит так:



Нижняя панель навигации, которая будет добавлена в приложение CatChat. На ней размещаются три элемента: по одному для каждой цели, к которой вы сможете перейти.

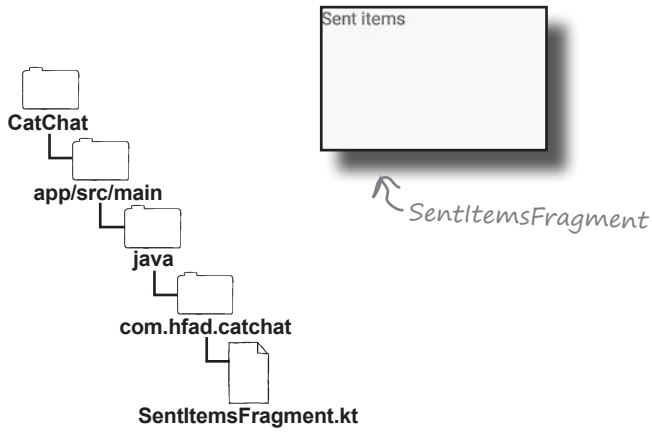
Панель содержит три элемента: Inbox, Sent Items и Help. При щелчке на каждом элементе приложение переходит к фрагменту, связанному с ним. Например, когда вы щелкаете на элементе Item, приложение переходит к `HelpFragment`, а когда вы щелкаете на Sent Items, оно перейдет к новому фрагменту (который мы еще не создали) с именем `SentItemsFragment`.

Нижняя панель будет реализована на нескольких ближайших страницах, а начнем мы с создания нового фрагмента: `SentItemsFragment`.

Создание SentItemsFragment

На следующем шаге мы создадим новый фрагмент с именем `SentItemsFragment`.

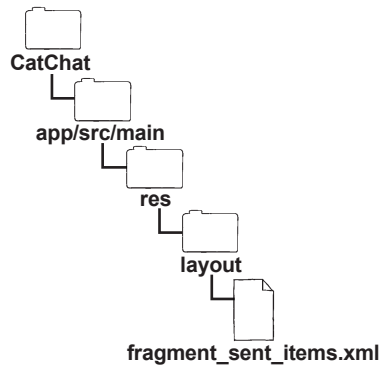
Выделите пакет *com.hfad.catchat* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «SentItemsFragment» и имя макета «fragment_sent_items»; убедитесь в том, что выбран язык Kotlin. Затем обновите код *SentItemsFragment.kt* и приведите его к следующему виду:




```
package com.hfad.catchat

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class SentItemsFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_sent_items, container, false)
    }
}
```

А это код фрагмента *fragment_sent_items.xml* (обновите код и в этом файле):

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SentItemsFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Sent Items" />
</FrameLayout>
```

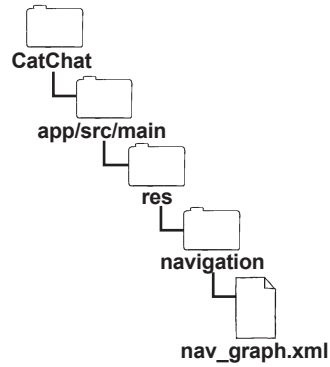


Включение фрагмента SentItemsFragment в граф навигации

Мы хотим иметь возможность перейти к `SentItemsFragment` с нижней панели навигации с использованием компонента Navigation. Для этого мы добавим фрагмент в граф навигации как новую цель.

Откройте файл графа навигации *nav_graph.xml* (если он не был открыт ранее) и обновите файл, чтобы он соответствовал приведенному ниже коду (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/inboxFragment">

    <fragment
        android:id="@+id/inboxFragment"
        android:name="com.hfad.catchat.InboxFragment"
        android:label="Inbox"
        tools:layout="@layout/fragment_inbox" />
    <fragment
        android:id="@+id/helpFragment"
        android:name="com.hfad.catchat.HelpFragment"
        android:label="Help"
        tools:layout="@layout/fragment_help" />
    <fragment
        android:id="@+id/sentItemsFragment"
        android:name="com.hfad.catchat.SentItemsFragment"
        android:label="Sent Items"
        tools:layout="@layout/fragment_sent_items" />
</navigation>
```



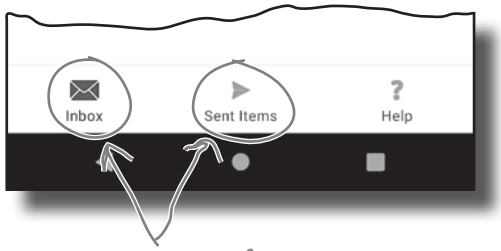
Приложение должно переходить к SentItemsFragment, поэтому фрагмент необходимо включить в граф навигации.

Как видите, новой цели присваивается идентификатор `sentItemsFragment` и метка «Sent Items». Идентификатор будет использоваться в файле ресурсов меню нижней панели навигации, который будет создан на следующем этапе.

Нижней панели навигации потребуется новый файл ресурсов меню

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [ ] **Нижняя панель**
- [ ] Выдвижная панель

Ранее в этой главе мы создали файл ресурсов меню с именем *menu_toolbar.xml* для добавления элемента Help на панель инструментов. Хотя одно меню может совместно использоваться разными UI-компонентами навигации, для нижней панели нам понадобится создать новое меню, потому что на ней будут отображаться два новых элемента: Inbox и Sent Items.



Нижняя панель навигации содержит дополнительные элементы, поэтому для нее нужно создать новый файл меню.

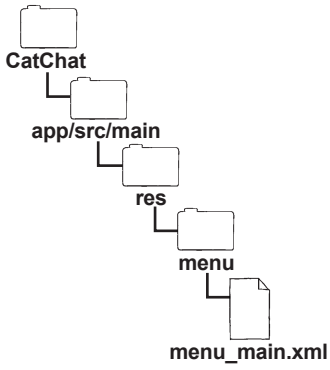
Чтобы создать новый файл ресурсов меню, выделите папку *app/src/main/res*, откройте меню File, выберите команду New, а затем выберите вариант создания нового файла ресурсов Android. Введите имя «menu_main», укажите тип ресурса «Menu» и убедитесь в том, что выбрано имя каталога menu. При нажатии кнопки OK среда Android Studio создает файл и добавляет его в папку *app/src/main/res/menu*.

Затем откройте файл *menu_main.xml* (если он не был открыт ранее) и обновите его код, чтобы добавить элементы для `InboxFragment`, `SentItemsFragment` и `HelpFragment` (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/inboxFragment"
        android:icon="@android:drawable/ic_dialog_email"
        android:title="Inbox" />
    <item
        android:id="@+id/sentItemsFragment"
        android:icon="@android:drawable/ic_menu_send"
        android:title="Sent Items" />
    <item
        android:id="@+id/helpFragment"
        android:icon="@android:drawable/ic_menu_help"
        android:title="Help" />

</menu>
```

Три элемента, которые мы добавляем на нижнюю панель инструментов.

Их идентификаторы должны совпадать с соответствующими идентификаторами в графе навигации.



После того как файл ресурсов меню будет создан, добавим нижнюю панель навигации в `MainActivity`.

Нижняя панель является разновидностью View

Нижняя панель, как и панель инструментов, является разновидностью представления, которое добавляется в макет. Код добавления нижней панели выглядит примерно так:

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottom_nav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:menu="@menu/menu_main" />
```

Определяет нижнюю панель, содержимое которой определяется файлом menu_main.xml.

Все начинается с определения нижней панели:

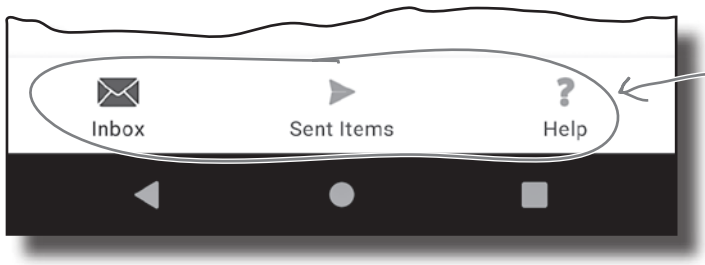
```
<com.google.android.material.bottomnavigation.BottomNavigationView
    ... />
```

где `com.google.android.material.bottomnavigation.BottomNavigationView` — полный путь к `BottomNavigationView` — классу, определяющему панель. Затем дополнительные атрибуты используются для определения его идентификатора и настройки его внешнего вида.

В отличие от панели инструментов, вам не придется писать код Kotlin для добавления элементов на нижнюю панель. Достаточно указать, какой файл ресурсов меню следует добавить на панель, при помощи атрибута `app:menu`:

```
app:menu="@menu/menu_main"
```

Приведенный выше код присоединяет файл ресурсов меню *menu_main.xml* к нижней панели, а его элементы добавляются на панель во время выполнения:



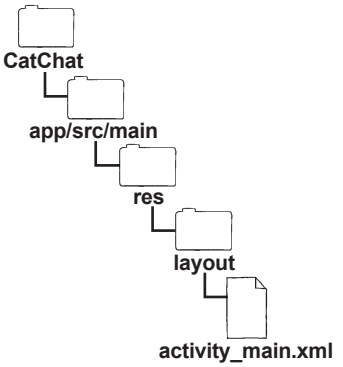
Эти три элемента добавлены в menu_main.xml. Нижняя панель навигации знает, что включить нужно элементы из этого меню, потому что атрибуту app:menu присвоено значение «@menu/menu_main».

Итак, теперь вы знаете, как выглядит код нижней панели навигации. Давайте добавим ее в макет `MainActivity`.

Полный код activity_main.xml

Ниже приведен полный код *activity_main.xml*: обновите свою версию (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent""0dp"
        android:layout_weight="1"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />

    <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/bottom_nav"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:menu="@menu/menu_main" />
</LinearLayout>
```



К хосту навигации добавляется атрибут layout_weight, чтобы панель навигации отображалась в нижней части экрана.

Нижняя панель навигации добавляется в макет.

После добавления нижней панели в макет `MainActivity` нужно сделать так, чтобы она выполняла перемещения между целями.

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [ ] **Нижняя панель**
- [ ] Выдвижная панель

Связывание нижней панели с контроллером навигации

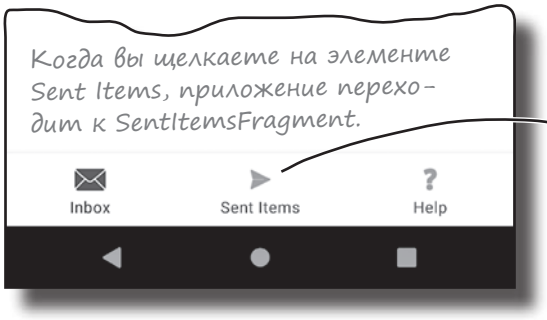
Код, обеспечивающий переходы между целями с нижней панели, проще кода, необходимого для реализации навигации с панели инструментов. Все, что для этого нужно, — получить ссылку на объект `BottomNavigationView`, определяющую панель навигации, и вызвать метод `setupWithNavController()`.

Код выглядит так:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    ...
    val navHostFragment = supportFragmentManager
        .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
    val navController = navHostFragment.navController
    val bottomNavView = findViewById<BottomNavigationView>(R.id.bottom_nav)
    bottomNavView.setupWithNavController(navController)
}
```

Это весь код активности, необходимый для реализации нижней панели. В нем панель связывается с контроллером навигации.

Каждый раз, когда вы щелкаете на элементе панели навигации, контроллер навигации получает его идентификатор и ищет цель с совпадающим идентификатором в графе навигации. Затем найденная цель передается хосту навигации, чтобы она отображалась на экране устройства.



Добавим код нижней панели навигации в *MainActivity.kt* и проведем тест-драйв приложения.

Обновленный код MainActivity.kt

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [ ] Нижняя панель
- [ ] Выдвижная панель

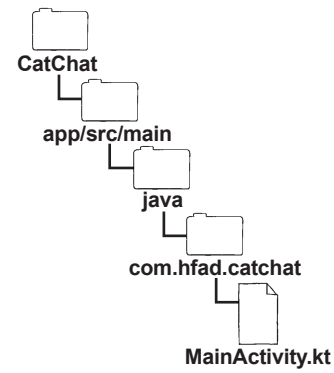
Ниже приведен обновленный код *MainActivity.kt*; включите в него изменения, выделенные жирным шрифтом:

```
package com.hfad.catchat

...
import com.google.android.material.bottomnavigation.BottomNavigationView

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val navHostFragment = supportFragmentManager
            .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController
        val builder = AppBarConfiguration.Builder(navController.graph)
        val appBarConfiguration = builder.build()
        toolbar.setupWithNavController(navController, appBarConfiguration)
        val bottomNavView = findViewById<BottomNavigationView>(R.id.bottom_nav)
        bottomNavView.setupWithNavController(navController)
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

Необходимо импортировать этот класс.

Нижняя панель навигации связывается с контроллером навигации.

Эти два метода необходимы только для меню панели инструментов. Для нижней панели навигации они не нужны.

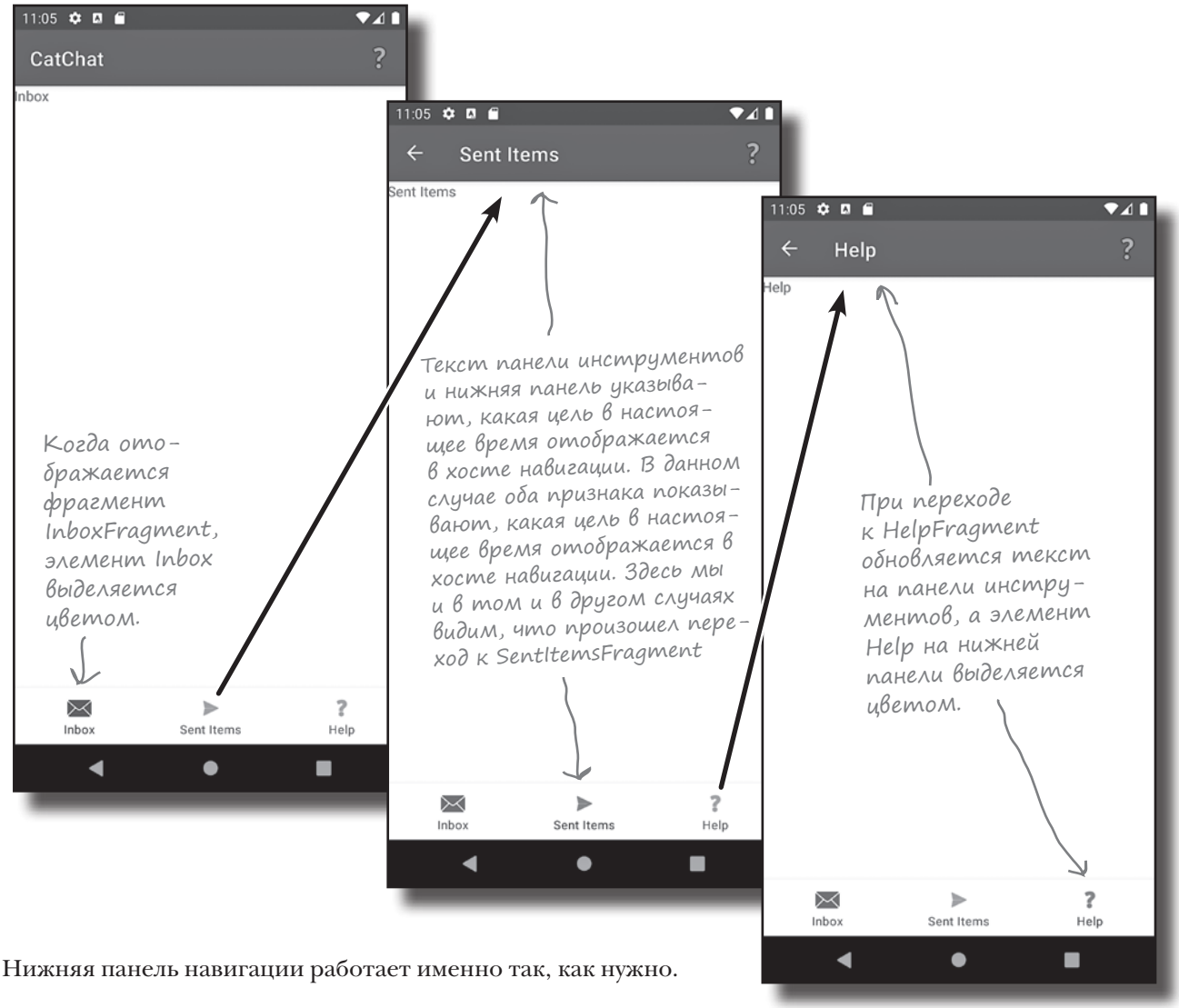


Проведем тест-драйв приложения.

Тест-драйв

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] **Нижняя панель**
- [ ] Выдвижная панель

При запуске приложения CatChat запускается активность `MainActivity`, а фрагмент `InboxFragment` отображается в макете `MainActivity`. На экране появляется нижняя панель с тремя элементами. Щелчок на каждом элементе нижней панели выполняет переход к соответствующему экрану. Например, если щелкнуть на Sent Items, отображается фрагмент `SentItemsFragment` в `MainActivity`, а если щелкнуть на Help, то вместо него отображается фрагмент `HelpFragment`.



Нижняя панель навигации работает именно так, как нужно.

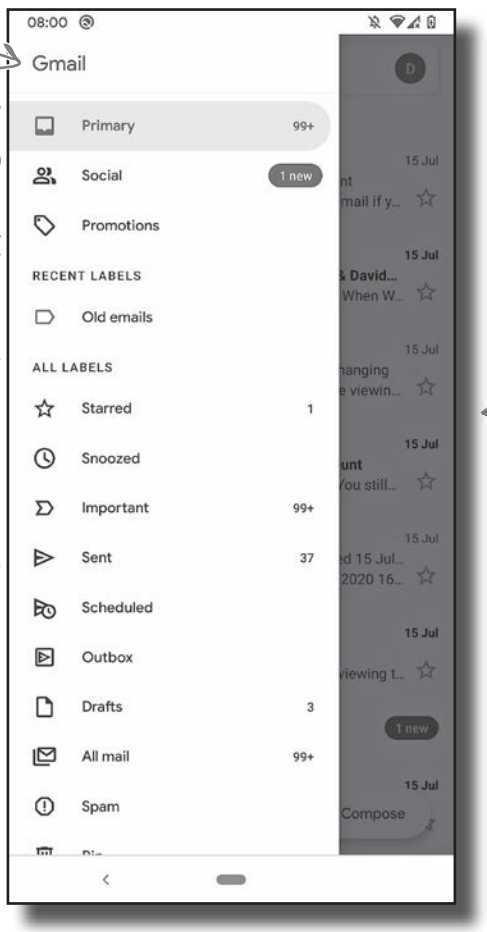
На выдвижной панели могут отображаться разные навигационные элементы

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] **Выдвижная панель** ←

Как вы уже узнали, нижняя панель навигации хорошо подходит для приложений с небольшим количеством элементов навигации, потому что на ней помещается до пяти таких элементов. Но что, если элементов меню должно быть больше?

Если вы хотите, чтобы у пользователя было много вариантов навигации, выдвижная панель может оказаться более эффективным решением. Это выдвижная панель с поддержкой прокрутки, содержащая ссылки на другие части приложения, которые можно группировать по разным разделам.

Выдвижные панели широко используются в приложениях Android. Например, в приложении Gmail используется выдвижная панель для перехода к разным экранам приложения, и она делится на такие секции, как категории электронной почты, недавно использованные ярлыки и все ярлыки:



В заголовке выдвижной панели выводится имя приложения.

Основные категории электронной почты выводятся в верхней части панели.

Элементы, на которых недавно выполнялись щелчки, выводятся в отдельной категории.

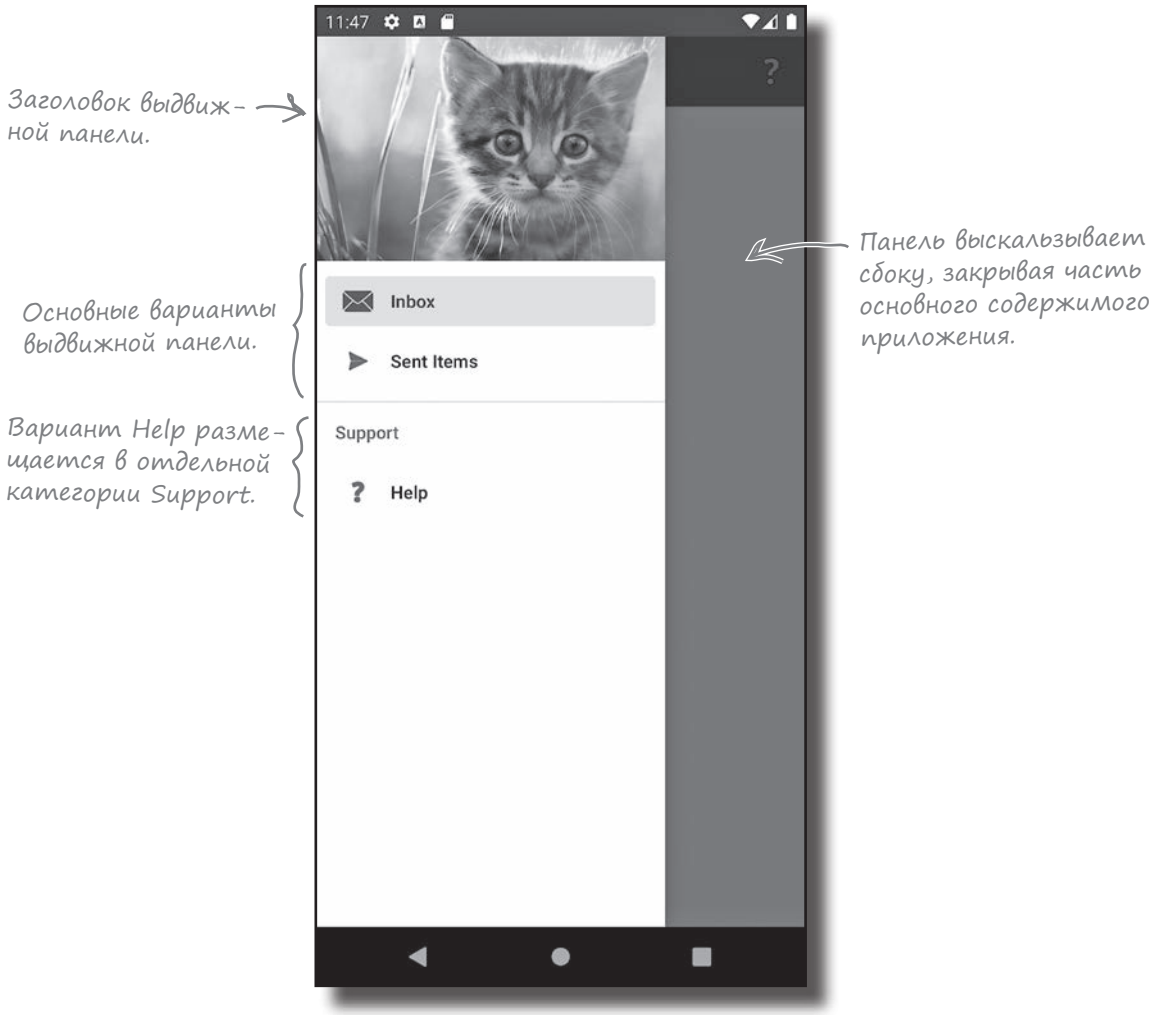
А потом идет длинный список всех категорий.

Это приложение Gmail. Оно содержит выдвижную панель навигации, которая появляется над основным содержимым приложения. Выдвижная панель предоставляет многочисленные варианты для перехода к разным частям приложения.

Когда вы щелкаете на элементе выдвижной панели, она закрывается, а здесь выводится содержимое, соответствующее выбранному варианту.

Замена нижней панели выдвижной панелью

Сейчас заменим нижнюю панель, добавленную в приложение CatChat, выдвижной панелью. Выдвижная панель содержит графический заголовок и набор вариантов. Основные варианты позволяют перейти к `InboxFragment` и `SentItemsFragment`, а элемент `HelpFragment` будет помещен в отдельную категорию Support. Вот как должна выглядеть панель:



Выдвижная панель состоит из нескольких компонентов, которые будут рассмотрены на следующей странице.

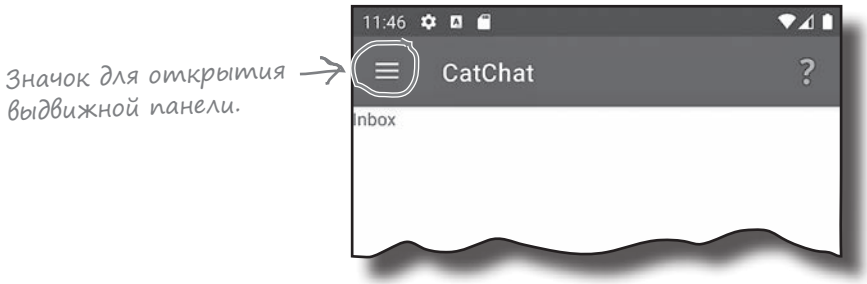
Как устроены выдвижные панели навигации

Выдвижная панель реализуется добавлением макета выдвижной панели в корень макета активности. Макет выдвижной панели содержит два представления:

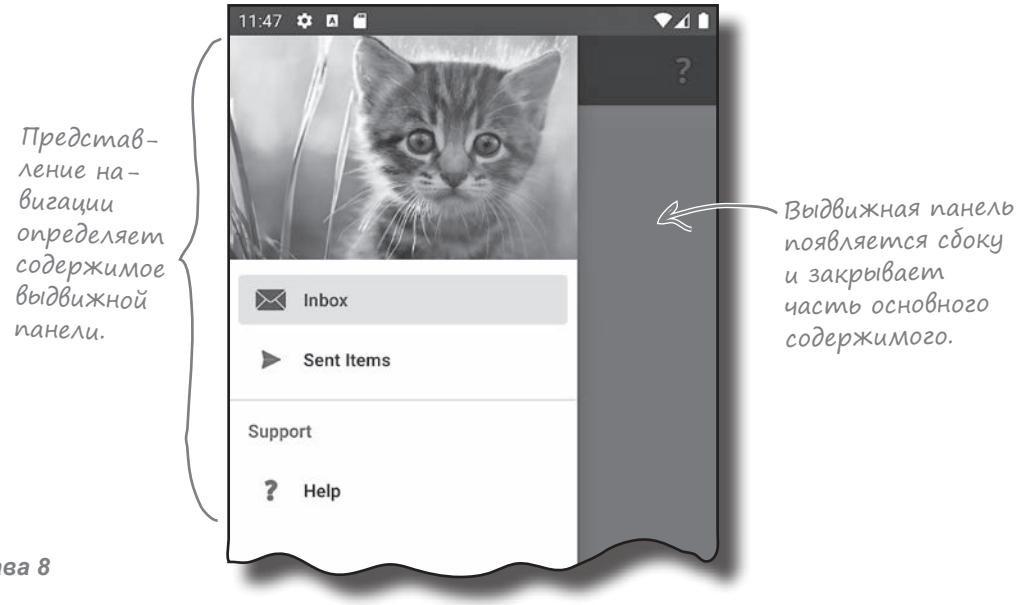
1. **Представление для вывода основного содержимого.** Обычно это макет с панелью инструментов и хостом навигации, используемый для отображения фрагментов.

2. **Представление для содержимого выдвижной панели.** Представление навигации — разновидность фреймового макета, которая используется для отображения меню навигации. В нашем приложении она тоже отображает заголовок выдвижной панели.

Когда выдвижная панель закрыта, макет выдвижной панели выглядит как обычная активность, не считая того, что на панели инструментов присутствует значок, который открывает выдвижную панель:



Когда вы открываете выдвижную панель, представление навигации появляется сбоку, закрывая часть основного содержимого. Когда вы щелкаете на элементе, компонент Navigation используется для отображения соответствующей цели, а выдвижная панель закрывается:



- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] **Выдвижная панель**

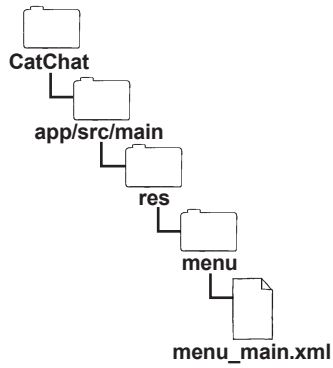
Выдвижная панель получает свои элементы из меню

Как и панель инструментов и нижняя панель навигации, выдвижная панель получает элементы, которые должны на ней выводиться, из файла ресурсов меню.

Вместо того чтобы создавать новый файл ресурсов меню для выдвижной панели, мы повторно используем menu_main.xml: файл, который использовался для нижней панели. Напомним, как выглядит текущая версия кода в *menu_main.xml*:

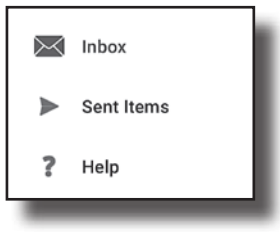
Тот же файл меню, который использовался для нижней панели.

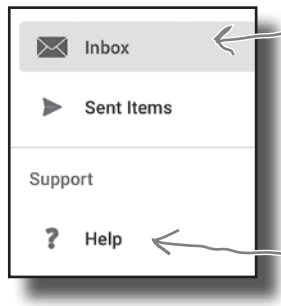
```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/inboxFragment"
        android:icon="@android:drawable/ic_dialog_email"
        android:title="Inbox" />
    <item
        android:id="@+id/sentItemsFragment"
        android:icon="@android:drawable/ic_menu_send"
        android:title="Sent Items" />
    <item
        android:id="@+id/helpFragment"
        android:icon="@android:drawable/ic_menu_help"
        android:title="Help" />
</menu>
```



Приведенный выше файл ресурсов меню, добавленный в выдвижную панель, создает список элементов, каждый из которых представлен значком. Мы настроим меню, чтобы текущий выделенный элемент выделялся цветом, а меню было разбито на категории:

Это меню выдвижной панели создается приведенным выше кодом.





Элемент, выбранный пользователем, будет выделяться цветом.

Элемент Help будет помещен в отдельную категорию Support.

Давайте посмотрим, как это делается.

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] **Выдвижная панель**

Добавление категории Support...

Начнем с добавления заголовка Support, для чего в меню добавляется новый элемент. Так как это всего лишь заголовок категории, достаточно задать текст: у элемента нет значка, и ему не нужно назначать идентификатор, так как он не будет использоваться для перехода.

Заголовок Support создается следующим кодом:

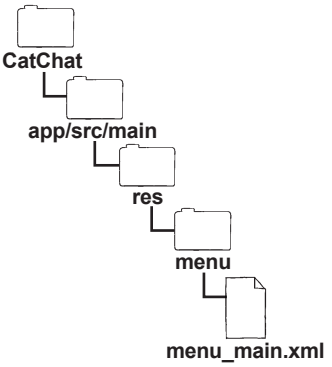
```
...
    <item android:title="Support">
    </item>
...
```

Добавляет на выдвижную панель заголовок Support. Код menu_main.xml будет обновлен через пару страниц.

...как отдельного подменю

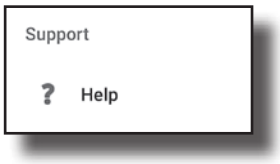
Элемент Help должен отображаться под заголовком Support, чтобы он образовал отдельную категорию. Для этого мы определим подменю внутри элемента Support, заданное элементом <menu>. В это подменю будет добавлен элемент Help.

Ниже приведен код добавления подменю с элементом Help; код *menu_main.xml* будет обновлен через пару страниц:




```
...
    <item android:title="Support">
        <menu>
            <item
                android:id="@+id/helpFragment"
                android:icon="@android:drawable/ic_menu_help"
                android:title="Help" />
        </menu>
    </item>
...
```

Приведенный выше код отображает элемент Help в категории Support.



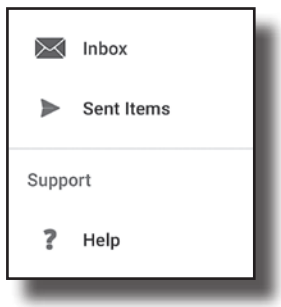
Подменю содержит всего один элемент. Если вы хотите, чтобы оно содержало несколько элементов, просто включите каждый элемент в подменю.

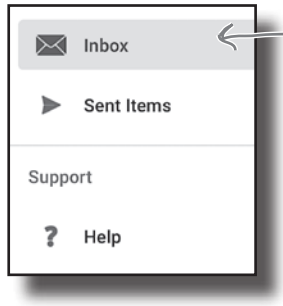
И это весь код, необходимый для добавления категории Support в меню выдвижной панели. Теперь можно сделать следующий шаг — добавить дополнительное выделение цветом текущего элемента, выбранного пользователем.

Выделение текущего элемента в группах

Текущее меню создает выдвижную панель навигации, которая изменяет цвет текста текущего выделенного элемента. Чтобы пользователь более четко видел, какой элемент выделен, можно добавить дополнительное выделение цветом:

Так выглядит меню без дополнительного выделения.





Дополнительное выделение более наглядно обозначает выделенный элемент.

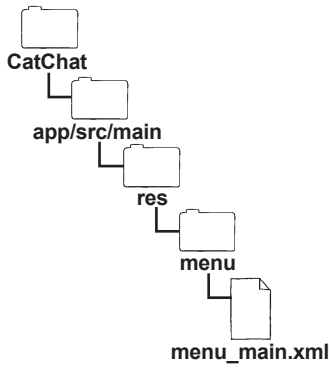
Чтобы добавить дополнительное выделение, следует включить элементы в группу при помощи элемента **<group>**. Затем атрибут `android:checkableBehavior` определяет поведение группы при выделении элемента.

Эта задача решается следующим кодом (код *menu_main.xml* будет обновлен через пару страниц):

Означает, что один элемент в группе будет выделяться цветом при щелчке на нем (элемент, выбранный пользователем).

```
...
<group android:checkableBehavior="single">
    <item
        android:id="@+id/inboxFragment"
        android:icon="@android:drawable/ic_dialog_email"
        android:title="Inbox" />
    <item
        android:id="@+id/sentItemsFragment"
        android:icon="@android:drawable/ic_menu_send"
        android:title="Sent Items" />
</group>
...
```

Эти элементы включаются в группу. Когда один из них выбирается пользователем, ему назначается дополнительное цветовое выделение на панели навигации.



Приведенный выше код присваивает атрибуту `android:checkableBehavior` значение «single». Это означает, что в любой момент времени выделенным может быть только один элемент в группе — элемент, выбранный пользователем.

Вот и все, что необходимо сделать, чтобы придать меню нужный внешний вид и поведение при использовании в качестве выдвижной панели навигации. Полный код приводится на следующей странице.

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] **Выдвижная панель**

Полный код menu_main.xml

Обновим меню выдвижной панели навигации, чтобы меню Help отображалось в категории Support. Также мы определим группы, чтобы добавить дополнительное цветовое выделение для элемента, выбранного пользователем.

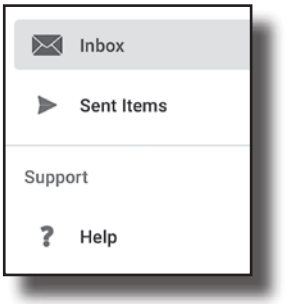
Ниже приведен полный код menu_main.xml; обновите его (изменения выделены жирным шрифтом):

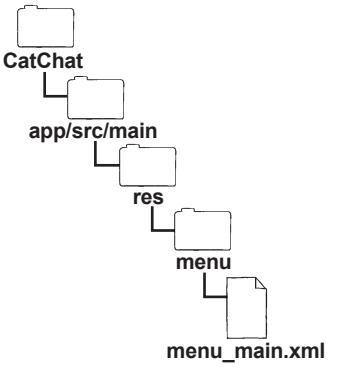
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/inboxFragment"
            android:icon="@android:drawable/ic_dialog_email"
            android:title="Inbox" />
        <item
            android:id="@+id/sentItemsFragment"
            android:icon="@android:drawable/ic_menu_send"
            android:title="Sent Items" />
    </group>

    <item android:title="Support">
        <menu>
            <group android:checkableBehavior="single">
                <item
                    android:id="@+id/helpFragment"
                    android:icon="@android:drawable/ic_menu_help"
                    android:title="Help" />
            </group>
        </menu>
    </item>
</menu>
```

Первые два элемента объединяются в группу.

Добавляем раздел Support.

Элемент Help включается в группу, чтобы при выборе он получал дополнительное выделение цветом.





С меню мы разобрались. Перейдем к созданию заголовка выдвижной панели.

Создание заголовка выдвижной панели

Заголовок выдвижной панели представляет собой простой макет, который будет размещен в новом файле макета с именем *nav_header.xml*.

Создайте этот файл: выделите папку *app/src/main/res/layout* в Android Studio и выберите команду File→New→Layout. Введите имя макета «nav_header», и если среда запросит тип ресурса, выберите Layout.

Добавление графического файла...

Макет состоит из одного графического изображения, которое необходимо разместить в папке *app/src/main/res/drawable*. Вероятно, среда Android Studio создала эту папку за вас при создании проекта. Если папка отсутствует, добавьте ее вручную: выделите папку *app/src/main/res*, откройте меню File, выберите команду New и щелкните на варианте создания нового каталога ресурсов Android. Выберите тип ресурса Drawable, введите имя «drawable» и щелкните на кнопке OK.

После того как папка drawable будет создана, загрузите файл *kitten_small.webp* по адресу *tinyurl.com/hfad3* и добавьте его в папку *drawable*.

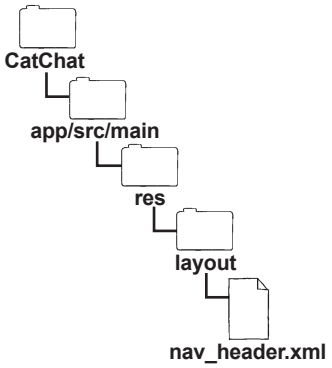
Изображение выводится в ImageView.

...и обновление кода nav_header.xml

Для добавления изображения в *nav_header.xml* будет использоваться элемент `<ImageView>`. Вы уже знаете, как пользоваться этим элементом, поэтому обновите код *nav_header.xml* и приведите его в соответствие со следующим кодом:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="180dp" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/kitten_small" />
</FrameLayout>
```

Макет нужен только для вывода изображения, поэтому мы используем представление ImageView внутри FrameLayout.



Итак, заголовок для выдвижной панели готов; давайте добавим саму панель.

Как создать выдвижную панель

Чтобы создать выдвижную панель навигации, добавьте макет DrawerLayout в макет активности как корневой элемент. Макет DrawerLayout должен содержать две составляющие: представление или группу представлений для содержимого активности (первый элемент) и представление NavigationView, определяющее содержимое выдвижной панели (второй элемент).

Типичный код `DrawerLayout` выглядит примерно так:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        ...
    </LinearLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/nav_header"
        app:menu="@menu/menu_main"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

Элемент DrawerLayout определяет макет, включающий выдвижную панель.

Макету присваивается идентификатор, чтобы к нему можно было обращаться из кода активности.

Первое представление в DrawerLayout — макет основного содержимого активности. Он отображается при закрытой выдвижной панели.

NavigationView — разновидность FrameLayout, предоставляющая стандартное меню навигации.

Присоединяет выдвижную панель к начальной стороне активности (левая сторона для языков с написанием слева направо).

Макет для заголовка выдвижной панели.

Файл ресурсов меню, содержащий элементы выдвижной панели.

Внешний вид выдвижной панели определяется двумя ключевыми атрибутами `<NavigationView>`: **`app:headerLayout`** и **`app:menu`**.

Атрибут `app:headerLayout` задает макет, который должен использоваться для заголовка выдвижной панели (в данном случае *nav_header.xml*). Этот атрибут не является обязательным.

Атрибут `app:menu` указывает, какой файл ресурсов меню содержит элементы выдвижной панели (в данном случае *menu_main.xml*). Если не включить этот атрибут, выдвижная панель не будет содержать никаких элементов.

Полный код activity_main.xml

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] **Выдвижная панель** ←

Требуется заменить нижнюю панель навигации активности `MainActivity` выдвижной панелью навигации. Код для решения этой задачи приведен ниже; обновите файл *activity_main.xml* (изменения выделены жирным шрифтом):

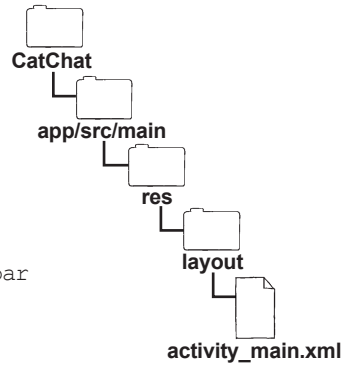
```
<?xml version="1.0" encoding="utf-8"?>
~~<LinearLayout~~
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ~~android:orientation="vertical"~~
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            style="@style/Widget.MaterialComponents.Toolbar.Primary" />

        <androidx.fragment.app.FragmentContainerView
            android:id="@+id/nav_host_fragment"
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            app:defaultNavHost="true"
            app:navGraph="@navigation/nav_graph" />
```

Корневой элемент макета заменяется элементом DrawerLayout.

Выдвижной панели назначается идентификатор, потому что он будет использоваться в коде активности.

Удалите эту строку.

Основное содержимое активности определяется линейным макетом. Как и прежде, в этот макет включается панель инструментов и хост навигации.



Продолжение на следующей странице. →

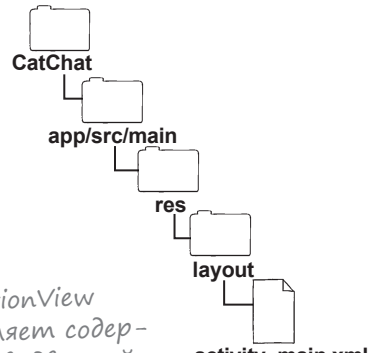
- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] Выдвижная панель

Полный код activity_main.xml (продолжение)

Нижняя панель навигации удаляется.

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottom_nav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:menu="@menu/menu_main" />
</LinearLayout>
```

Закройте линейный макет.




```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:headerLayout="@layout/nav_header"
    app:menu="@menu/menu_main"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

NavigationView определяет содержимое выдвижной панели.

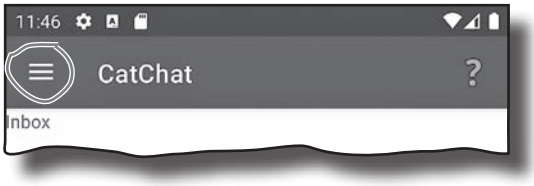
Выдвижная панель включает макет заголовка.

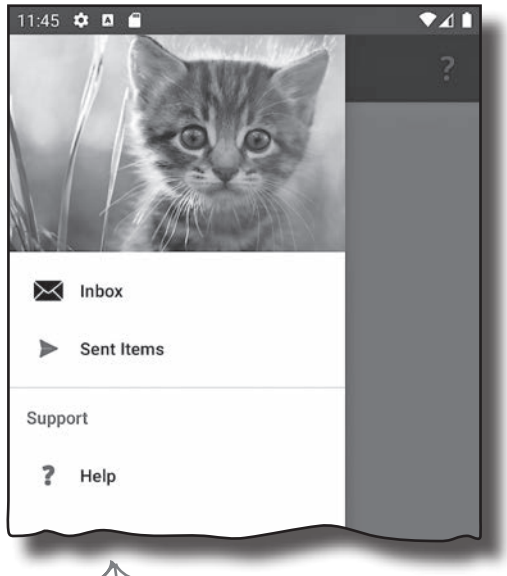
Выдвижная панель навигации добавлена в макет

Мы заменили нижнюю панель в макете `MainActivity` выдвижной панелью. На ней выводится графический заголовок и все элементы, заданные в файле ресурсов меню *menu_main.xml*.

Но прежде чем запускать приложение, необходимо связать выдвижную панель с контроллером навигации, чтобы щелчок на элементе вызывал переход к соответствующему фрагменту. Также необходимо настроить панель инструментов, чтобы на ней размещался значок, открывающий или закрывающий выдвижную панель навигации:

Значок выдвижной панели добавляется на панель инструментов.





Если не связать выдвижную панель с контроллером навигации, элементы будут отображаться на выдвижной панели, но по щелчку на них ничего не произойдет.

Оба изменения реализуются изменением кода в файле *MainActivity.kt*. Давайте посмотрим, как это делается.

Настройка значка выдвижной панели на панели инструментов...

Ранее в этой главе мы настроили панель инструментов так, чтобы на ней выводилась метка текущей цели и кнопка Up. Для этого мы построили объект `AppBarConfiguration` и связали его с панелью инструментов.

Теперь на панели инструментов нужно разместить значок выдвижной панели, для чего можно добавить выдвижную панель `AppBarConfiguration`. Код выглядит так:

Получение ссылки на DrawerLayout.

Эта строка означает, что на панели инструментов выводится значок для открытия выдвижной панели. Значок выдвижной панели будет отображаться на всех экранах без кнопки Up.

```
val drawer = findViewById<DrawerLayout>(R.id.drawer_layout)
val builder = AppBarConfiguration.Builder(navController.graph)
builder.setOpenableLayout(drawer)
val appBarConfiguration = builder.build()
toolbar.setupWithNavController(navController, appBarConfiguration)
```

Приведенный выше код добавляет макет выдвижной панели на объект `AppBarConfiguration`. Это позволяет панели инструментов взаимодействовать с выдвижной панелью и включать значок выдвижной панели на каждый экран без кнопки Up.

...и связывание выдвижной панели с контроллером навигации

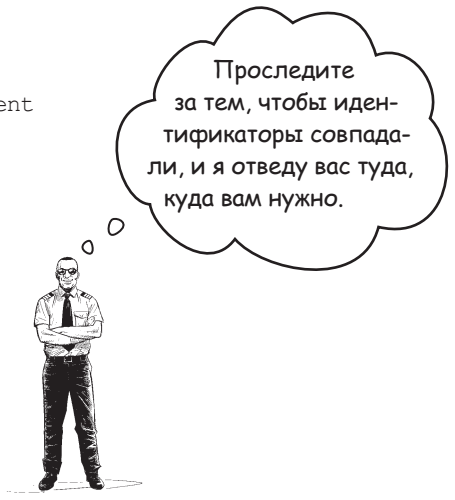
Наконец, необходимо обеспечить переход с выдвижной панели к правильной цели каждый раз, когда пользователь щелкает на одном из ее элементов. Как и в случае с нижней панелью навигации, для этого необходимо связать выдвижную панель с контроллером навигации.

Код для решения этой задачи выглядит так:

```
val navHostFragment = supportFragmentManager
    .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val navController = navHostFragment.navController
val navView = findViewById<NavigationView>(R.id.nav_view)
NavigationUI.setupWithNavController(navView, navController)
```

Каждый раз, когда пользователь щелкает на элементе выдвижной панели, контроллер навигации получает идентификатор из файла ресурсов меню и ищет совпадающий идентификатор в графе навигации. Затем он переходит к цели с указанным идентификатором.

И это весь код, необходимый для управления поведением выдвижной панели навигации. Добавим его в `MainActivity.kt` и проведем тест-драйв приложения.

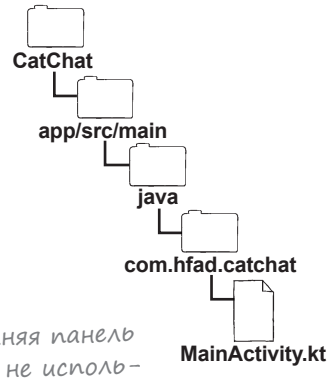


Контроллер навигации

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [ ] **Выдвижная панель**

Полный код MainActivity.kt

В коде `MainActivity` необходимо заменить код нижней панели навигации кодом выдвижной панели. Обновите *MainActivity.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.catchat

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.navigation.findNavController
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.onNavDestinationSelected
import androidx.navigation.ui.setupWithNavController
~~import com.google.android.material.bottomnavigation.BottomNavigationView~~
import androidx.drawerlayout.widget.DrawerLayout
import androidx.navigation.ui.NavigationUI
import com.google.android.material.navigation.NavigationView
import com.google.android.material.appbar.MaterialToolbar

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<MaterialToolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val navHostFragment = supportFragmentManager
            .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController
        val drawer = findViewById<DrawerLayout>(R.id.drawer_layout)
        val builder = AppBarConfiguration.Builder(navController.graph)
        builder.setOpenableLayout(drawer)
        val appBarConfiguration = builder.build()
        toolbar.setupWithNavController(navController, appBarConfiguration)
        ~~val bottomNavView = findViewById<BottomNavigationView>(R.id.bottom_nav)~~
        ~~bottomNavView.setupWithNavController(navController)~~
        val navView = findViewById<NavigationView>(R.id.nav_view)
        NavigationUI.setupWithNavController(navView, navController)
    }
```

Нижняя панель уже не используется, удаляем эту строку.

Эти дополнительные классы необходимо импортировать, так как они используются в коде выдвижной панели.

Получение ссылки на выдвижную панель.

Добавление выдвижной панели в AppBarConfiguration.

Эти строки удаляются, потому что нижняя панель навигации более не используется.

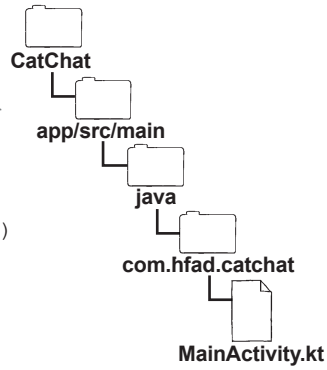
Навигация по щелчку на элементах включается связыванием выдвижной панели с контроллером навигации.

Продолжение на следующей странице.

Полный код MainActivity.kt (продолжение)

```
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_toolbar, menu)
        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        val navController = findNavController(R.id.nav_host_fragment)
        return item.onNavDestinationSelected(navController)
                || super.onOptionsItemSelected(item)
    }
}
```

Эти два метода необходимы только для панели инструментов. Для выдвижной панели они не нужны.



И это весь код, необходимый для выдвижной панели.
Давайте опробуем приложение на практике.

Часто Задаваемые Вопросы

В: Почему мы удалили нижнюю панель? Разве приложение не может иметь как нижнюю, так и выдвижную панель?

О: Приложение может иметь нижнюю и выдвижную панель. Мы решили заменить нижнюю панель выдвижной, чтобы более четко обозначить, какой код необходим для выдвижной панели.

В: Я слышал, что создание выдвижной панели требует большого объема кода активности. Это правда?

О: Когда-то включение выдвижной панели действительно требовало значительных усилий.

С появлением компонента Navigation реализация выдвижной панели заметно упростилась. Компонент берет на себя большую часть рутины, в результате чего код активности становится проще.

В: Как выдвижная панель узнает, к какому фрагменту нужно перейти?

О: Код активности вызывает метод `setupWithNavController()`, а затем связывает представление NavigationView выдвижной панели с контроллером навигации.

Когда пользователь щелкает на элементе выдвижной панели, идентификатор этого элемента передается контроллеру навигации. Контроллер навигации ищет идентификатор в графе навигации и переходит к цели с совпадающим идентификатором.

В: Напомните, что такое NavigationView?

О: Это разновидность FrameLayout, которая используется для отображения стандартного меню навигации. Обычно она используется в выдвижных панелях.

В: Я попытался щелкнуть на одном из элементов, но приложение не перешло к нужному фрагменту. Почему?

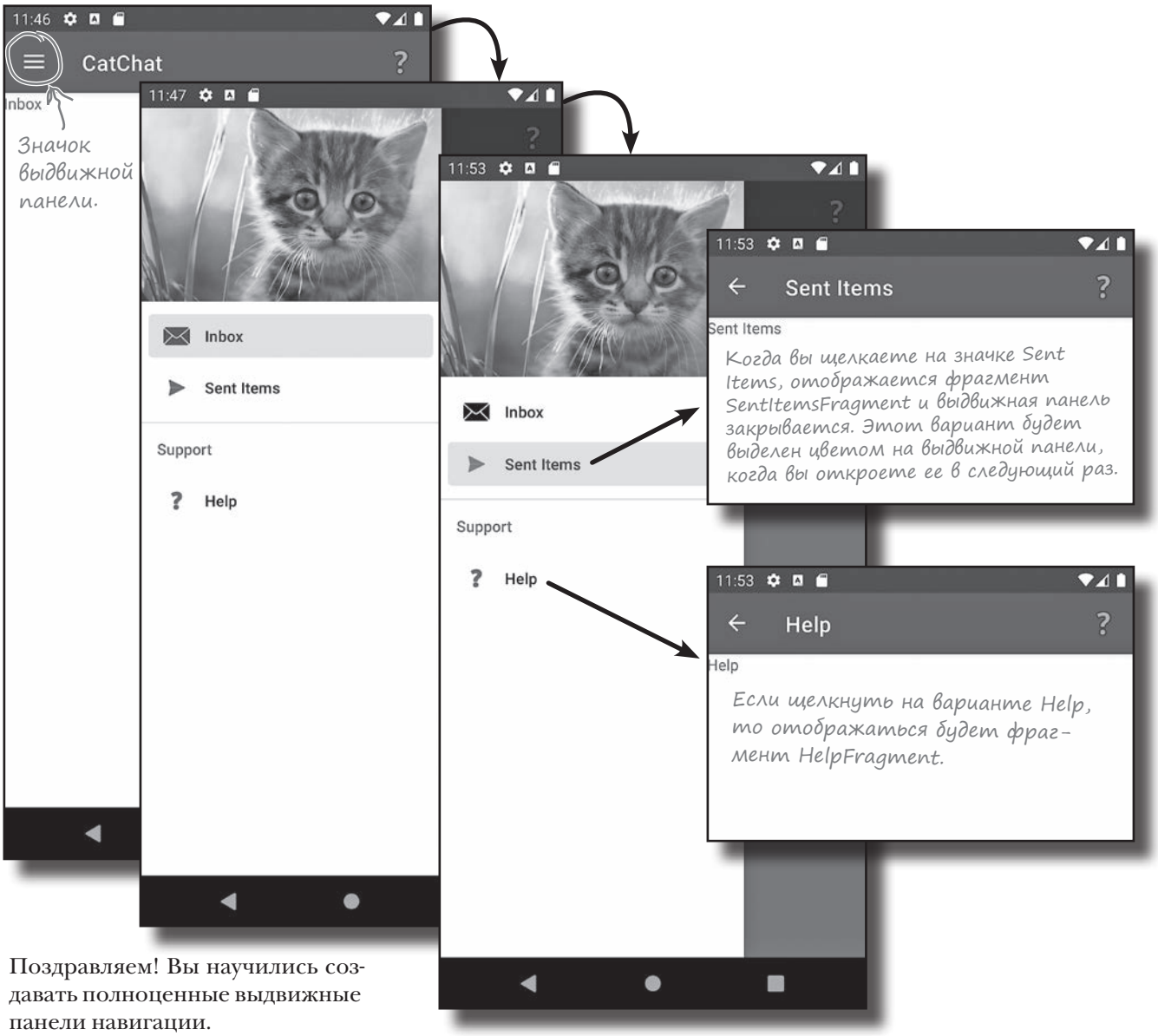
О: Прежде всего убедитесь в том, что в коде активности вызывается метод `setupWithNavController()`. Он связывает выдвижную панель с контроллером навигации, и без этого вызова выдвижная панель не сможет использовать компонент Navigation.

Затем убедитесь в том, что идентификаторы в файле ресурсов меню выдвижной панели и в графе навигации совпадают. Если контроллер навигации не найдет подходящий идентификатор, он не сможет использовать его для навигации.

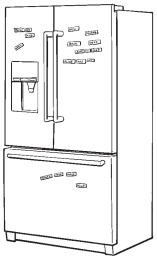
Тест-драйв

- [x] Панель инструментов
- [x] Панель инструментов навигации
- [x] Нижняя панель
- [x] **Выдвижная панель**

При запуске приложения на панели инструментов отображается значок выдвижной панели. Щелчок на этом значке открывает выдвижную панель. Когда вы щелкаете на одном из элементов выдвижной панели, фрагмент для этого варианта отображается в `MainActivity` и выдвижная панель закрывается.



Поздравляем! Вы научились создавать полноценные выдвижные панели навигации.



Развлечения с магнитами

Кто-то попытался выложить код макета для выдвижной панели на холодильнике, но от порыва ветра часть магнитов упала на пол. Удастся ли вам восстановить код?

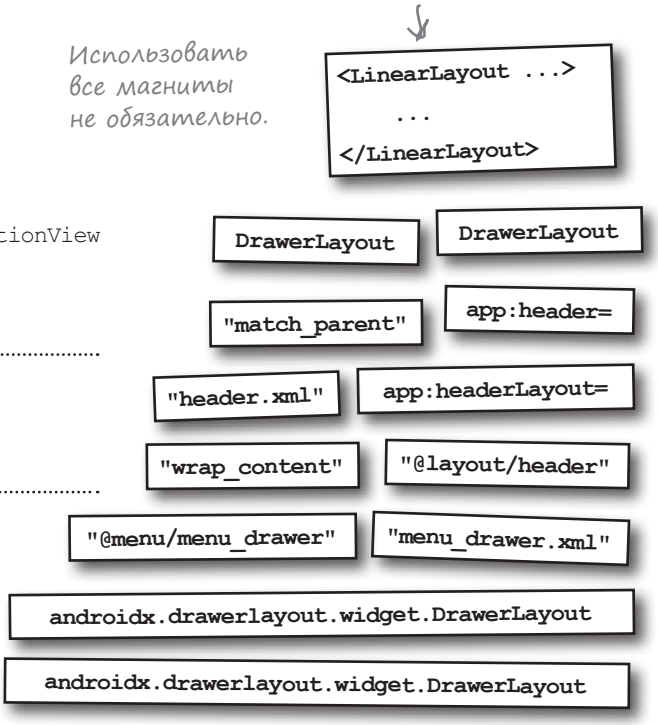
Активность использует линейный макет для основного контента (большая часть этого кода пропущена). Выдвижная панель должна отображать заголовок, определенный в файле макета с именем *header.xml*. Элементы меню задаются в файле ресурсов меню *menu_drawer.xml*.

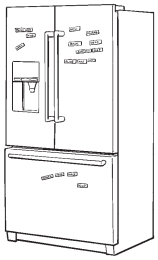
```
<?xml version="1.0" encoding="utf-8"?>

<.....
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"

        android:layout_width=  .....
        android:layout_height="match_parent"
        android:layout_gravity="start"

        .....

        app:menu=  ...../>

</.....>
```

Места, в которых должны располагаться отсутствующие магниты (кроме LinearLayout), обозначены пропусками. Вам придется решить, где должен располагаться магнит.

Большая часть кода LinearLayout пропущена, потому что для него не хватало места на холодильнике.

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Кто-то попытался выложить код макета для выдвижной панели на холодильнике, но от порыва ветра часть магнитов упала на пол. Удастся ли вам восстановить код?

Активность использует линейный макет для основного контента (большая часть этого кода пропущена). Выдвижная панель должна отображать заголовок, определенный в файле макета с именем *header.xml*. Элементы меню задаются в файле ресурсов меню *menu_drawer.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout ...>
        ...
    </LinearLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/header"
        app:menu="@menu/menu_drawer" />

</androidx.drawerlayout.widget.DrawerLayout>
```

Здесь необходимо задать полное имя класса DrawerLayout.

Основное содержимое приложения располагается под первым элементом DrawerLayout.

Назначает ширину выдвижной панели по ширине ее содержимого.

Задает заголовок выдвижной панели.

Использует menu_drawer.xml для меню выдвижной панели.

Закрывающий тег для DrawerLayout.

Эти магниты не пригодились.

DrawerLayout

"match_parent"

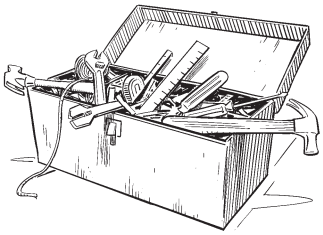
"header.xml"

app:header=

DrawerLayout

"menu_drawer.xml"

Ваш инструментарий Android



Глава 8 осталась позади, а ваш инструментарий пополнился пользовательскими интерфейсами навигации.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Для добавления панели приложения по умолчанию используется тема.
- Атрибут `android:theme` в *AndroidManifest.xml* указывает, какая тема должна применяться.
- Стили определяются элементами `<style>` в одном или нескольких ресурсных файлах.
- Панель приложения по умолчанию можно заменить панелью инструментов, поддерживающей новейшую функциональность Android.
- Панель инструментов Material хорошо работает в сочетании с темами Material.
- Файл ресурсов меню добавляет элементы к панелям инструментов, нижним и выдвижным панелям навигации.
- Проследите за тем, чтобы идентификаторы элемента и цели в файле ресурсов меню и графе навигации совпадали. От совпадения идентификаторов зависит работа компонента Navigation.
- Элементы добавляются на панель инструментов вызовом `onCreateOptionsMenu()`.
- Навигация по панели инструментов реализуется методом `onOptionsItemSelected()`.
- Объект `AppBarConfiguration` используется для настройки панели инструментов, обеспечивающей ее работу с компонентом Navigation.
- Нижняя панель может содержать до пяти элементов.
- Выдвижная панель может содержать много элементов, сгруппированных по категориям.
- Чтобы создать выдвижную панель навигации, добавьте выдвижную панель в макет активности. Первым элементом макета выдвижной панели должно быть представление, определяющее основное содержимое активности. Второй элемент определяет содержимое выдвижной панели.

Материальный мир

Многим приложениям нужен эффектный пользовательский интерфейс, быстро реагирующий на действия пользователя. В предыдущих главах вы научились пользоваться разными представлениями: **текстовыми представлениями, кнопками и полями выбора**, а также применять **темы Material** для внесения масштабных изменений в оформление и поведение приложения. Впрочем, это далеко не все. В этой главе вы узнаете, как улучшить скорость реакции вашего приложения применением **координирующего макета**. Вы создадите **панели инструментов** с возможностью сворачивания или прокрутки содержимого по вашему усмотрению. Вы познакомитесь с **интересными новыми представлениями: флажками**, **переключателями**, **плашками** и **плавающими кнопками действий**. Наконец, вы узнаете, как отображать удобные всплывающие сообщения с использованием панелей **Toast** и **Snackbar**. Переверните страницу, чтобы узнать больше.

Material широко используется в мире Androidville

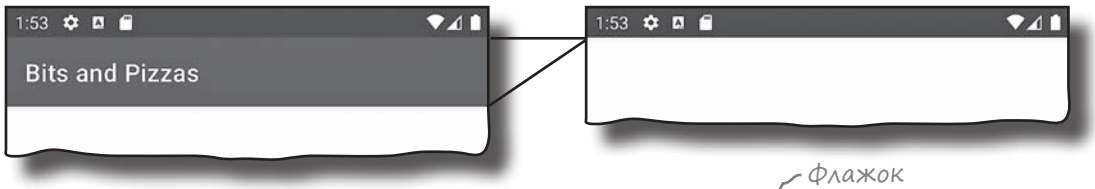
В предыдущей главе вы научились пользоваться панелями инструментов, нижними и выдвижными панелями навигации, упрощающими перемещение в приложении, и оформлять их применением тем из библиотеки Material. Как вы, вероятно, помните, Material представляет собой систему дизайна, которая помогает строить приложения, обладающие целостным оформлением и поведением по всем экранам.

Библиотека Material не ограничивается панелями инструментов, выдвижными и нижними панелями; стилевое оформление применяется ко всем представлениям в вашем приложении, от кнопок до текстовых представлений.

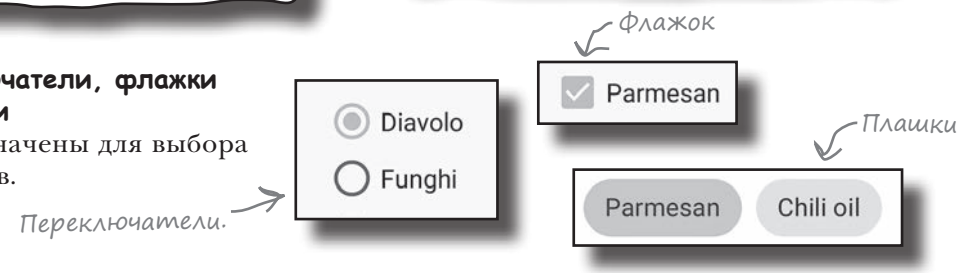
Несколько примеров компонентов и возможностей, использующих Material:

Панели инструментов с прокруткой и сверткой
Панель инструментов может уходить за пределы экрана (сворачиваться) при прокрутке содержимого пользователем.

Панель инструментов может исчезать с экрана при прокрутке, чтобы на экране оставалось больше места.

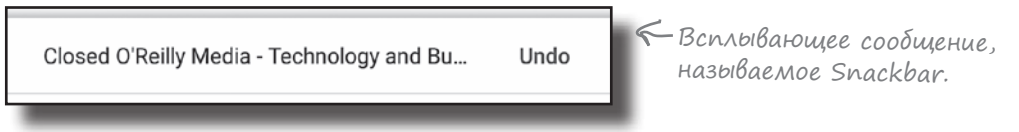


Переключатели, флажки и плашки
Предназначены для выбора вариантов.



Плавающие кнопки действий (FAB)
FAB — специальные кнопки, которые «плавают» над основным экраном.

Snackbar
Всплывающие сообщения, с которыми можно взаимодействовать.

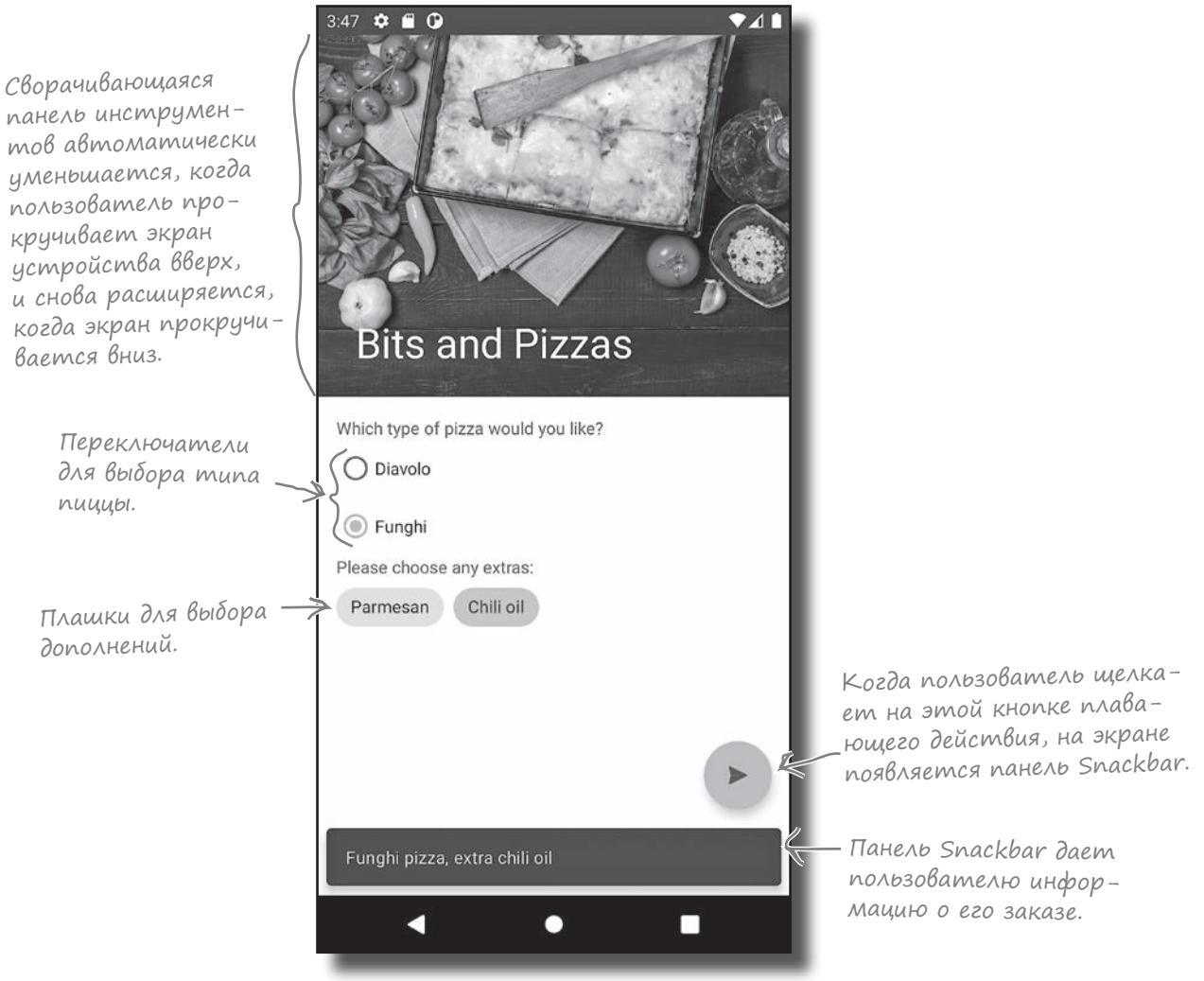


Чтобы показать, как использовать эти представления и возможности, мы построим новое приложение.

Приложение Bits and Pizzas

Мы построим новое приложение, которое называется Bits and Pizzas. Основное внимание будет уделено экрану Create Order, который позволяет пользователю оформить заказ на пиццу.

Экран выглядит так:



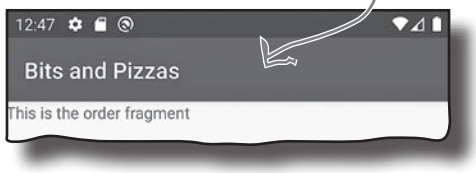
Приложение состоит из активности `MainActivity`, отображающей фрагмент с именем `OrderFragment`. Фрагмент определяет внешний вид и функциональность экрана Create Order.

Рассмотрим последовательность действий для построения приложения.

Что предстоит сделать

Чтобы построить приложение, мы выполним следующие действия:

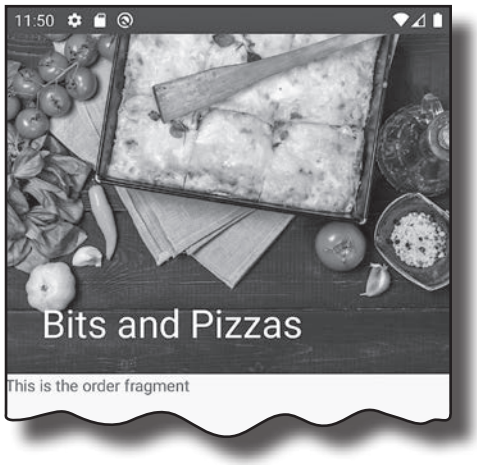
Когда пользователь прокручивает экран, панель инструментов тоже участвует в прокрутке.



1. **Добавление панели инструментов с прокруткой.** Мы создадим фрагмент `OrderFragment` и добавим в его макет панель инструментов, которая уходит с экрана, когда пользователь прокручивает экран вверх, и появляется снова при прокрутке вниз.

2. **Реализация сворачиваемой панели инструментов.** Когда панель инструментов будет поддерживать прокрутку, мы добавим к ней изображение и заставим ее сворачиваться и разворачиваться, когда пользователь прокручивает содержимое экрана.

Панель инструментов с изображением. Когда пользователь прокручивает экран, панель будет сворачиваться.



3. **Добавление представлений.** Пользователь должен иметь возможность оформить заказ на пиццу. Для этого в макет `OrderFragment` будут добавлены переключатели, плашки и плавающая кнопка действия.

Фрагмент будет включать эти представления и FAB.

4. **Программирование реакции FAB на щелчки.** Когда пользователь щелкает на FAB, приложение будет отображать всплывающее сообщение с подробным описанием заказа.

Это кнопка FAB, мы добавим ее на шаге 3.

Эта панель Snackbar будет отображаться при щелчке на FAB.

Создание проекта Bits and Pizzas

Мы создадим новый проект для приложения Bits and Pizzas по схеме, уже знакомой вам по предыдущим главам. Выберите вариант Empty Activity, введите имя проекта «Bits and Pizzas» и имя пакета «com.hfad.bitsandpizzas», подтвердите папку сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin с минимальным уровнем SDK API 21, чтобы приложение работало на большинстве устройств Android.

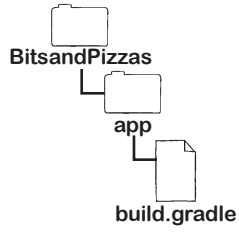
Включение зависимости для библиотеки Material в файл build.gradle приложения.

В этой главе будут использоваться темы, представления и функциональность библиотеки Material. Следовательно, мы должны убедиться в том, что она включена в файл *build.gradle* приложения как зависимость.

Откройте файл *BitsandPizzas/app/build.gradle* и убедитесь в том, что раздел `dependencies` включает следующую строку (выделенную жирным шрифтом):

Используемая версия библиотеки Material.

```
dependencies {
    ...
    implementation 'com.google.android.material:material:1.4.0'
    ...
}
```



Весьма вероятно, что среда Android Studio уже включила в файл эту зависимость за вас. Если она отсутствует, добавьте ее самостоятельно и щелкните на ссылке Sync Now, появляющейся в верхней части редактора кода, чтобы синхронизировать изменения с остальными частями проекта.

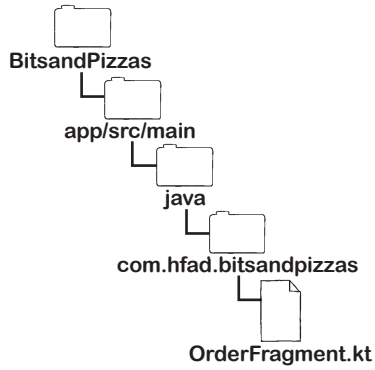
После включения библиотеки Material в приложение можно переходить к созданию `OrderFragment`.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Создание OrderFragment

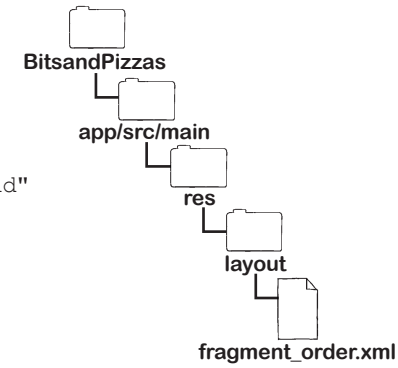
`OrderFragment` — главный экран приложения Bits and Pizzas, где пользователь будет оформлять заказ на пиццу.

Чтобы создать фрагмент, выделите пакет *com.hfad.bitsandpizzas* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «OrderFragment» и имя макета «fragment_order» и убедитесь в том, что выбран язык Kotlin. Затем обновите код *OrderFragment.kt* и приведите его к следующему виду:



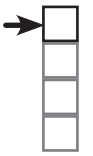

```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_order, container, false)
    }
}
```

Обновите код в файле макета *fragment_order.xml*, чтобы он содержал фреймовый макет:




```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">
</FrameLayout>
```

В оставшейся части главы мы будем заниматься обновлением `OrderFragment`. Начнем с отображения его в макете `MainActivity`.

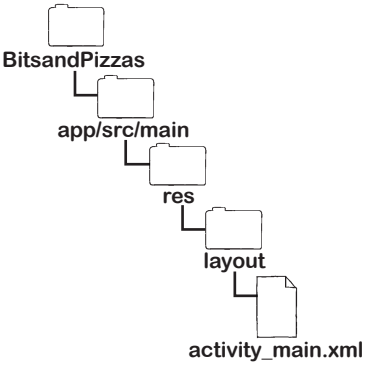


Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Отображение OrderFragment в макете MainActivity

Чтобы включить `OrderFragment` в макет `MainActivity`, воспользуемся `FragmentContainerView` с указанием имени фрагмента.

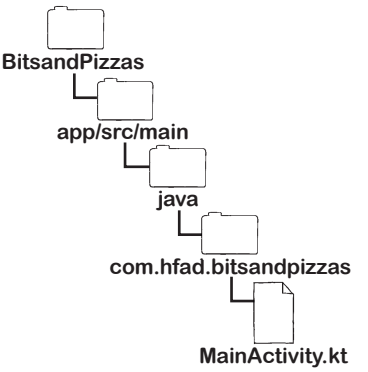
Ниже приведен код решения этой задачи; откройте файл *activity_main.xml* и обновите его код (изменения выделены жирным шрифтом):




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.hfad.bitsandpizzas.OrderFragment"
    tools:context=".MainActivity" />
```

В этой главе навигация не используется, поэтому макету MainActivity достаточно просто отобразить OrderFragment.

Наконец, откройте *MainActivity.kt* и убедитесь в том, что код в файле соответствует приведенному ниже:



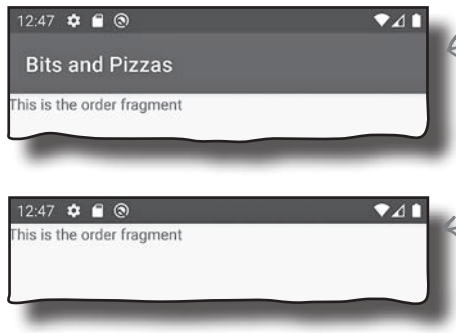

```
package com.hfad.bitsandpizzas

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Теперь в `MainActivity` отображается `OrderFragment`. Давайте посмотрим, как заставить панель приложения реагировать на прокрутку.

Замена стандартной панели приложения панелью инструментов

Сейчас мы займемся реакцией панели приложения Bits and Pizzas на прокрутку. Первая задача — позаботиться о том, чтобы панель приложения уходила с экрана, когда пользователь прокручивает экран вверх, и появлялась снова, когда экран прокручивается вниз.

Для этого необходимо сначала заменить стандартную панель приложения панелью инструментов. Так как стандартная панель приложения закрепляется у верхнего края экрана, она не может прокручиваться. С другой стороны, панель инструментов обладает куда большей гибкостью.



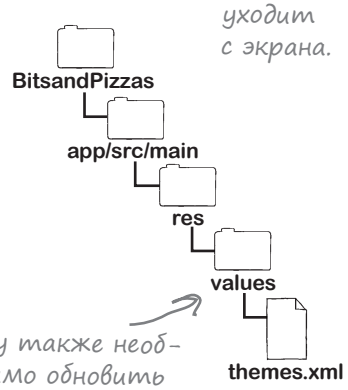
Это панель инструментов.

Когда вы прокручиваете экран, панель инструментов уходит с экрана.

Для этого сначала необходимо заменить тему приложения другой, не содержащей панели приложения. Откройте файл *themes.xml* в *app/src/main/res/values*, обновите его и включите стиль, выделенный ниже жирным шрифтом:

```
<resources>
    <style name="Theme.BitsAndPizzas"
           parent="Theme.MaterialComponents.DayNight.NoActionBar">
        ...
    </style>
</resources>
```

Вместо стандартной панели приложения будет использоваться панель инструментов, поэтому необходимо применить стиль «NoActionBar».



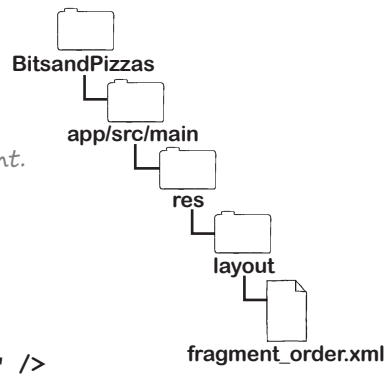
Тему также необходимо обновить в версии этого файла из папки values-night (если она существует).

Если ваш проект содержит файл *themes.xml* в папке *values-night*, описанное выше изменение также придется применить в этом файле.

После того как изменение будет внесено, добавьте панель инструментов в `FragmentOrder`. Для этого обновите код файла *fragment_order.xml*, чтобы он совпадал с приведенным ниже:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        style="@style/Widget.MaterialComponents.Toolbar.Primary" />

</FrameLayout>
```

Добавляет панель инструментов Material в макет OrderFragment.



У фрагментов нет метода setSupportActionBar()

После того как панель инструментов будет добавлена в проект, необходимо наделить ее поведением обычной панели приложения, в которой выводится имя приложения. Как вы узнали в предыдущей главе, для этого следует вызвать метод `setSupportActionBar()` активности.

Но здесь панель инструментов добавляется в фрагмент, а **у фрагментов нет метода `setSupportActionBar()`**. Чтобы обойти это затруднение, мы получим ссылку на активность, отображающую фрагмент, преобразуем ее к типу `AppCompatActivity` и вызовем метод `setSupportActionBar()`.

Задача решается следующим кодом:

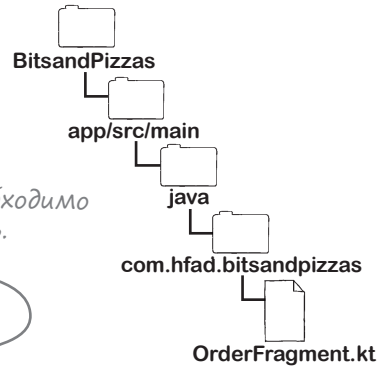
```
val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
(activity as AppCompatActivity).setSupportActionBar(toolbar)
```

Преобразует активность, отображающую фрагмент, в AppCompatActivity и вызывает ее метод setSupportActionBar.

Ниже приведен полный код *OrderFragment.kt*; включите изменения в свою версию (выделены жирным шрифтом):

```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.appbar.MaterialToolbar

class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return val view = inflater.inflate(R.layout.fragment_order, container, false)
        val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
        (activity as AppCompatActivity).setSupportActionBar(toolbar)
        return view
    }
}
```

Эти классы необходимо импортировать.

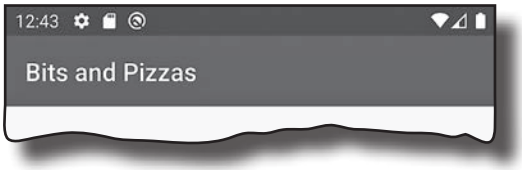


Это позволяет активности использовать панель инструментов как панель приложения, чтобы на ней выводилось имя приложения.

- **Панель с прокруткой**
- Сворачиваемая панель
- Представления
- Реакция на щелчки

Добавили панель инструментов... что дальше?

Мы включили панель инструментов в макет `OrderFragment`, чтобы при запуске приложения панель инструментов отображалась у верхнего края экрана:



Но если вы попытаетесь прокрутить экран, панель инструментов останется неподвижной. Чтобы она реагировала на прокрутку, необходимо внести ряд изменений.

Панель инструментов должна реагировать на прокрутку

Чтобы панель инструментов перемещалась, необходимо добавить дополнительные представления в макет фрагмента. Макет должен иметь следующую структуру:

В корне должен находиться координирующий макет.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>

    <com.google.android.material.appbar.AppBarLayout ...>

        <com.google.android.material.appbar.MaterialToolbar .../>

    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView ...>

        ...

    </androidx.core.widget.NestedScrollView>

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Панель инструментов должна размещаться внутри макета панели приложения.

Основное содержимое фрагмента должно размещаться внутри вложенного представления с прокруткой.

Макет должен включать три составляющие: **координирующий макет**, **макет панели приложения** и **вложенное представление прокрутки**. Их комбинация позволяет панели инструментов реагировать на прокрутку экрана пользователем.

Давайте разберемся, что делает каждый из них, начиная с координирующего макета.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Координирующий макет согласовывает анимации между представлениями

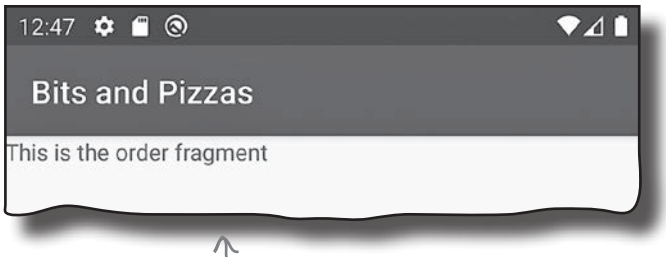
Координирующий макет напоминает расширенный фреймовый макет, который используется для координации анимаций между разными представлениями. Например, он может координировать прокрутку основного содержимого макета с выходом панели инструментов за границы экрана.

Координирующий макет добавляется в код макета следующим образом:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ...>

    ...

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Здесь размещаются все представления, поведение которых должно координироваться.

Включите все представления, анимации которых должны координироваться в координирующем макете. Например, в приложении Bits and Pizzas необходимо координировать две операции: прокрутку основного содержимого макета пользователем и выход панели инструментов за границы экрана. Это означает, что панель инструментов и основное содержимое экрана необходимо включить в координирующий макет:



Панель инструментов должна выходить за пределы экрана при прокрутке основного содержимого макета. Так как эти две анимации должны координироваться, все, что отображается в макете, необходимо включить в координирующий макет.

CoordinatorLayout позволяет поведению одного представления влиять на поведение другого представления.

CoordinatorLayout обычно является корневым элементом вашего макета.

Итак, теперь вы знаете, как работает координирующий макет. Перейдем к макету панели приложения.

Макет панели приложения включает анимацию панели инструментов

Макет панели приложения представляет собой разновидность вертикального линейного макета, разработанную для работы с панелями приложений. Его взаимодействие с координирующим макетом делает возможной анимацию панели инструментов.

Макет панели приложения добавляется в код приложения следующим образом:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ...>

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar .../>

    </com.google.android.material.appbar.AppBarLayout>

    ...

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

AppBarLayout добавляется в CoordinatorLayout.

Панель инструментов размещается внутри AppBarLayout.

Применяет стиль ко всем представлениям макета панели приложения.

Обратите внимание на атрибут `android:theme` в приведенном выше коде макета панели приложения:

```
android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar"
```

Значение применяет заданный стиль к макету панели приложения и всем его представлениям. В данном случае это означает, что панель инструментов — и все остальное, что будет добавлено в макет панели приложения, — будет оформлена с использованием свойств панели приложения темы Material, включая цвета ее фона и текста.

Превосходно! Теперь ваша панель инструментов может реагировать на события прокрутки. Впрочем, это еще не все: также необходимо указать, *как* она должна на них реагировать. Давайте посмотрим, как это делается.

Программирование реакции панели инструментов на события прокрутки

После того как панель приложения будет добавлена, нужно сообщить ей, как реагировать на прокрутку. Для этого следует добавить к панели инструментов атрибут **`app:layout_scrollFlags`** и присвоить ему значение. Значение указывает, как панель инструментов должна реагировать на события прокрутки.

В приложении Bits and Pizzas панель инструментов должна прокручиваться вверх за пределы экрана, когда пользователь перемещает экран вверх, и быстро возвращаться в исходную позицию, когда пользователь перемещает экран вниз. Для этого следует присвоить атрибуту `app:layout_scrollFlags` панели инструментов значение `"scroll|enterAlways"`, для чего используется код следующего вида:

Чтобы панель инструментов прокручивалась, она ДОЛЖНА находиться внутри макета панели приложения. Прокрутка обеспечивается совместной работой макета панели приложения и координирующего макета.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>
    <com.google.android.material.appbar.AppBarLayout...>

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways"/>

    </com.google.android.material.appbar.AppBarLayout>
    ...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Эта строка сообщает CoordinatorLayout (и AppBarLayout), как панель инструментов должна реагировать на прокрутку содержимого пользователем.

Строка

```
app:layout_scrollFlags="scroll|enterAlways"
```

задает два аспекта поведения: `scroll` и `enterAlways`.

Значение `scroll` означает, что представление может прокручиваться за верхний край экрана при прокрутке экрана вверх. Без этого значения панель инструментов осталась бы закрепленной у верхнего края экрана и не участвовала бы в прокрутке.

Значение `enterAlways` означает, что панель инструментов быстро прокручивается вниз до своей исходной позиции, когда пользователь прокручивает экран вниз. Без этого значения панель инструментов все равно прокручивалась бы вниз, но это происходило бы намного медленнее.

Почти все! Чтобы панель инструментов `OrderFragment` прокручивалась, осталось сделать последний шаг: добавить **вложенное представление прокрутки**.

Представление вложенной прокрутки делает возможной прокрутку содержимого макета

Теперь необходимо добавить **представление вложенной прокрутки** `NestedScrollView`, чтобы основное содержимое макета могло прокручиваться. Такие представления работают так же, как и обычные представления прокрутки, не считая того, что они поддерживают *вложенную* прокрутку. Это важно, потому что координирующий макет *прослушивает события только вложенной прокрутки*. Если вы используете обычное представление прокрутки в своем макете, панель инструментов не сможет реагировать на прокрутку экрана пользователем.

Еще одно представление, которое поддерживает вложенную прокрутку, — RecyclerView. Вы узнаете об этом представлении в главе 15.

Для включения представления вложенной прокрутки в макет используется код следующего вида:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
    <com.google.android.material.appbar.AppBarLayout...>
        ...
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="This is the order fragment" />

    </androidx.core.widget.NestedScrollView>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Определяет NestedScrollView.

Эта строка гарантирует, что представление NestedScrollView не будет скрываться AppBarLayout.

Все представления, для которых должна поддерживаться прокрутка, добавляются в NestedScrollView.

В приведенном выше примере представление вложенной прокрутки включает дополнительный атрибут с именем `app:layout_behavior`, которому присваивается встроенное строковое значение `"@string/appbar_scrolling_view_behavior"`. Оно гарантирует, что содержимое представления вложенной прокрутки будет размещено под макетом панели приложения и будет перемещаться при его прокрутке.

Обратите внимание: представление вложенной прокрутки может иметь только одного прямого потомка (в приведенном примере это текстовое представление). Если вы захотите добавить в представление вложенной прокрутки более одного представления, вам придется сначала включить их в группу представлений (например, линейный макет), а потом добавить группу в представление вложенной прокрутки.

Пример будет приведен позднее в этой главе.

Вот и все, что необходимо знать для обеспечения прокрутки панели инструментов Bits and Pizzas. Сделаем следующий шаг и обновим макет `OrderFragment`.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

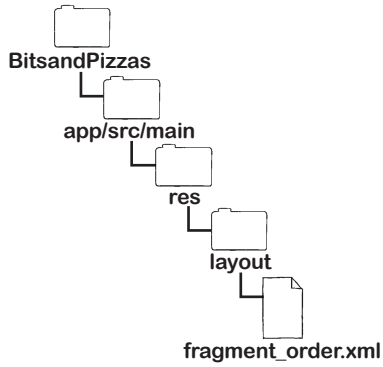
Полный код fragment_order.xml

Ниже приведен полный код *fragment_order.xml*; обновите свою версию файла (изменения выделены жирным шрифтом):

Необходимо использовать CoordinatorLayout вместо FrameLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<~~FrameLayout~~ androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">
```

Добавляет пространство имен.




```
    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            ~~style="@style/Widget.MaterialComponents.Toolbar.Primary"~~
            app:layout_scrollFlags="scroll|enterAlways" />
    </com.google.android.material.appbar.AppBarLayout>
```

Чтобы панель инструментов реагировала на прокрутку, она должна быть заключена в AppBarLayout.

Удалите эту строку, так как для оформления панели инструментов будет использоваться тема AppBarLayout.

Эта строка указывает, как панель инструментов должна реагировать на прокрутку.

```
    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="This is the order fragment" />
    </androidx.core.widget.NestedScrollView>
</~~FrameLayout~~ androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Добавляем NestedScrollView, чтобы панель инструментов могла реагировать на прокрутку содержимого.

Добавляем TextView, чтобы иметь содержимое для прокрутки во время пробного запуска приложения.

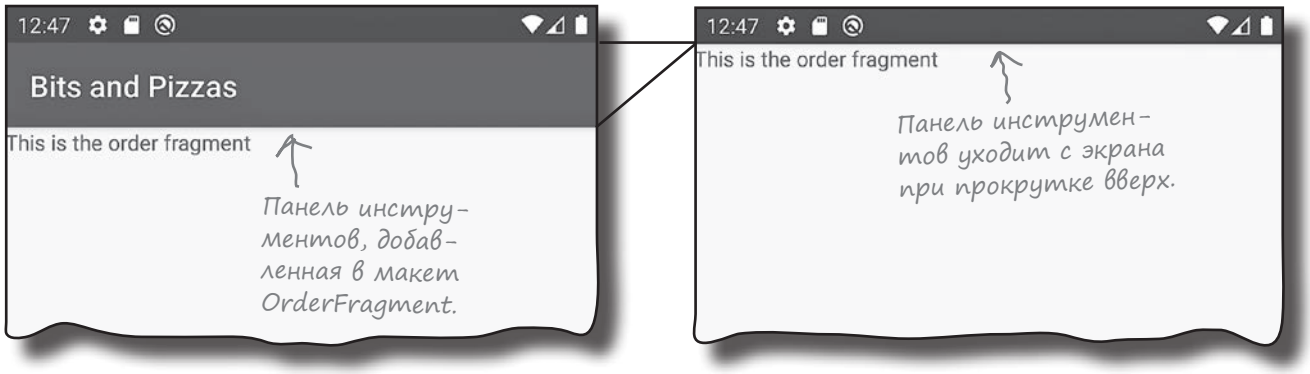
Заменяем FrameLayout на CoordinatorLayout.

Тест-драйв

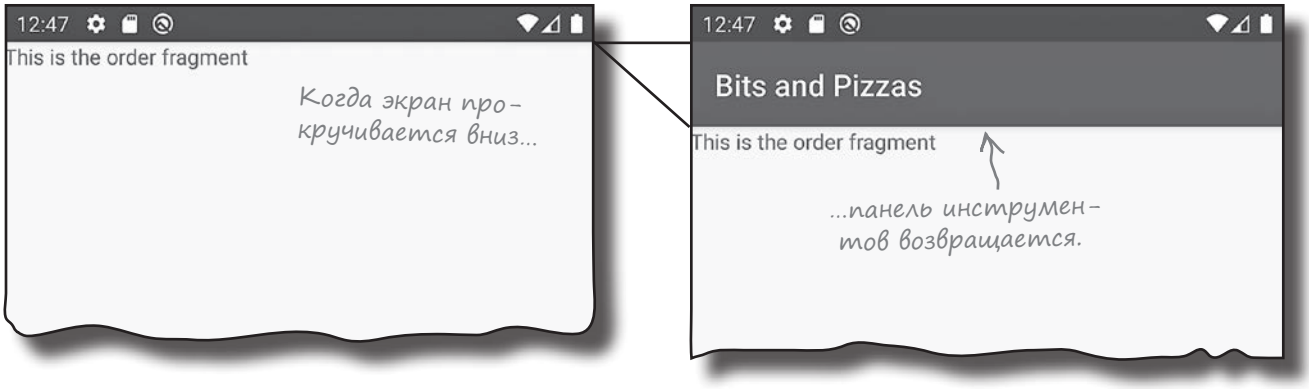
- [x] **Панель с прокруткой**
- [ ] **Сворачиваемая панель**
- [ ] **Представления**
- [ ] **Реакция на щелчки**

При запуске приложения в макете `MainActivity` отображается фрагмент `OrderFragment`. Панель инструментов отображается у верхнего края экрана.

Когда вы прокручиваете экран вверх, панель инструментов уходит с экрана за верхний край.



Когда экран прокручивается вниз, панель инструментов возвращается.



Поздравляем! Вы узнали, как создать панель инструментов, которая реагирует на прокрутку.

После следующего упражнения вы узнаете, как преобразовать панель инструментов с прокруткой в панель, которая сворачивается и разворачивается при прокрутке экрана пользователем.

СТАНЬ макетом

Ниже приведен файл макета для фрагмента с именем MyFragment. Представьте себя на месте макета и измените код, чтобы при прокрутке экрана вверх панель инструментов уходила с экрана, а при прокрутке вниз быстро возвращалась на экран.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MyFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize" />
    </com.google.android.material.appbar.AppBarLayout>

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        ... ← Здесь идут другие представления, но мы их опускаем.

    </ScrollView>
</FrameLayout>
```

СТАНЬ макетом. Решение

Ниже приведен файл макета для фрагмента с именем MyFragment. Представьте себя на месте макета и измените код, чтобы при прокрутке экрана вверх панель инструментов уходила с экрана, а при прокрутке вниз быстро возвращалась на экран.

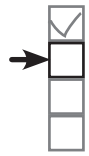
Необходимо использовать CoordinatorLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MyFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways" />
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        ...

    </androidx.core.widget.NestedScrollView>
</FrameLayout androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Обеспечивает прокрутку панели инструментов.

Необходимо использовать NestedScrollView, а не ScrollView.

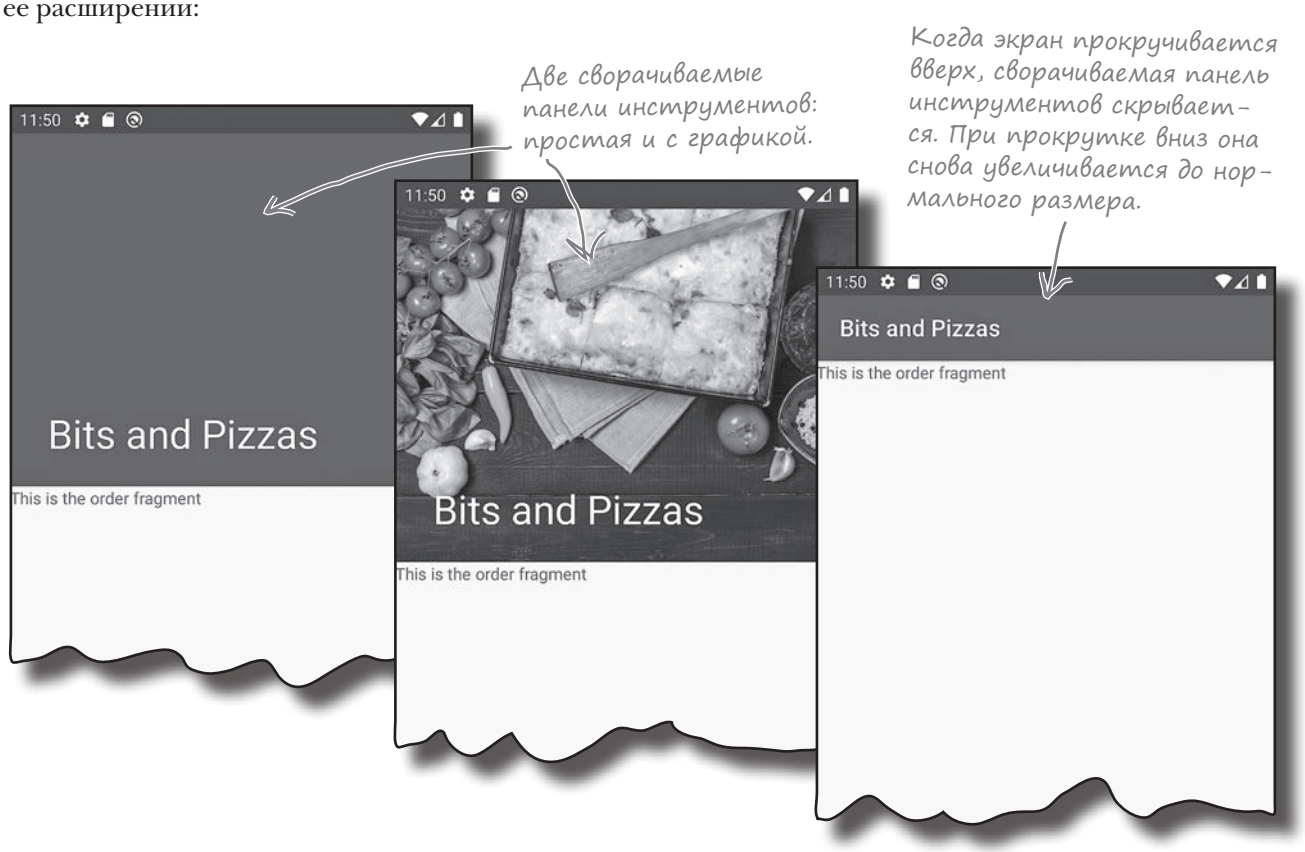
Размещает NestedScrollView так, чтобы представление не скрывалось AppBarLayout.



Создание сворачиваемой панели инструментов

Вы знаете, как заставить панель инструментов уходить с экрана. Сделаем следующий шаг и заменим ее несколько иной разновидностью панелей инструментов: **сворачиваемой панелью инструментов**.

Сворачиваемая панель инструментов изначально имеет обычные размеры, уменьшается при прокрутке экрана вверх и снова принимает нормальные размеры при прокрутке вниз. К ней даже можно добавить изображение, которое исчезает при достижении панелью минимальной высоты и снова становится видимым при ее расширении:



На нескольких ближайших страницах вы узнаете, как превратить простую панель инструментов в сворачиваемую, для этого нужно добавить ее в макет `OrderFragment`. Начнем с создания простой сворачиваемой панели инструментов, а затем создадим панель с графическим изображением.

Итак, за дело.

Создание простой сворачиваемой панели инструментов

Панель инструментов с прокруткой относительно просто преобразуется в сворачиваемую панель инструментов. Для этого достаточно заключить панель инструментов в **макет сворачиваемой панели инструментов** и настроить атрибуты панели инструментов. Базовая структура кода выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>
    <com.google.android.material.appbar.AppBarLayout ...>

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed">

            <com.google.android.material.appbar.MaterialToolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin"/>

        </com.google.android.material.appbar.CollapsingToolbarLayout>

    </com.google.android.material.appbar.AppBarLayout>

    ...

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Добавляет макет сворачиваемой панели инструментов, который находится внутри макета панели приложения.

Максимальная высота сворачиваемой панели инструментов.

Панель инструментов заключается в макет сворачиваемой панели инструментов.

Эта строка не позволяет элементам (таким, как кнопка Up) уходить за пределы экрана при сворачивании панели инструментов.

Закрывающий элемент сворачиваемой панели инструментов.

Как видите, макет сворачиваемой панели инструментов определяется элементом <...CollapsingToolbarLayout> из библиотеки *com.google.android.material*. Максимальная высота определяется его атрибутом `layout_height`, а строка:

```
app:layout_scrollFlags="scroll|exitUntilCollapsed"
```

Это означает, что панель инструментов должна сворачиваться, пока не свернется окончательно.

отдает команду панели сворачиваться при прокрутке вверх, пока остается возможность для уменьшения, и расширяться при прокрутке вниз, пока панель не достигнет своей полной высоты.

Также необходимо позаботиться о том, чтобы при сворачивании панели инструментов все, что на ней отображается, оставалось на экране (например, кнопка Up и любые элементы меню). Для этого в элемент Toolbar добавляется следующий атрибут:

```
app:layout_collapseMode="pin"
```

Без этой строки кнопка Up и все элементы меню будут уходить с экрана при прокрутке.

Как добавить изображение к сворачиваемой панели инструментов

После того как вы создадите простую сворачиваемую панель инструментов, к ней можно добавить изображение — для этого добавьте в макет сворачиваемой панели инструментов элемент <ImageView> и укажите изображение, которое вы хотите использовать. Код имеет следующую структуру:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout ...>
    <com.google.android.material.appbar.AppBarLayout ...>

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary">

            <ImageView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                android:src="@drawable/image"
                app:layout_collapseMode="parallax"/>

            < com.google.android.material.appbar.MaterialToolbar .../>

        </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>
...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Эта строка необязательна. Она назначает панели инструментов в свернутом состоянии простой цвет фона.

Добавляет изображение к макету сворачиваемой панели инструментов.

Добавляет параллаксную анимацию, чтобы при сворачивании панели инструментов изображение прокручивалось со скоростью, отличающейся от скорости остальных частей панели инструментов.

Строка

```
app:contentScrim="?attr/colorPrimary"
```

добавленная в <...CollapsingToolbarLayout>, назначает панели инструментов простой цвет фона в свернутом состоянии. Также к изображению была добавлена параллаксная анимация, для чего использовалась следующая строка:

```
app:layout_collapseMode="parallax"
```

Этот атрибут не является обязательным; просто с ним изображение прокручивается со скоростью, отличающейся от скорости прокрутки остальных частей панели инструментов.

Итак, теперь вы знаете, как создаются сворачиваемые панели инструментов. Добавим такую панель в макет OrderFragment.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Добавление изображения

Мы хотим, чтобы сворачиваемая панель инструментов `OrderFragment` включала изображение ресторана, поэтому давайте начнем с добавления ее в проект.

Убедитесь в том, что в проекте присутствует папка с именем *app/src/main/res/drawable*, затем загрузите файл *restaurant.webp* по адресу *tinyurl.com/hfad3* и добавьте его в папку *drawable*. Тем самым вы включаете изображение в свой проект как графический ресурс.

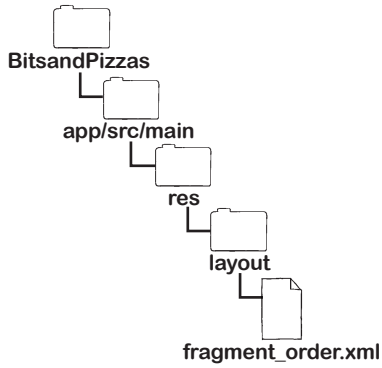
А теперь добавим сворачиваемую панель инструментов.

Это изображение необходимо включить в папку drawable.

Полный код fragment_order.xml

Приведенный ниже код добавляет сворачиваемую панель инструментов в макет `OrderFragment`. Обновите код *fragment_order.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary">
```

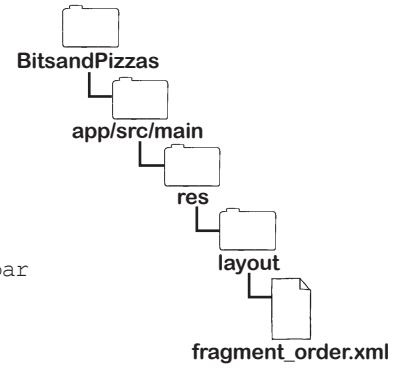


Добавляет CollapsingToolbarLayout.

Продолжение на следующей странице.

fragment_order.xml (продолжение)

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки



Добавляет изображение на панель инструментов.

```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="centerCrop"
    android:src="@drawable/restaurant"
    app:layout_collapseMode="parallax"/>

<com.google.android.material.appbar.MaterialToolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_scrollFlags="scroll|enterAlways"
    app:layout_collapseMode="pin" />
</com.google.android.material.appbar.CollapsingToolbarLayout>
</com.google.android.material.appbar.AppBarLayout>

<androidx.core.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="This is the order fragment" />
</androidx.core.widget.NestedScrollView>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Удалите эту строку, чтобы базовая панель инструментов не уходила с экрана.

Если панель инструментов включала кнопку Up или элементы меню, эта строка не позволяет им уходить с экрана.

Закрывающий элемент CollapsingToolbarLayout.

И это все изменения, которые были необходимы для создания сворачиваемой панели инструментов. Проведем тест-драйв приложения и посмотрим, как оно выглядит.

Тест-драйв

- Панель с прокруткой
- **Сворачиваемая панель**
- Представления
- Реакция на щелчки

При запуске приложения отображается фрагмент `OrderFragment`. Он включает сворачиваемую панель инструментов с изображением.

При прокрутке вверх панель инструментов сворачивается, изображение растворяется, а фон панели инструментов заменяется основным цветом приложения. При прокрутке вниз панель расширяется и изображение появляется снова.

В свернутом состоянии панель инструментов меняет цвет.

Изображение выводится на панели инструментов.

Часто Задаваемые Вопросы

В: Макет сворачиваемой панели инструментов является разновидностью группы представлений?

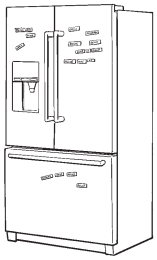
О: Да. На самом деле это субкласс `FrameLayout`, поэтому с ним могут использоваться любые атрибуты `FrameLayout`.

В: Нужно ли написать код Kotlin, чтобы панель инструментов сворачивалась?

О: Нет, все необходимое делается кодом макета.

В: Вы сказали, что координирующий макет координирует анимации между своими представлениями. Существуют ли другие альтернативы?

О: Да, существует более новая разновидность макета MotionLayout. Этот макет входит в библиотеку макетов с ограничениями, начиная с версии 2.0. За дополнительной информацией о MotionLayout обращайтесь по адресу *https://developer.android.com/training/constraint-layout/motionlayout*



Развлечения с магнитами

Кто-то попытался выложить на холодильнике структуру файла макета, реализующего сворачиваемую панель инструментов с изображением. К сожалению, магниты отвалились, когда большой птеродактиль пролетел рядом в поисках еды.

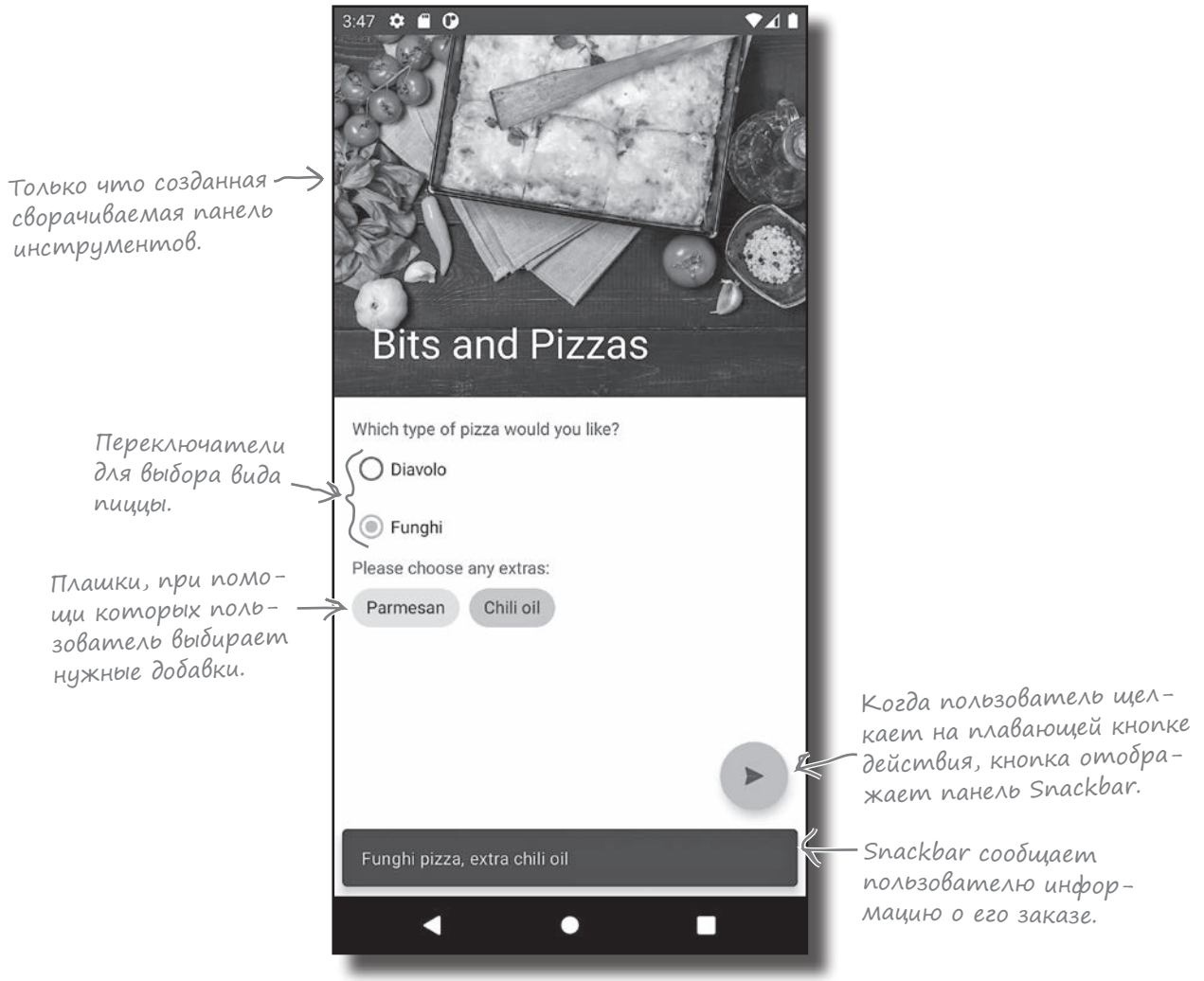
Удастся ли вам вернуть магниты на место в правильном порядке?

```
</com.google.android.material.appbar.MaterialToolbar>
<com.google.android.material.appbar.MaterialToolbar...>
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
</androidx.core.widget.NestedScrollView>
<androidx.core.widget.NestedScrollView...>
</com.google.android.material.appbar.CollapsingToolbarLayout>
</ImageView>
<com.google.android.material.appbar.CollapsingToolbarLayout...>
<ImageView...>
</com.google.android.material.appbar.AppBarLayout>
<com.google.android.material.appbar.AppBarLayout...>
```

Ответы на с. 439.

Построение основного контента OrderFragment

После добавления сворачиваемой панели инструментов в макет `OrderFragment` необходимо добавить дополнительные представления. С их помощью пользователь сможет выбрать вид пиццы и добавки (например, пармезан или масло чили); при щелчке на кнопке будет выводиться сообщение. Экран должен выглядеть примерно так:

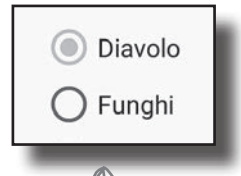


Как видите, макет `OrderFragment` включает некоторые дополнительные представления, которыми вы еще не умеете пользоваться. Прежде чем строить макет, узнаем побольше об этих представлениях.

Выбор вида пиццы при помощи переключателя

Первое представление, которое мы используем, — группа **переключателей**, в которой пользователь выбирает вид пиццы. Переключатели позволяют вывести набор вариантов, из которого выбирается один вариант, поэтому переключатели хорошо подходят для этой ситуации.

Для добавления переключателей в макет используются два элемента: `<RadioButton>` и `<RadioGroup>`. Элемент `<RadioButton>` используется для определения каждого переключателя, а элемент `<RadioGroup>` — для их группировки. Объединение переключателей в группу означает, что в любой момент времени в группе может быть установлен только один переключатель.



Группа из двух переключателей. В любой момент времени может быть установлен только один переключатель.

В приложении Bits and Pizzas должны отображаться переключатели Diavolo (острый соус) и Funghi (грибы). Код выглядит так:

```
<RadioGroup
    android:id="@+id/pizza_group"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <RadioButton android:id="@+id/radio_diavolo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Diavolo" />

    <RadioButton android:id="@+id/radio_funghi"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Funghi" />
</RadioGroup>
```

После того как вы определите группу и переключатели, можно написать код Kotlin для получения выбранного переключателя с использованием свойства `checkedRadioButtonId` группы. Его значением является идентификатор выбранного переключателя, или –1, если ни один переключатель не был выбран:

```
val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
val pizzaType = pizzaGroup.checkedRadioButtonId
if (pizzaType == -1) {
    //Выбранного элемента нет
} else {
    val radio = view.findViewById<RadioButton>(id)
    //Сделать что-то с переключателем
}
```

Это свойство позволяет узнать, какой переключатель был выбран (если он есть).

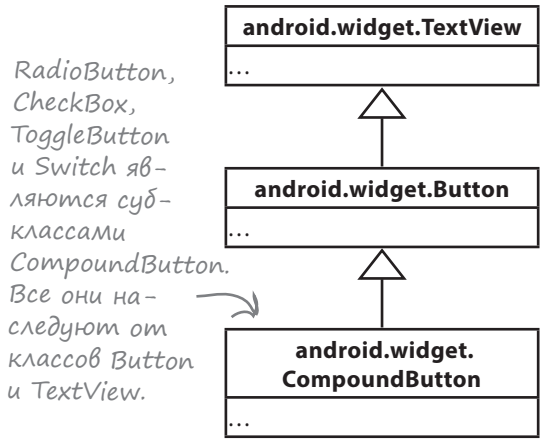
RadioGroup является субклассом LinearLayout, поэтому с группами переключателей можно использовать те же атрибуты, что и с линейными макетами.

Переключатель — разновидность композитной кнопки

Во внутренней реализации переключатели наследуют от класса с именем `CompoundButton`, а тот является субклассом `Button`. `CompoundButton` представляет композитную кнопку с двумя состояниями: установленным и снятым (включенным и выключенным).

Android включает другие разновидности композитных кнопок (помимо переключателей): **флажки**, **селекторы** и **выключатели**. Эти представления хорошо подходят для решений «да/нет»: например, «Добавить ли масло чили?» или «Хотите ли вы двойной пармезан?»

Для добавления флажков, селекторов и выключателей в макеты используется код следующего вида:



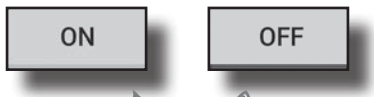

```
<CheckBox
    android:id="@+id/parmesan"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Parmesan" />
```

Установленный флажок.

```
<Switch
    android:id="@+id/switch_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Селектор может находиться в установленном и снятом состоянии.

```
<ToggleButton
    android:id="@+id/toggle_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="On"
    android:textOff="Off" />
```



Выключатель в обоих состояниях.

Свойство `isChecked` каждого представления можно использовать в коде Kotlin, чтобы узнать, находится ли он в установленном состоянии, как в следующем примере:

```
val parmesan = view.findViewById<CheckBox>(R.id.parmesan)
if (parmesan.isChecked) {
    //Что-то сделать
} else {
    //Сделать что-то другое
}
```

По свойству `isChecked` можно определить, находится ли заданная композитная кнопка `CompoundButton` в установленном (включенном) состоянии.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Плашка — разновидность композитной кнопки

До настоящего момента мы рассматривали использование различных видов композитных кнопок — переключателей, флажков и селекторов. Еще одну, более гибкую разновидность композитных кнопок составляют **плашки**. Это представление доступно при условии, что вы используете тему из библиотеки Material, например `Theme.MaterialComponents.DayNight.NoActionBar`. Как и другие разновидности композитных кнопок, плашки используются для принятия решений «да/нет», но у них есть и другое применение: они также могут использоваться для ввода, фильтрации данных и выполнения действий.

Это плашки (chips). Здесь плашка Parmesan включена, а плашка chili oil отключена.

В приложении Bits and Pizzas плашки будут использоваться для того, чтобы пользователь мог выбрать добавки (дополнительную порцию пармезана или масла чили) для своей пиццы.

Для добавления плашек в макет используется код следующего вида:

```
<com.google.android.material.chip.Chip
    android:id="@+id/parmesan"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Parmesan"
    style="@style/Widget.MaterialComponents.Chip.Choice"/>
```

Плашка Parmesan оформляется стилем Choice, чтобы она изменяла цвет при выборе.

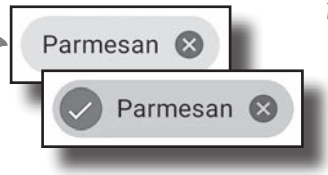
Ключевой частью кода плашки является атрибут `style`, управляющий ее внешним видом. Код

```
style="@style/Widget.MaterialComponents.Chip.Choice"
```

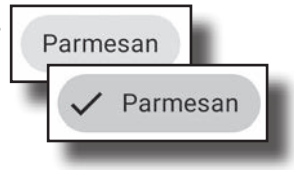
в приведенном примере назначает для плашки стиль Choice, чтобы ее цвет изменялся при выборе. Другие возможные варианты — Entry (позволяет использовать плашки для ввода данных), Filter (для плашек, предназначенных для фильтрации данных) и Action (действует как кнопка).

Вот как выглядят плашки Entry, Filter и Action:

Плашка ввода (Entry).

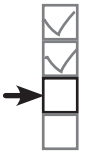


Плашка Filter.



Внешний вид плашки действия (Action) не изменяется при щелчке на ней.

Помимо добавления одиночных плашек в макет, вы также можете группировать несколько плашек. Давайте посмотрим, как это делается.



Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Объединение нескольких плашек в группу

Если вы хотите, чтобы ваш макет включал несколько плашек, имеющих сходное назначение, включите их в **группу плашек**. Группа плашек — разновидность группы представлений, предназначенная для аккуратного размещения нескольких плашек.

В приложении Bits and Pizzas используются две плашки добавок: для пармезана и для масла чили. Мы объединим их в группу следующим кодом:

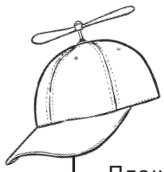
```
<com.google.android.material.chip.ChipGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <com.google.android.material.chip.Chip
        android:id="@+id/parmesan"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Parmesan"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>

    <com.google.android.material.chip.Chip
        android:id="@+id/chili_oil"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Chili oil"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>
</com.google.android.material.chip.ChipGroup>
```

Плашки аккуратно размещены внутри группы плашек.

Используйте isChecked для проверки состояния плашки

После того как вы добавите плашки в свой макет, для проверки, находится ли плашка в установленном состоянии, можно воспользоваться свойством `isChecked` каждой плашки — как и для других разновидностей композитных кнопок (селекторов, выключателей, флажков). Например, следующий код проверяет, выбрана ли плашка *parmesan*:

```
val parmesan = view.findViewById<Chip>(R.id.parmesan)
if (parmesan.isChecked) {
    //Что-то сделать
}
```



Серьезное программирование

Плашки чрезвычайно гибки и поддерживают множество параметров конфигурации. За дополнительной информацией обращайтесь по адресу

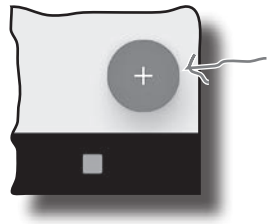
https://material.io/develop/android/components/chips

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

FAB — плавающая кнопка действия

Осталось последнее представление, о котором необходимо узнать, прежде чем мы сможем завершить макет `OrderFragment`: **FAB**.

FAB, или плавающая кнопка действия (**Floating Action Button**), круглая кнопка, которая «парит» над пользовательским интерфейсом. Она используется для привлечения внимания к стандартным или важным действиям. Как и обычную кнопку, кнопку FAB можно заставить реагировать на щелчки, назначив ей слушатель `OnClickListener` в коде Kotlin.

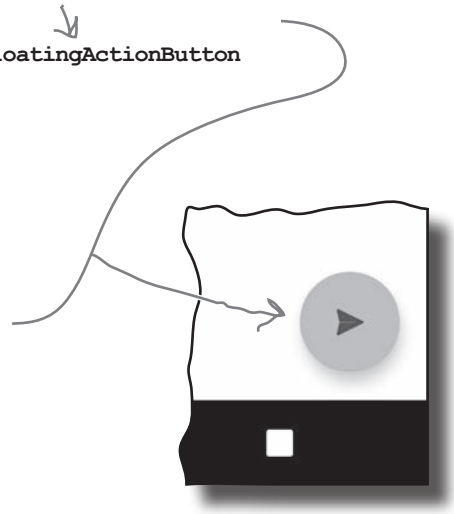


Кнопка FAB, используемая в приложении Google Contacts для добавления контакта.

Для добавления FAB в макет используется код следующего вида:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
    ...
    <androidx.core.widget.NestedScrollView...>
        ...
    </androidx.core.widget.NestedScrollView>

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="16dp"
        android:src="@android:drawable/ic_menu_send" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Добавляет круглую кнопку FAB со значком. Также существует тип ExtendedFloatingActionButton, отображающий значок с текстом.



Приведенный выше код использует атрибут `layout_gravity` для закрепления FAB в нижнем углу экрана устройства с отступом 16dp.

Строка:

```
android:src="@android:drawable/ic_menu_send"
```

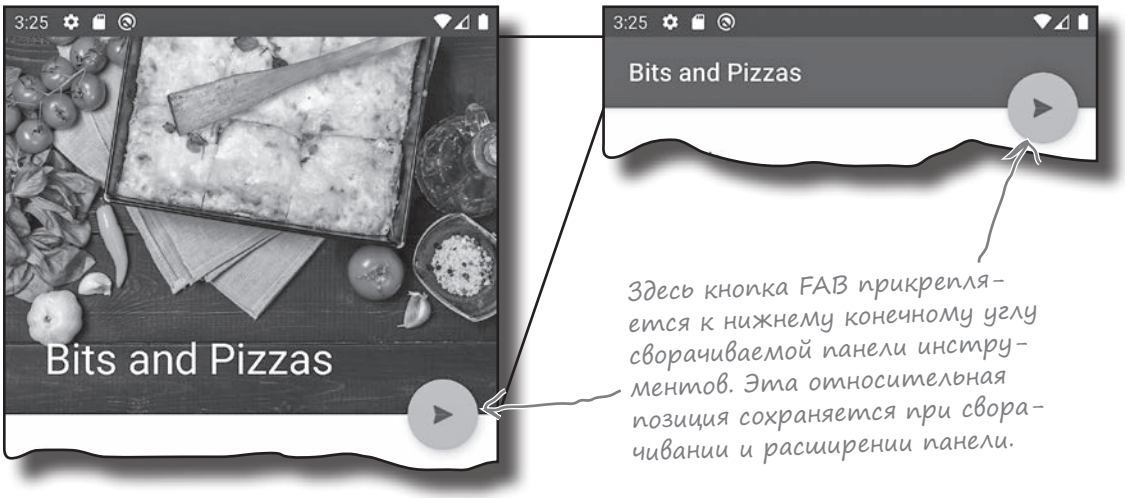
добавляет значок в FAB. В приведенном примере отображается один из встроенных значков Android с именем *ic_menu_send*, но вы можете использовать любую разновидность графического объекта при условии, что он помещается на FAB.

При добавлении атрибута src в FAB редактор кода позволяет просмотреть набор встроенных значков Android. Если ни один вариант вам не подходит, множество других вариантов можно найти по адресу https://material.io/resources/icons

Обычно кнопки FAB используются внутри координирующего макета, чтобы вы могли координировать перемещения между разными представлениями вашего макета. Рассмотрим пример.

FAB можно прикрепить к сворачиваемой панели инструментов

Кнопки FAB часто размещаются в нижнем конечном углу экрана, но их также можно прикрепить к другому представлению, например сворачиваемой панели инструментов. Когда вы это делаете, FAB перемещается со сворачиваемой панелью инструментов при ее сворачивании и расширении:



Код макета приведен ниже. Как видите, в нем используются атрибуты FAB `app:layout_anchor` и `app:layout_anchorGravity` для прикрепления FAB к нижнему конечному углу сворачиваемой панели инструментов:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
    ...
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_anchor="@id/collapsing_toolbar"
        app:layout_anchorGravity="bottom|end"
        android:layout_margin="16dp"
        android:src="@android:drawable/ic_menu_send" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

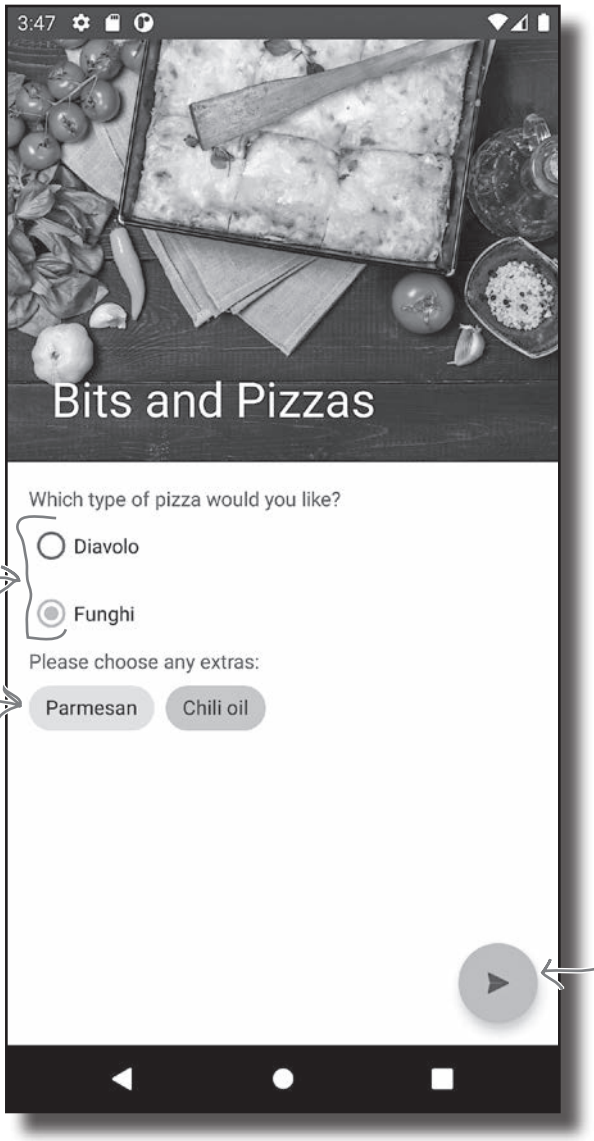
Эти строки прикрепляют FAB к нижнему конечному углу сворачиваемой панели инструментов с идентификатором collapsing_toolbar.

После знакомства с переключателями, плашками, FAB и другими представлениями можно переходить к построению основного содержимого макета `OrderFragment`.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Построение макета OrderFragment

В макет фрагмента `OrderFragment` необходимо добавить представления, чтобы пользователь мог выбрать вид пиццы и включить добавки. Макет должен выглядеть примерно так:



Сворачиваемая панель инструментов уже добавлена.

Необходимо добавить группу с двумя переключателями, чтобы пользователь мог выбрать вид пиццы.

Эти плашки (объединенные в группу) предназначены для выбора добавок.

Когда пользователь выберет все настройки, он щелкает на FAB.

Весь код, необходимый для создания макета, вам уже знаком, поэтому мы можем перейти к обновлению *fragment_order.xml*. Полный код будет приведен через несколько страниц.

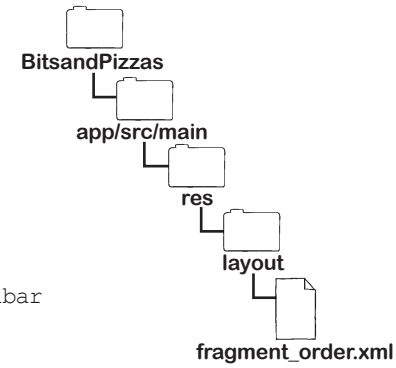
Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Полный код fragment_order.xml

Ниже приведен полный код макета фрагмента `OrderFragment`. Обновите код файла *fragment_order.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:layout_width="match_parent"
            android:layout_height="300dp"
            app:layout_scrollFlags="scroll|exitUntilCollapsed"
            app:contentScrim="?attr/colorPrimary">

            <ImageView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                android:src="@drawable/restaurant"
                app:layout_collapseMode="parallax"/>

            <com.google.android.material.appbar.MaterialToolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin" />
        </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>
```

Вам не придется изменять код, приведенный на этой странице.

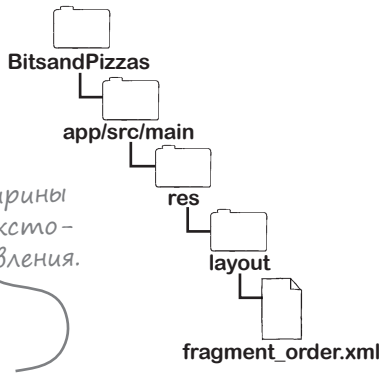


Продолжение на следующей странице.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Полный код fragment_order.xml (продолжение)

```
<androidx.core.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="16dp"
        android:orientation="vertical">

        <TextView
            android:layout_width="match_parentwrap_content"
            android:layout_height="match_parentwrap_content"
            android:text="This is the order fragment"
            android:text="Which type of pizza would you like?" />

        <RadioGroup
            android:id="@+id/pizza_group"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content">

            <RadioButton android:id="@+id/radio_diavolo"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Diavolo" />

            <RadioButton android:id="@+id/radio_funghi"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Funghi" />
        </RadioGroup>

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Please choose any extras:" />
```



Представление вложенной прокрутки может иметь только одного прямого потомка, поэтому все представления заключаются в линейный макет.

Изменение ширины и высоты текстового представления.

Замена выводимого текста. В реальном приложении все строки следовало бы разместить в файле строковых ресурсов.

Два переключателя для двух видов пиццы объединяются в группу переключателей.

Второе текстовое представление для выбора добавок.

Продолжение на следующей странице.

Полный код fragment_order.xml (продолжение)

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

```
<com.google.android.material.chip.ChipGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <com.google.android.material.chip.Chip
        android:id="@+id/parmesan"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Parmesan"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>

    <com.google.android.material.chip.Chip
        android:id="@+id/chili_oil"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Chili oil"
        style="@style/Widget.MaterialComponents.Chip.Choice"/>
</com.google.android.material.chip.ChipGroup>
```

Две плашки добавляются в группу плашек.

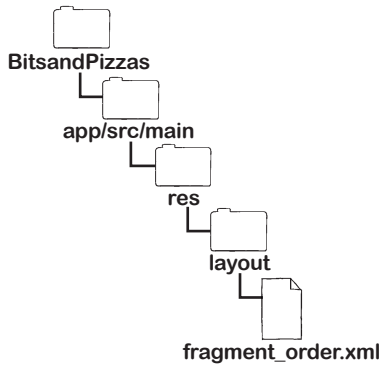
Плашки форматируются как плашки выбора, чтобы при выделении они изменяли цвет.

```
    </LinearLayout>
</androidx.core.widget.NestedScrollView>
```

Конец линейного макета.

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="16dp"
    android:src="@android:drawable/ic_menu_send" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

FAB добавляется у нижнего края экрана.



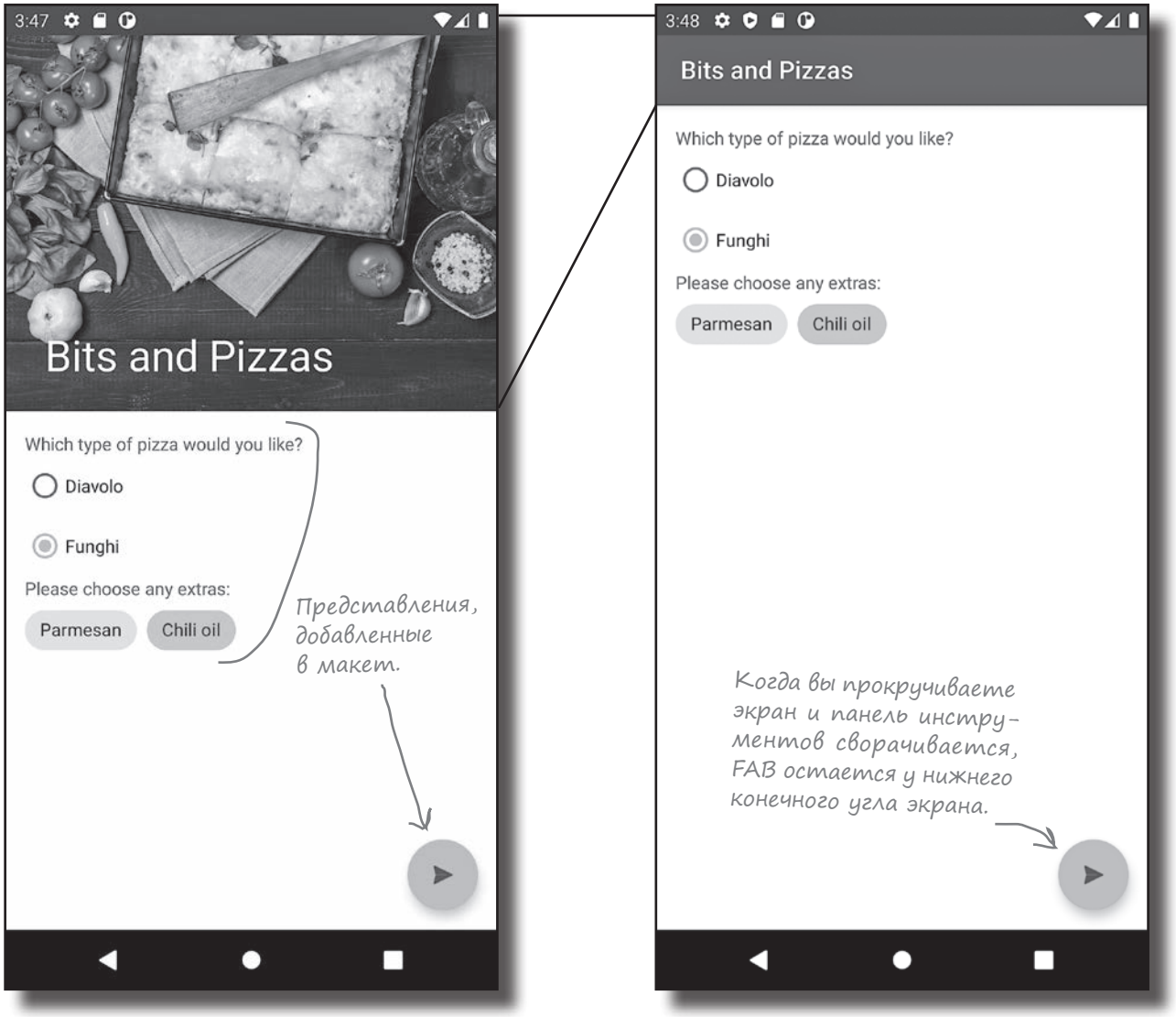
И это весь код макета, который нам понадобится. Давайте проведем тест-драйв приложения и посмотрим, как оно выглядит.

Тест-драйв

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

При запуске приложения в `MainActivity` отображается фрагмент `OrderFragment`. Он включает сворачиваемую панель инструментов, как и прежде, но на этот раз в основном содержимом присутствуют текстовые представления, переключатели, плашки и FAB.

Когда вы прокручиваете экран устройства, основное содержимое прокручивается вверх, и панель инструментов сворачивается. Кнопка FAB остается зафиксированной в правом конечном углу экрана.



Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

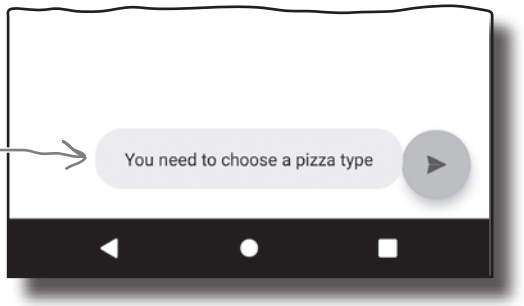
Как заставить FAB реагировать на щелчки

Мы построили макет `OrderFragment`, но когда пользователь щелкает на кнопке FAB, ничего не происходит. Обновим код Kotlin фрагмента, чтобы кнопка FAB реагировала на щелчки.

FAB должна решать две задачи:

1. **Если вид пиццы не выбран, вывести сообщение.**
Пользователь должен выбрать, какой вид пиццы он желает заказать. Если он щелкнет на FAB, *не* выбрав его, на экране появляется всплывающее сообщение (**Toast**) с предложением выбрать вид пиццы.

Такая разновидность сообщений называется Toast.



2. **Вывести заказ в отдельном сообщении.**
Если пользователь выбрал вид пиццы, будет выводиться сообщение с описанием заказа. Для этой цели будет использоваться другая разновидность всплывающих сообщений — **Snackbar**.

В реальном приложении по щелчку на FAB приложение должно размещать заказ пиццы. Здесь мы просто хотим, чтобы кнопка FAB что-то делала (к тому же это хороший повод для того, чтобы рассказать вам о всплывающих сообщениях).

Эта разновидность сообщений называется Snackbar.

Для начала заставим FAB реагировать на щелчки.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Добавление OnClickListener к FAB

Как было сказано выше, реакция FAB на щелчки реализуется так же, как и для других видов кнопок: присоединением `OnClickListener` к FAB.

Ниже приведен код добавления `OnClickListener` к FAB в `OrderFragment`; как видите, он не отличается от кода, используемого для обычных кнопок:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    //Код, выполняемый по щелчку на FAB
}
```

Чтобы заставить кнопку FAB реагировать на щелчки, нужно получить ссылку на нее и вызвать метод setOnClickListener.

Теперь сделаем так, чтобы слушатель `OnClickListener` выводил сообщение, если пользователь не выбрал вид пиццы.

Проверка выбранного вида пиццы

Чтобы узнать, выбрал ли пользователь вид пиццы, можно воспользоваться свойством `checkedRadioButtonId` группы `pizzas_group`. Значение свойства содержит идентификатор выбранного переключателя, если он выбран, или –1, если пользователь еще не принял решение.

Следующий код проверяет, не щелкнул ли пользователь на FAB, не выбрав вид пиццы:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
    val pizzaType = pizzaGroup.checkedRadioButtonId
    if (pizzaType == -1) {
        //Пицца не выбрана, вывести сообщение
    } else {
        //Вывести заказ в другом сообщении
    }
}
```

Если pizzaType содержит -1, значит, ни один переключатель в группе не выбран, поэтому выводится сообщение.

Получает значение свойства checkedRadioButtonId группы переключателей.

Если пользователь установил один из переключателей, выводится другое сообщение.

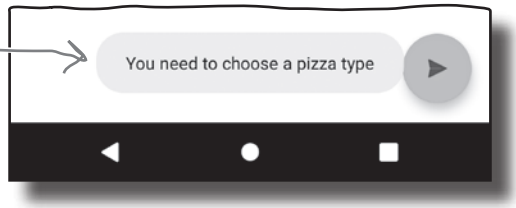
Если пользователь не выбрал вид пиццы, на экране должно появиться всплывающее сообщение **Toast**. Посмотрим, как это делается.

- Панель с прокруткой
- Сворачиваемая панель
- Представления
- **Реакция на щелчки**

Toast — простое всплывающее сообщение

Если пользователь щелкает на FAB, не выбрав вид пиццы, на экране устройства должно появиться сообщение Toast. Toast – простое всплывающее сообщение, которое предоставляет пользователю информацию и автоматически исчезает по истечении заданного времени:

Сообщение Toast, которое мы создадим.

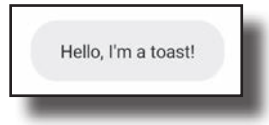


Сообщение создается вызовом `Toast.makeText()`. Метод `makeText` получает три параметра: `Context` (обычно `this` или активность в зависимости от того, вызывается ли Toast из активности или фрагмента), `CharSequence!` (выводимое сообщение) и промежуток времени. Затем сообщение отображается на экране вызовом метода `show()` сообщения.

Следующий пример кода на короткое время выводит сообщение Toast на экран:

```
val text = "Hello, I'm a toast!"
Toast.makeText(activity, text, Toast.LENGTH_SHORT).show()
```

Код отображает это сообщение Toast.



Часто Задаваемые Вопросы

В: Когда я запускаю приложения в эмуляторе, сообщения Toast не появляются на экране. С чего бы?

О: Прежде всего убедитесь в том, что вы вызываете метод `show()` сообщения Toast. Без этого Toast не появится.

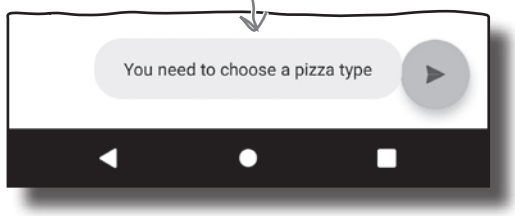
Если эта мера не помогла, возможно, вы столкнулись с известной проблемой эмулятора, из-за которой сообщения Toast иногда пропадают. Попробуйте закрыть эмулятор и откройте AVD Manager из меню Tools. Выберите виртуальное устройство, а затем вариант Wipe Data из его списка действий.

Добавление Toast к слушателю OnClickListener

Сообщение Toast должно выводиться в том случае, если пользователь щелкает на FAB без выбора вида пиццы. Код решения этой задачи приведен ниже:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
    val pizzaType = pizzaGroup.checkedRadioButtonId
    if (pizzaType == -1) {
        val text = "You need to choose a pizza type"
        Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
    } else {
        //Вывести заказ в другом сообщении
    }
}
```

Это сообщение будет отображаться, если пользователь не выбрал вид пиццы.



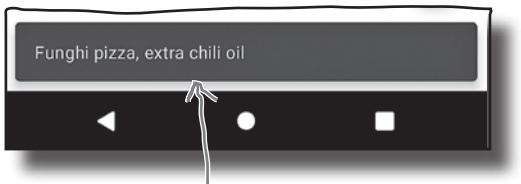
Вот и все, что необходимо сделать, если пользователь не выбрал вид пиццы. Теперь напишем код для вывода заказа.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Вывод заказа в сообщении Snackbar

Если пользователь выбрал вид пиццы, то его заказ по щелчку на FAB должен выводиться во всплывающем сообщении, которое называется **Snackbar**. Сообщение Snackbar похоже на Toast, но обладает большей интерактивностью. Например, его можно смахнуть с экрана или заставить что-то делать по щелчку.

Сообщения Snackbar создаются вызовом `Snackbar.make()`. Метод `make` получает три параметра: представление, инициирующее появление Snackbar (в данном случае FAB), `CharSequence!` (выводимый текст) и промежуток времени. Чтобы сообщение Snackbar появилось на экране, вызовите его метод `show()`.

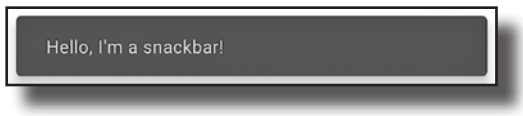


В сообщении Snackbar будет выводиться вид пиццы и все добавки, выбранные пользователем.

Следующий пример на короткое время выводит сообщение на экран:

```
val text = "Hello, I'm a snackbar!"
Snackbar.make(fab, text, Snackbar.LENGTH_SHORT).show()
```

Код выводит это сообщение Snackbar.



В приведенном выше коде значение `LENGTH_SHORT` используется для отображения Snackbar на короткое время. Другие возможные варианты — `LENGTH_LONG` (сообщение выводится на долгое время) и `LENGTH_INDEFINITE` (сообщение остается на экране неопределенно долго).

У Snackbar могут быть действия

При желании к Snackbar можно добавить действие, чтобы пользователь мог, скажем, отменить только что выполненную операцию. Для этого перед вызовом `show()` следует вызвать метод `setAction()` сообщения Snackbar. `setAction` получает два параметра: текст, который должен выводиться для действия, и лямбда-выражение, которое выполняется по щелчку на действии. Пример кода Snackbar с действием:

Необходимо указать, что должно происходить, если пользователь щелкает на действии Undo.

```
Snackbar.make(fab, text, Snackbar.LENGTH_SHORT)
        .setAction("Undo") {
            //Код, выполняемый по щелчку на кнопке
        }
        .show()
```

Сообщения Snackbar обычно отображаются в нижней части экрана, но вы можете переопределить это поведение при помощи метода `setAnchorView()` сообщения. Метод прикрепляет Snackbar к конкретному представлению, чтобы сообщение выводилось над ним. Например, такая возможность может оказаться полезной, если сообщение Snackbar должно находиться над нижней панелью навигации:

```
Snackbar.make(fab, text, Snackbar.LENGTH_SHORT)
    .setAnchorView(bottomNavBar)
    .show()
```

Фиксирует сообщение Snackbar, чтобы оно отображалось над представлением bottomNavBar.

Код Snackbar для заказа пиццы

Теперь вы знаете, как создаются сообщения Snackbar, и мы можем написать код для вывода заказанной пиццы. Мы будем отображать сообщение Snackbar с видом пиццы, выбранным пользователем, а также всеми добавками (например, пармезан или масло чили).

Код вывода сообщения Snackbar:

```
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
    val pizzaType = pizzaGroup.checkedRadioButtonId
    if (pizzaType == -1) {
        val text = "You need to choose a pizza type"
        Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
    } else {
        var text = (when (pizzaType) {
            R.id.radio_diavolo -> "Diavolo pizza"
            else -> "Funghi pizza"
        })
        val parmesan = view.findViewById<Chip>(R.id.parmesan)
        text += if (parmesan.isChecked) ", extra parmesan" else ""
        val chiliOil = view.findViewById<Chip>(R.id.chili_oil)
        text += if (chiliOil.isChecked) ", extra chili oil" else ""
        Snackbar.make(fab, text, Snackbar.LENGTH_LONG).show()
    }
}
```

Указать, какая пицца была выбрана.

Включает все выбранные добавки.

Выводит текст в Snackbar.

И это весь код Kotlin, необходимый для приложения Bits and Pizzas. Рассмотрим полный код *OrderFragment.kt* и проведем тест-драйв приложения.

Часто Задаваемые Вопросы

В: Объекты Toast и Snackbar тоже являются представлениями? Я могу включать их в свои макеты?

О: Нет, они не являются представлениями, так что их можно вывести на экран только с использованием кода Kotlin.

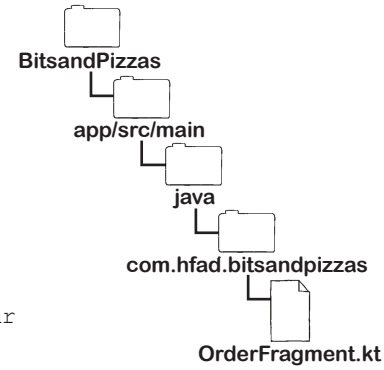
В: Можно ли использовать несколько FAB в одном макете?

О: Лучше ограничиться одной кнопкой FAB на экран. FAB используются для важных действий, и если их будет слишком много, они перестанут привлекать внимание.

Панель с прокруткой
Сворачиваемая панель
Представления
Реакция на щелчки

Полный код OrderFragment.kt

Ниже приведен полный код *OrderFragment.kt*; обновите свою версию (изменения выделены жирным шрифтом):




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.appbar.MaterialToolbar
import android.widget.RadioGroup
import com.google.android.material.chip.Chip
import com.google.android.material.floatingactionbutton.FloatingActionButton
import android.widget.Toast
import com.google.android.material.snackbar.Snackbar

class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_order, container, false)
        val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
        (activity as AppCompatActivity).setSupportActionBar(toolbar)

        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        fab.setOnClickListener {
            val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
            val pizzaType = pizzaGroup.checkedRadioButtonId
            if (pizzaType == -1) {
                val text = "You need to choose a pizza type"
                Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
            } else {
                var text = (when (pizzaType) {
                    R.id.radio_diavolo -> "Diavolo pizza"
                    else -> "Funghi pizza"
                })
```

Дополнительные классы, используемые в программе, необходимо импортировать.

Добавление слушателя OnClickListener к FAB.

Если пользователь щелкнул на кнопке FAB, не выбрав вид пиццы, отображается сообщение Toast.

Начало построения текста заказа.

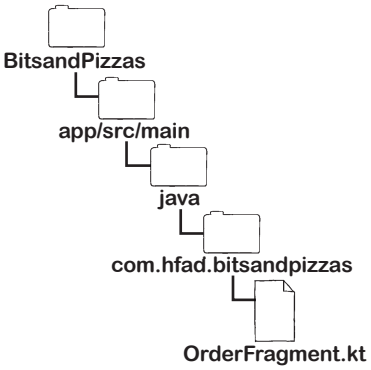
Продолжение на следующей странице.

Полный код OrderFragment.kt (продолжение)

```
            val parmesan = view.findViewById<Chip>(R.id.parmesan)
            text += if (parmesan.isChecked) ", extra parmesan" else ""
            val chiliOil = view.findViewById<Chip>(R.id.chili_oil)
            text += if (chiliOil.isChecked) ", extra chili oil" else ""
            Snackbar.make(fab, text, Snackbar.LENGTH_LONG).show()
        }
    }
    return view
  }
}
```

В текст включаются добавки.

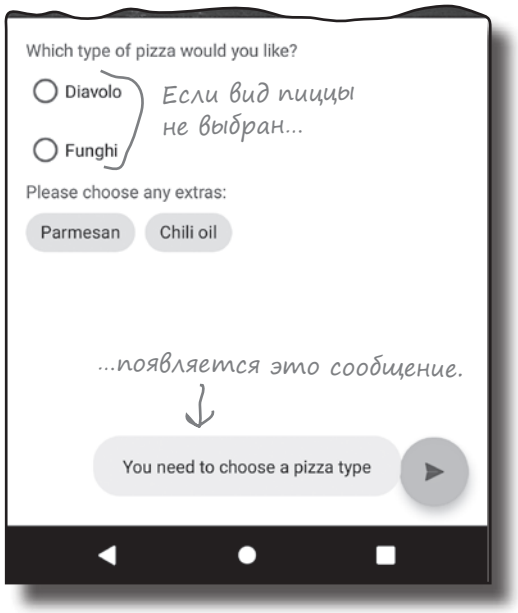
Создание и отображение Snackbar.

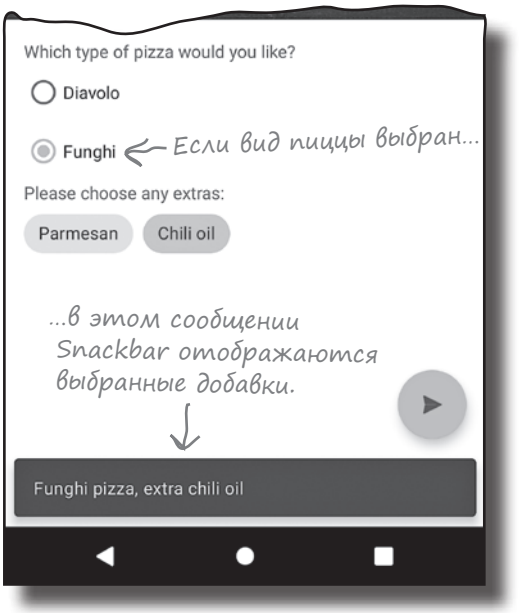


Тест-драйв

Если щелкнуть на FAB, не выбрав вид пиццы, появляется сообщение Toast с предложением выбрать его.

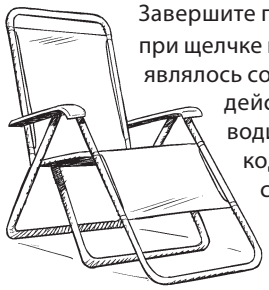
Если выбрать вид пиццы и снова щелкнуть на FAB, в нижней части экрана появляется сообщение Snackbar с подробной информацией о заказе.





Поздравляем! Теперь вы знаете, как заставить FAB реагировать на щелчки выводом всплывающего сообщения.

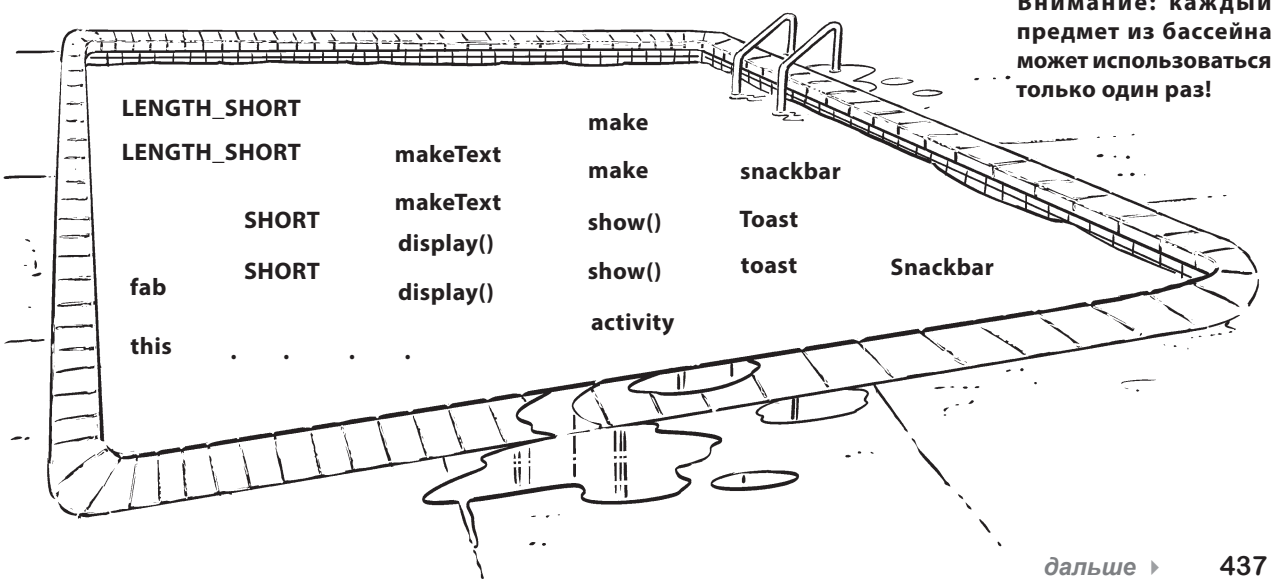
У бассейна



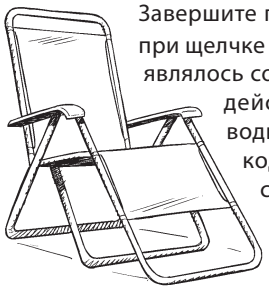
Завершите приведенный ниже фрагмент кода, чтобы при щелчке на кнопке FAB с идентификатором `fab` появилось сообщение Snackbar. Оно должно включать действие «Undo», по щелчку на котором выводится сообщение Toast. Выловите сегменты кода из бассейна и разместите их в пустых строках в коде. Каждый сегмент может использоваться только один раз; использовать все сегменты не обязательно.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_order, container, false)
    val fab = view.findViewById<FloatingActionButton>(R.id.fab)
    fab.setOnClickListener {
        Snackbar. ..........(fab, "Your order has been updated",  ..........)
                .setAction("Undo") {
                    Toast.  ..........(... ........,..."Undone!",  ..........)
                            ..........
                }
                ..........
    }
    return view
}
```

Внимание: каждый предмет из бассейна может использоваться только один раз!



У бассейна. Решение



Завершите приведенный ниже фрагмент кода, чтобы при щелчке на кнопке FAB с идентификатором `fab` появлялось сообщение Snackbar. Оно должно включать действие «Undo», по щелчку на котором выводится сообщение Toast. Выловите сегменты кода из бассейна и разместите их в пустых строках в коде. Каждый сегмент может использоваться только один раз; использовать все сегменты не обязательно.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_order, container, false)
    val fab = view.findViewById<FloatingActionButton>(R.id.fab)
    fab.setOnClickListener {
        Snackbar. make (fab, "Your order has been updated", Snackbar. LENGTH_SHORT )
            .setAction("Undo") {
                Toast. makeText ( activity , "Undone!", Toast . LENGTH_SHORT )
                    . show()
            }. show()
    }
    return view
}
```

Snackbar отображается на короткое время.

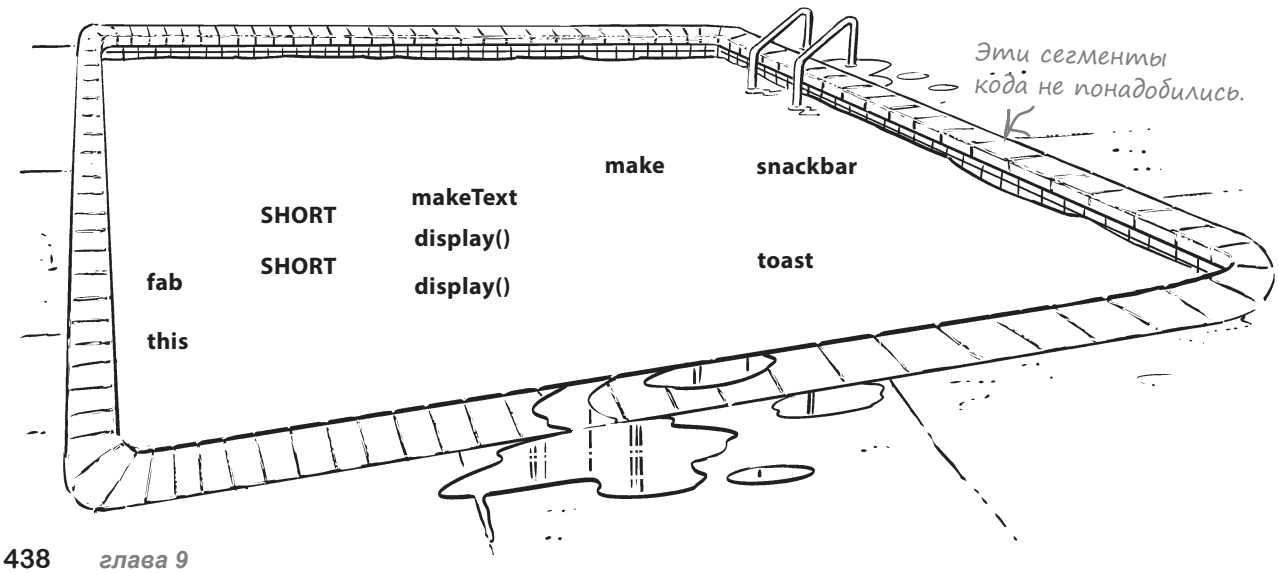
Метод make() используется для построения Snackbar.

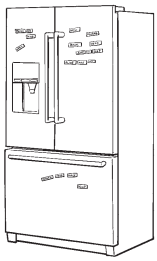
Метод makeText используется для построения Toast.

Отображает Toast.

Сообщение Toast отображается на короткое время с использованием контекста активности, в котором отображается фрагмент.

Отображает Snackbar.





Развлечения с магнитами. Решение

Кто-то попытался выложить на холодильнике структуру файла макета, реализующего сворачиваемую панель инструментов с изображением. К сожалению, магниты отвалились, когда большой птеродактиль пролетел рядом в поисках еды.

Удастся ли вам вернуть магниты на место в правильном порядке?

```
<androidx.coordinatorlayout.widget.CoordinatorLayout...>
```

CoordinatorLayout становится корневым элементом.

```
<com.google.android.material.appbar.AppBarLayout...>
```

AppBarLayout размещается внутри CoordinatorLayout и содержит CollapsingToolbarLayout.

```
<com.google.android.material.appbar.CollapsingToolbarLayout...>
```

```
<ImageView...>
```

ImageView и MaterialToolbar размещаются внутри CollapsingToolbarLayout.

```
</ImageView>
```

```
<com.google.android.material.appbar.MaterialToolbar...>
```

```
</com.google.android.material.appbar.MaterialToolbar>
```

```
</com.google.android.material.appbar.CollapsingToolbarLayout>
```

```
</com.google.android.material.appbar.AppBarLayout>
```

```
<androidx.core.widget.NestedScrollView...>
```

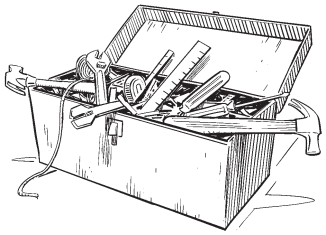
NestedScrollView размещается за пределами AppBarLayout.

```
</androidx.core.widget.NestedScrollView>
```

```
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

ГЛАВА 9

Ваш инструментарий Android



Глава 9 осталась позади, а ваш инструментарий пополнился новыми представлениями и компонентами.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Используйте `CoordinatorLayout` для координации анимации между представлениями.
- Используйте `AppBarLayout` для анимации панелей инструментов.
- Используйте `NestedScrollView`, если вы хотите, чтобы представления реагировали на прокрутку экрана устройства.
- Используйте `CollapsingToolbarLayout` для добавления панелей инструментов, которые сворачиваются и расширяются в соответствии с действиями пользователей.
- Чтобы определить группу переключателей, сначала создайте элемент `<RadioGroup>`, а затем поместите отдельные переключатели `<RadioButton>` в группу.
- Переключатели, выключатели, селекторы и флажки являются разновидностями композитных кнопок. Композитная кнопка может находиться в установленном или снятом состоянии.
- Плашка — более гибкая композитная кнопка, которая может использоваться для принятия решений, ввода данных пользователем, фильтрации данных и выполнения действий.
- Несколько плашек можно объединить в группу.
- Используйте FAB (плавающая кнопка действия) для отображения типичных или особенно важных действий пользователя.
- Toast — разновидность всплывающих сообщений.
- Snackbar — другая разновидность всплывающих сообщений, предназначенных для взаимодействия с пользователем.

Связанные одной целью

Пришло время попрощаться с findViewById(). Как вы, вероятно, уже заметили, чем больше у вас представлений и чем более интерактивным становится ваше приложение, тем больше вызовов ***findViewById()*** приходится использовать. И если вам надоело вызывать этот метод каждый раз, когда вам потребуется поработать с представлением, знайте: *вы не одиноки*. В этой главе вы узнаете, **как связывание представлений делает *findViewById()* пережитком прошлого**. Вы научитесь применять этот механизм в коде **активностей и фрагментов** и узнаете, почему этот подход предоставляет более безопасный и эффективный способ обращения к представлениям вашего макета. Итак, за дело!

Под капотом findViewById()

Как вам уже известно, каждый раз, когда вы хотите взаимодействовать с представлением в коде активности или фрагмента, сначала следует получить ссылку на него вызовом `findViewById()`. Например, следующий код активности получает ссылку на кнопку `Button` с идентификатором `start_button`, чтобы кнопка могла реагировать на щелчки:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val startButton = findViewById<Button>(R.id.start_button)
        startButton.setOnClickListener {
            //Код, который делает что-то полезное
        }
    }
}
```

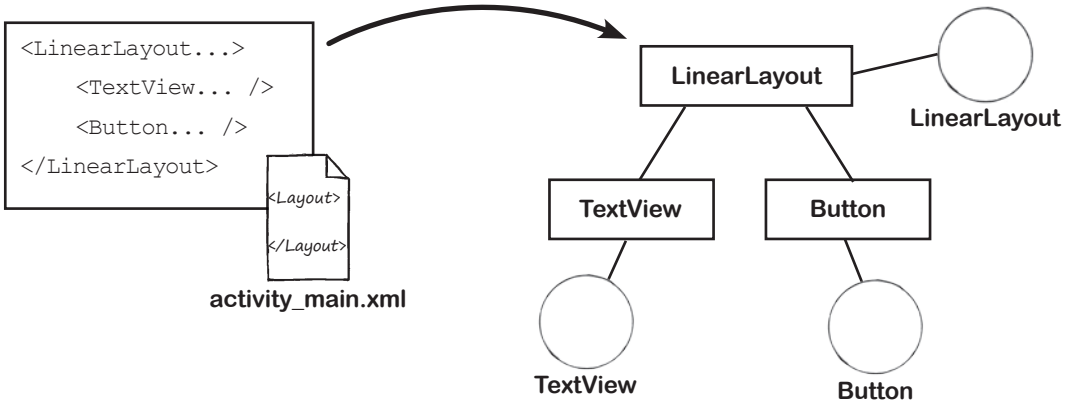
findViewById() получает ссылку на кнопку, чтобы вы могли вызывать ее методы.

Но что на самом деле происходит при вызове `findViewById()`?

findViewById() ищет представление в иерархии представлений

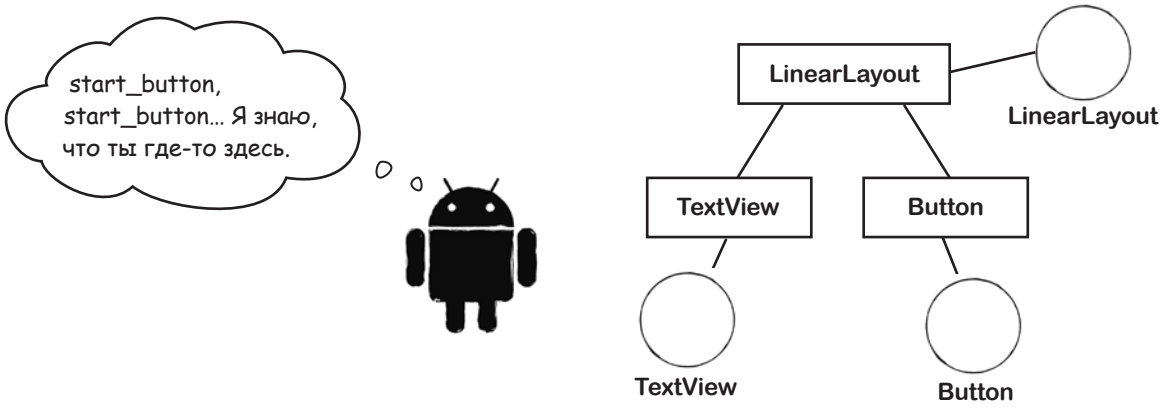
При выполнении этого кода происходит следующее:

1 **Файл макета MainActivity (activity_main.xml) заполняется, преобразуясь в иерархию объектов View.**
Если файл описывает линейный макет, содержащий текстовое представление и кнопку, то в результате заполнения будут созданы объекты `LinearLayout`, `TextView` и `Button`. `LinearLayout` становится корневым представлением этой иерархии.

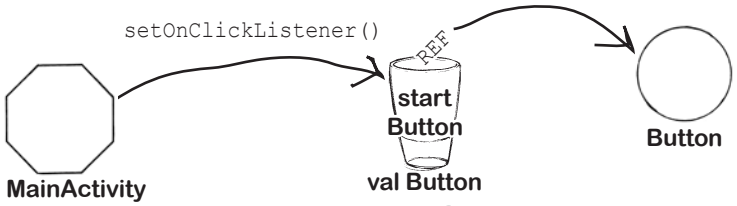


История продолжается

2 **Android ищет в иерархии представление с подходящим идентификатором.** Мы используем вызов `findViewById<Button>(R.id.start_button)`, поэтому Android ищет в иерархии объект `View` с идентификатором `start_button`.

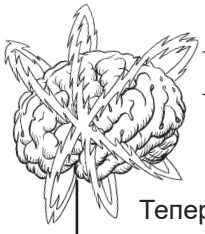


3 **Android возвращает объект View с искомым идентификатором, ближний к верху иерархии, и преобразует его к типу, заданному при вызове findViewById().** В данном случае `findViewById<Button>(R.id.start_button)` находит в иерархии первый объект `View` с идентификатором `start_button` и преобразует его к типу `Button`. Теперь `MainActivity` может взаимодействовать с объектом.



Переменная startButton содержит ссылку на объект Button, возвращенный методом findViewById().

Получается, что каждый раз, когда вы вызываете `findViewById()`, Android ищет в иерархии макета объект `View` с подходящим идентификатором и преобразует его к заданному типу.



Мозговой штурм

Теперь вы знаете, как работает `findViewById()`. Какие недостатки вы видите в этом подходе? Что, как вы думаете, можно было бы сделать иначе?

У findViewById() есть оборотная сторона

Хотя метод `findViewById()` удобен для получения ссылок на представления, у него есть ряд недостатков.

Удлинение кода
Чем больше представлений, с которыми вам придется взаимодействовать, тем больше вызовов придется сделать. Это может привести к удлинению кода и затруднит его чтение.

Неэффективность
Каждый раз, когда вы вызываете `findViewById()`, Android приходится искать в иерархии макета представление с подходящим идентификатором. Это неэффективно, особенно если макет состоит из множества представлений, образующих глубокую иерархию.

Небезопасность в отношении null
Метод `findViewById()` используется для поиска во время выполнения, а это означает, что компилятор не может проверить типичные ошибки.

Например, методу `findViewById()` может передаваться недействительный идентификатор, не существующий в макете. Если вы попытаетесь выполнить код

```
val message = view.findViewById<EditText>(R.id.message)
```

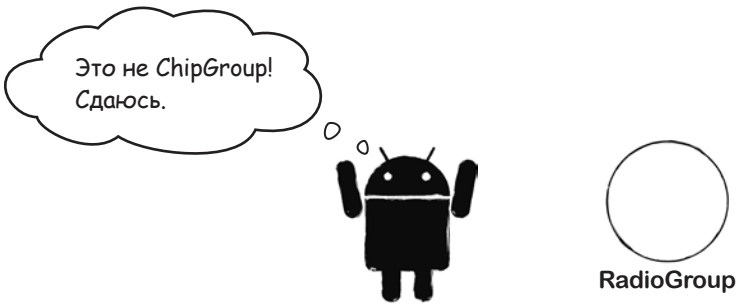
и в макете нет представления `View` с идентификатором `message`, выдается исключение null-указателя и в приложении происходит фатальный сбой.

Небезопасность в отношении типов
Другая проблема заключается в том, что компилятор не может проверить, правильно ли задан тип View, что может привести к исключению приведения типа.

Допустим, у вас имеется группа переключателей `pizza_group` и вы пытаетесь получить ссылку на нее кодом следующего вида:

```
val pizzaGroup = view.findViewById<ChipGroup>(R.id.pizza_group)
```

Хотя вместо `RadioGroup` задан тип `ChipGroup`, код все равно компилируется. Компилятор не проверяет тип на правильность при построении кода. Во время выполнения приложения будет выдано исключение приведения типа, и в приложении произойдет фатальный сбой.



Итак, если у `findViewById()` столько недостатков, то как выглядит альтернатива?

На помощь приходит связывание представлений

Связывание представлений — это часть Android Jetpack.

Вместо того чтобы вызывать `findViewById()` каждый раз, когда вам понадобится ссылка на `View`, можно воспользоваться **связыванием представлений**. Вы создаете объект связывания (об этом чуть позднее) и используете его для обращения к представлению.

Допустим, имеется макет, содержащий кнопку с идентификатором `start_button`. Если вы хотите, чтобы кнопка что-то делала по щелчку, можно получить ссылку на нее вызовом `findViewById()`:

До настоящего момента мы пользовались именно этим способом.

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
    //Код, который делает что-то полезное
}
```

Со связыванием представлений вам уже не придется вызывать `findViewById()` для получения ссылки на кнопку. Вместо этого можно использовать следующий код:

```
binding.startButton.setOnClickListener {
    //Код, который делает что-то полезное
}
```

Со связыванием представлений вы можете взаимодействовать с кнопкой при помощи свойства startButton объекта binding. Свойство содержит ссылку на представление с идентификатором start_button.

Он делает то же самое, но проще записывается, а ваш код становится короче и лучше читается.

Связывание представлений безопаснее и эффективнее findViewById()

При использовании связывания представлений Android уже не нужно искать подходящий объект `View` в иерархии: для обращения к нему просто используется объект связывания. Такой способ намного эффективнее `findViewById()`.

Другое преимущество заключается в том, что **компилятор предотвращает исключения null-указателей и приведения типов на стадии компиляции**. При обращении к представлениям с использованием объекта связывания компилятор знает, какие представления доступны и к каким типам они относятся. Он не позволит обратиться к несуществующему представлению, и вам уже не придется приводить представление к конкретному типу, потому что компилятору этот тип уже известен. В результате ваш код становится намного безопаснее.

Связывание представлений — безопасная по отношению к типам, более эффективная альтернатива для вызова findViewById().

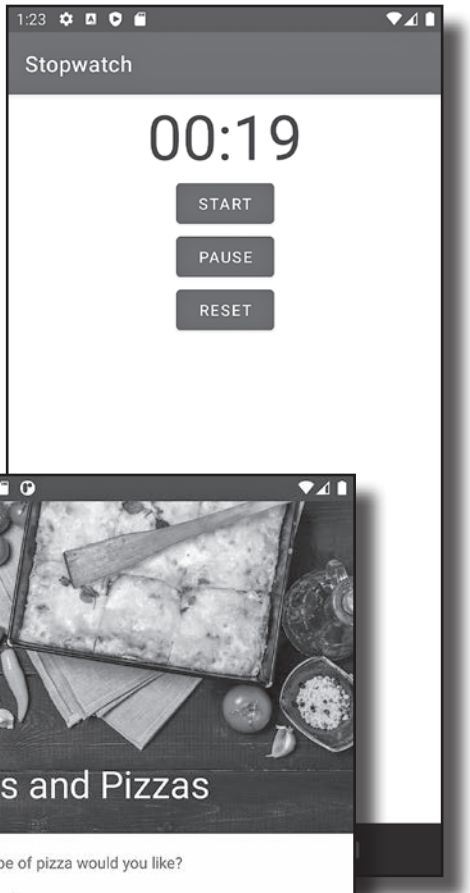
Итак, теперь вам известны преимущества связывания представлений. Разберемся в том, как же им пользоваться.

Как использовать связывание представлений

Код связывания представлений несколько отличается для активностей и фрагментов, и в этой главе мы продемонстрируем оба варианта. Основная последовательность действий выглядит так:

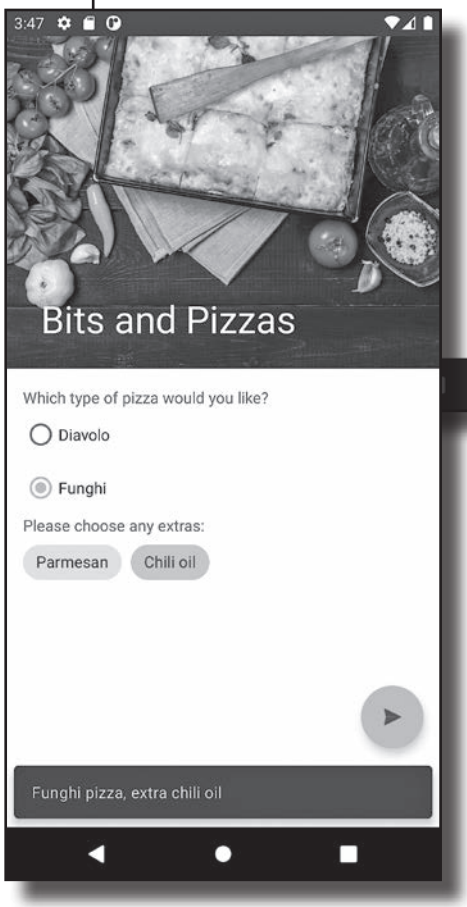
1. **Включение связывания представлений в код активности приложения Stopwatch.** В главе 5 мы построили приложение Stopwatch для изучения методов жизненного цикла Android. Вернемся к этому приложению и обновим его код активности, чтобы в нем использовалось связывание представлений.

В приложении Stopwatch в настоящее время используется одна активность без фрагментов. Изменим код активности, чтобы в нем использовалось связывание представлений.



2. **Включение связывания представлений во фрагмент приложения Bits and Pizzas.** Вернемся к приложению Bits and Pizzas, созданному в предыдущей главе, и посмотрим, как реализовать связывание представлений в его коде фрагмента.

Приложение Bits and Pizzas использует код фрагмента для взаимодействия с его представлениями. Изменим этот код, чтобы в нем использовалось связывание представлений.



За дело!

Код активности
Код фрагмента

Снова к приложению Stopwatch

Начнем с модификации приложения Stopwatch, созданного в главе 5. Откройте проект этого приложения.

Как вы, вероятно, помните, приложение Stopwatch выводит простой секундомер, который можно запускать, приостанавливать и сбрасывать тремя кнопками. Приложение выглядит примерно так:

В представлении Chronometer выводится количество прошедших секунд.

Если щелкнуть на кнопке Start, запускается отсчет времени.

Если щелкнуть на кнопке Pause, отсчет времени останавливается.

Если щелкнуть на кнопке Reset, показания секундомера обнуляются.

Не забудьте!

Мы обновим приложение Stopwatch, созданное в главе 5. Откройте проект этого приложения.

В приложении используется одна активность `MainActivity` с файлом макета *activity_main.xml*.

Каждый раз, когда ей потребуется взаимодействовать с одним из представлений, она вызывает `findViewById()` для получения ссылки на него. Например, чтобы кнопка Start реагировала на щелчки, используется код следующего вида:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
    //Код, выполняемый по щелчку на кнопке
}
```

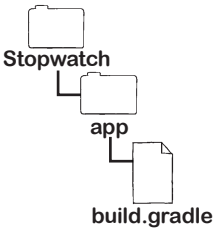
Давайте посмотрим, как обновить приложение, чтобы вместо `findViewById()` в нем использовалось связывание представлений.

Включение связывания представлений в файле build.gradle приложения

Чтобы использовать связывание представлений, сначала следует включить его в разделе `android` файла *build.gradle* приложения. Код, включающий связывание представлений, выглядит примерно так:

```
android {
    ...
    buildFeatures {
        viewBinding true
    }
}
```

Добавьте эти строки в раздел android файла build.gradle приложения.

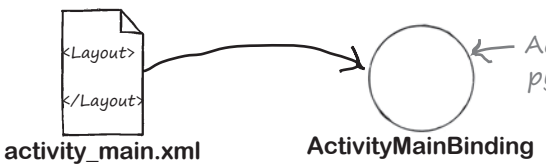


Мы собираемся использовать связывание представлений в приложении Stopwatch, поэтому убедитесь в том, что вы включили приведенное выше изменение в файл *Stopwatch/app/build.gradle*. Затем выберите команду Sync Now, чтобы синхронизировать изменения с остальными частями проекта.

Не забудьте синхронизировать эти изменения, иначе при попытке обновления кода активности будут обнаружены ошибки.

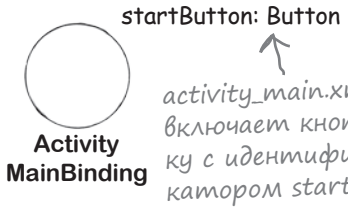
При включении связывания представлений генерируется код для каждого макета

Когда вы включаете связывание представлений, для каждого файла макета в приложении автоматически создается класс связывания. Например, приложение Stopwatch содержит файл макета с именем *activity_main.xml*, поэтому при включении связывания представлений автоматически генерируется класс связывания с именем `ActivityMainBinding`:



ActivityMainBinding генерируется из activity_main.xml.

Каждый класс связывания включает свойство для каждого представления в макете, обладающее идентификатором. Например, макет *activity_main.xml* включает кнопку с идентификатором `start_button`, поэтому класс связывания `ActivityMainBinding` включает свойство с именем `startButton` и типом `Button`.



activity_main.xml включает кнопку с идентификатором start_button, поэтому ActivityMainBinding включает свойство Button с именем startButton, содержащее ссылку на объект Button.

Классы связывания важны, потому что **представления макетов ассоциируются со свойствами класса связывания**. Вместо того чтобы вызывать `findViewById()` каждый раз, когда вам понадобится ссылка на представление, вы просто взаимодействуете со свойством этого представления в классе связывания.

Итак, мы включили связывание представлений в приложении Stopwatch. Давайте разберемся, как использовать его в коде `MainActivity`.

Как добавить связывание представлений в активность

Код использования связывания представлений будет практически одинаковым для всех активностей, которые вы будете создавать. Он выглядит примерно так:

```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.hfad.stopwatch.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)
    }
}
```

Импортируем класс связывания (в данном случае ActivityMainBinding).

Добавляет свойство связывания.

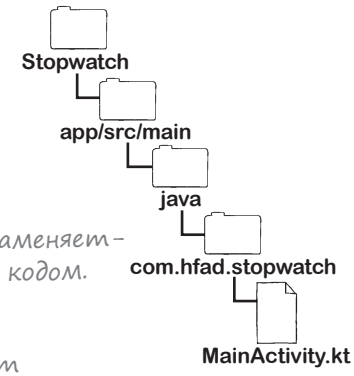
Эта строка заменяется следующим кодом.

Объявляет свойство.

view присваивается корневое представление.

Создает объект ActivityMainBinding, связанный с макетом.

Корневое представление передается setContentView().

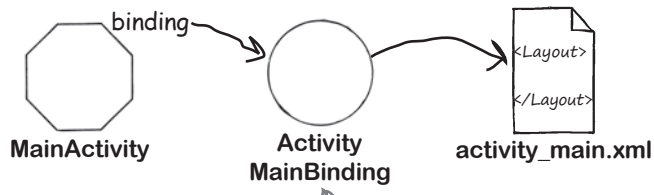


Приведенный выше код объявляет свойство с именем `binding` и типом `ActivityMainBinding`. Значение этого свойства задается в методе `onCreate()` активности, для чего используется следующий код:

```
binding = ActivityMainBinding.inflate(layoutInflater)
```

Эта строка необходима для получения объекта ActivityMainBinding.

Эта команда вызывает метод `inflate()` объекта `ActivityMainBinding`, который создает объект `ActivityMainBinding`, связанный с макетом активности.



Код:

```
val view = binding.root
setContentView(view)
```

получает ссылку на корневое представление объекта `binding` и использует метод `setContentView()` для его отображения.

Свойству binding объекта MainActivity присваивается объект ActivityMainBinding. Представления активности связываются с этим объектом.

После того как связывание представлений будет добавлено в активность, вы можете использовать свойство `binding` для взаимодействия с представлениями макета. Давайте посмотрим, как это делается.

Код активности
Код фрагмента

Использование свойства binding для взаимодействия с представлениями

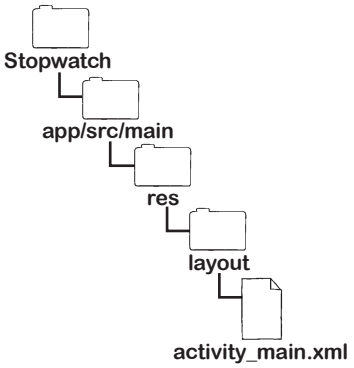
Текущий код `MainActivity` взаимодействует со своими представлениями для управления отсчетом времени и реакцией кнопок на щелчки. Мы обновим этот код, чтобы вместо вызовов `findViewById()` он обращался к представлениям через свойство `binding` активности.

Чтобы понять, как это работает, воспользуемся кнопкой Start активности `MainActivity` в качестве примера.

Код макета

Кнопка Start определяется в файле *activity_main.xml* следующим кодом:

```
...
<Button
    android:id="@+id/start_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/start" />
...
```



Как видите, ей присвоен идентификатор `start_button`.

Код активности

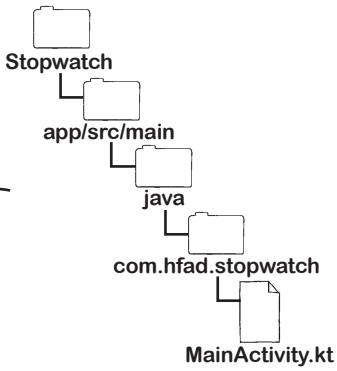
`MainActivity` обеспечивает реакцию кнопки на щелчки при помощи метода `findViewById()`:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
    //Код, выполняемый по щелчку на кнопке
}
```

При использовании связывания представлений этот код можно заменить следующим:

Свойство binding может использоваться для обращения к представлению start_button.

```
~~val startButton = findViewById<Button>(R.id.start_button)~~
binding.startButton.setOnClickListener {
    //Код, выполняемый по щелчку на кнопке
}
```



Код делает то же самое, что и исходный код, но использует свойство `binding` объекта `MainActivity` для взаимодействия с кнопкой.

Обновим полный код `MainActivity`, чтобы в нем использовалось связывание представлений.

Код активности
Код фрагмента

Полный код MainActivity.kt

Ниже приведен обновленный код `MainActivity`; измените свою версию *MainActivity.kt* (изменения выделены жирным шрифтом):

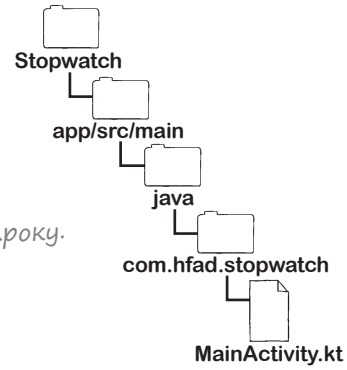
```
package com.hfad.stopwatch

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
~~import android.widget.Button~~
~~import android.widget.Chronometer~~
import com.hfad.stopwatch.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    ~~lateinit var stopwatch: Chronometer //Хронометр~~
    var running = false  //Хронометр работает?
    var offset: Long = 0  //Базовое смещение

    val OFFSET_KEY = "offset"
    val RUNNING_KEY = "running"
    val BASE_KEY = "base"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ~~setContentView(R.layout.activity_main)~~
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)
        ~~stopwatch = findViewById<Chronometer>(R.id.stopwatch)~~

        //Восстановление предыдущего состояния
        if (savedInstanceState != null) {
            offset = savedInstanceState.getLong(OFFSET_KEY)
            running = savedInstanceState.getBoolean(RUNNING_KEY)
            if (running) {
                binding.stopwatch.base = savedInstanceState.getLong(BASE_KEY)
                binding.stopwatch.start()
            } else setBaseTime()
        }
```

Эти директивы импортирования не нужны, удалите их.

Импортируем класс связывания.

Добавляем свойство binding.

Изначально было добавлено свойство stopwatch, но благодаря связыванию представлений оно стало ненужным.



Удалите эту строку.

Добавьте код связывания представлений.

Свойство stopwatch было удалено, поэтому присваивать ему значение не нужно.

Свойство binding используется для обращения к хронометру.

Продолжение на следующей странице.

Полный код MainActivity.kt (продолжение)

Код активности
Код фрагмента

```
        //Кнопка start запускает секундомер, если он не работал
        val startButton = findViewById<Button>(R.id.start_button)
        binding.startButton.setOnClickListener {
            if (!running) {
                setBaseTime()
                binding.stopwatch.start()
                running = true
            }
        }

        //Кнопка pause останавливает секундомер, если он работал
        val pauseButton = findViewById<Button>(R.id.pause_button)
        binding.pauseButton.setOnClickListener {
            if (running) {
                saveOffset()
                binding.stopwatch.stop()
                running = false
            }
        }

        //Кнопка reset обнуляет offset и базовое время
        val resetButton = findViewById<Button>(R.id.reset_button)
        binding.resetButton.setOnClickListener {
            offset = 0
            setBaseTime()
        }
    }

    override fun onPause() {
        super.onPause()
        if (running) {
            saveOffset()
            binding.stopwatch.stop()
        }
    }
```

Удалите эту строку.

Свойство binding используется для обращения к представлениям.

Stopwatch
app/src/main
java
com.hfad.stopwatch
MainActivity.kt

Удалите эту строку.

Здесь тоже используется свойство binding.

Здесь будет использоваться связывание представлений вместо вызова findViewById().

Здесь также используется связывание представлений.

Продолжение на следующей странице.

Полный код MainActivity.kt (продолжение)

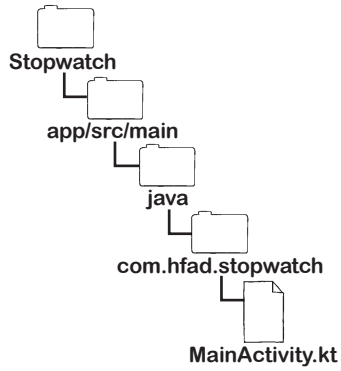
```
    override fun onResume() {
        super.onResume()
        if (running) {
            setBaseTime()
            binding.stopwatch.start()
            offset = 0
        }
    }

    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        savedInstanceState.putLong(OFFSET_KEY, offset)
        savedInstanceState.putBoolean(RUNNING_KEY, running)
        savedInstanceState.putLong(BASE_KEY, binding.stopwatch.base)
        super.onSaveInstanceState(savedInstanceState)
    }

    //Обновляет время stopwatch.base
    fun setBaseTime() {
        binding.stopwatch.base = SystemClock.elapsedRealtime() - offset
    }

    //Сохраняет offset
    fun saveOffset() {
        offset = SystemClock.elapsedRealtime() - binding.stopwatch.base
    }
}
```

Здесь используется связывание представлений.

Здесь тоже используется связывание представлений.

Здесь также должно использоваться свойство binding.

И здесь.



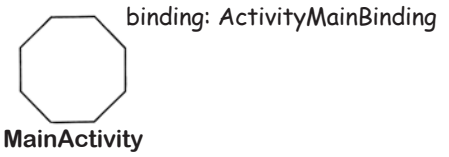
И это все изменения, которые необходимо внести в приложение Stopwatch, чтобы перевести его на связывание представлений. Давайте в общих чертах рассмотрим, что происходит при выполнении кода, и проведем его тест-драйв.

Код активности
Код фрагмента

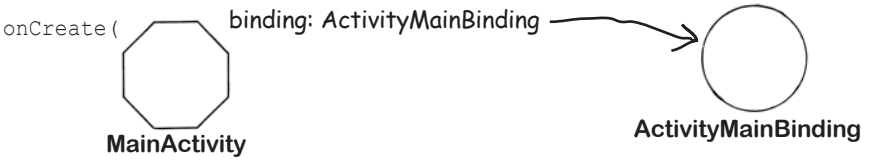
Что делает код

При выполнении приложения происходит следующее:

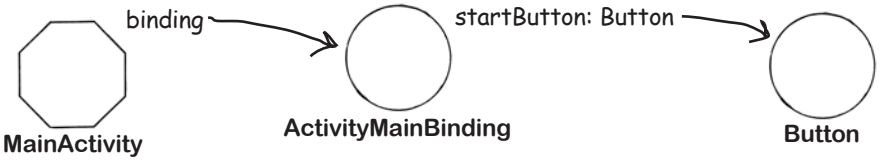
1. **При запуске приложения создается MainActivity.**
Активность включает свойство `ActivityMainBinding` с именем `binding`.



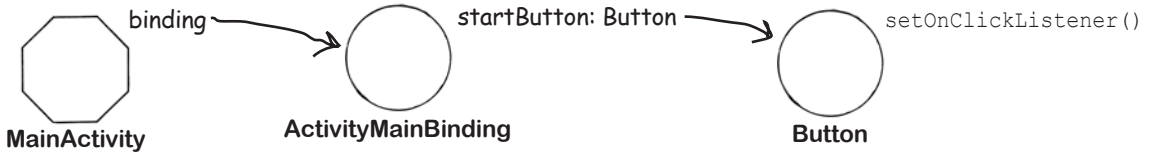
2. **При выполнении метода onCreate() активности MainActivity объект ActivityMainBinding присваивается свойству binding.**
Значение `binding` присваивается в `onCreate()`, потому что именно здесь `MainActivity` впервые получает доступ к приложениям.



3. ***Объект ActivityMainBinding включает свойство для каждого представления в макете с идентификатором.***
Например, файл макета *activity_main.xml* включает кнопку с идентификатором `start_button`, поэтому объект `ActivityMainBinding` включает свойство `Button` с именем `startButton`, открывающее доступ к этому представлению.



4. **MainActivity использует свойство binding для обращения к своим представлениям.**
Оно указывает, как кнопка Start должна реагировать на щелчки, например вызовом метода `setOnClickListener()` свойства `startButton`.

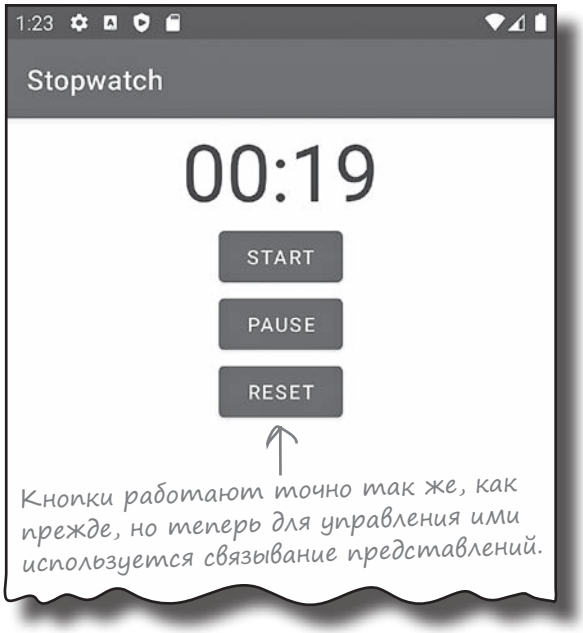


Проведем тест-драйв приложения.

Тест-драйв

Код активности
Код фрагмента

При запуске приложения все работает так же, как и прежде. Если щелкнуть на кнопке Start, секундомер запускается, а при щелчках на кнопках Pause и Reset он приостанавливается и сбрасывается.



Однако в новой версии `MainActivity` использует для взаимодействия с представлениями механизм связывания представлений (вместо вызовов `findViewById()`).

Часто Задаваемые Вопросы

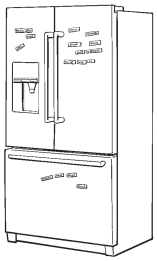
В: Почему связывание представлений не рассматривалось ранее в книге?

О: Мы хотели, чтобы перед знакомством со связыванием представлений у вас был некоторый опыт работы с активностями и фрагментами. Связывание представлений несколько усложняет отдельные части кода, и мы решили, что вам стоит поначалу получше изучить основы.

В: Можно ли запретить генерирование класса связывания для конкретного файла макета?

О: Да. Если вы знаете, что с некоторым макетом связывание представлений использоваться не будет, запретите генерировать для него класс связывания. Для этого включите в корневое представление элемент

```
tools:viewBindingIgnore="true"
```



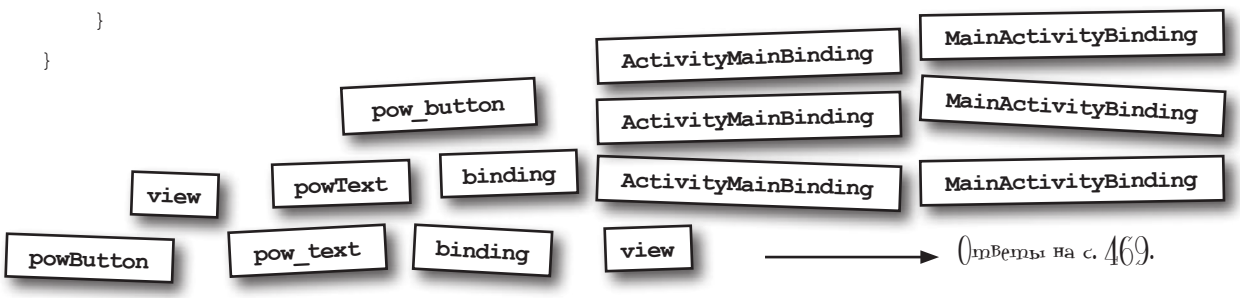
Развлечения с магнитами

Активность `MainActivity` использует файл *activity_main.xml* для определения макета. Макет включает кнопку `Button` (с идентификатором `pow_button`) и текстовое представление `TextView` (с идентификатором `pow_text`). По щелчку на кнопке в `TextView` должен выводиться текст «Pow!».

Код `MainActivity` был выложен из магнитов на холодильнике, но из-за неожиданной песчаной бури в кухне часть магнитов отвалилась. Сможете ли вы восстановить разрушенный код?

```
package com.hfad.myapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock

import com.hfad.myapp.databinding.  ..........

class MainActivity : AppCompatActivity() {

    private lateinit var binding:  ..........

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = ..........inflate(layoutInflater)
        val view = binding.root
        setContentView(view)


    ..........  ..........  .setOnClickListener {

        ..........  ..........  .text = "Pow!"
        }
    }
}
```

Использовать все магниты не обязательно.



Ответы на с. 469.

Код активности
Код фрагмента

Фрагменты тоже могут использовать связывание представлений, но код выглядит немного иначе

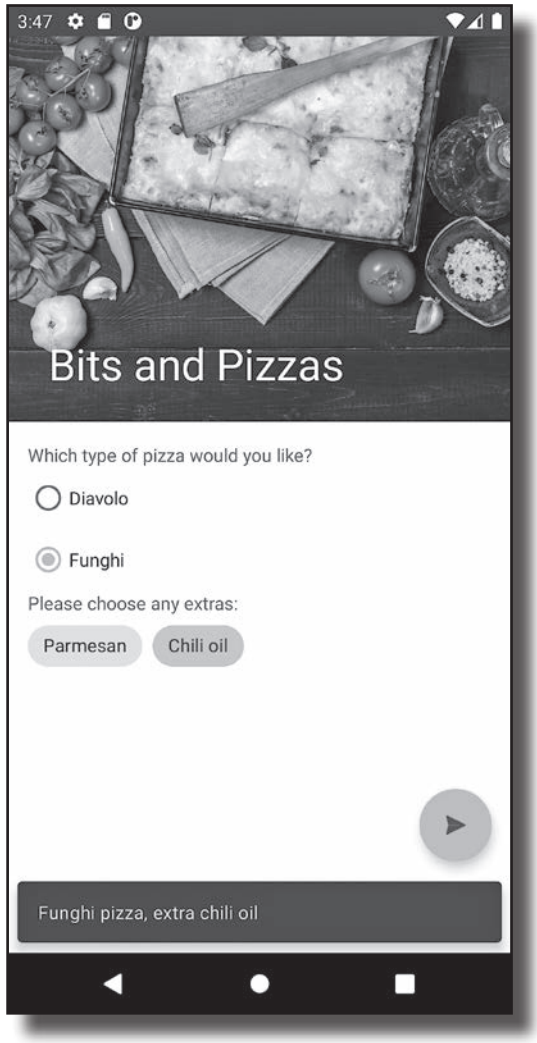
Итак, вы узнали, как реализовать связывание представлений в коде активности. Давайте посмотрим, как использовать его во фрагментах. Как было сказано ранее, код фрагментов несколько отличается от кода активностей.

Реализуем связывание представлений в приложении Bits and Pizzas

Чтобы понять, как работает связывание представлений для фрагментов, мы внесем изменения в приложение Bits and Pizzas, созданное в главе 9. Откройте проект этого приложения.

Как вы, вероятно, помните, приложение Bits and Pizzas использует представления Material, такие как сворачиваемые панели инструментов и FAB, для построения интерактивного пользовательского интерфейса. Пользовательский интерфейс определяется фрагментом приложения, который называется `FragmentOrder`.

Вспомните, как выглядит приложение Bits and Pizzas. В нем отображаются различные представления, которыми мы управляем из кода фрагмента. Все вызовы findViewById() можно заменить реализацией связывания представлений.



Код `FragmentOrder` включает множество вызовов `indViewById()`. Давайте посмотрим, как использовать связывание представлений для их замены.

Не забудьте!

Прежде чем продолжать, закройте приложение Stopwatch и откройте приложение Bits and Pizzas, созданное в главе 9.

Код активности
Код фрагмента

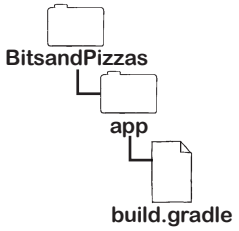
Включение связывания представлений для Bits and Pizzas

Как и прежде, чтобы использовать связывание представлений в проекте Bits and Pizzas, необходимо сначала разрешить его использование в файле *build.gradle* приложения.

Откройте файл *BitsandPizzas/app/build.gradle* и включите следующие строки в раздел `android`:

Добавьте эти строки в раздел android файла build.gradle приложения.

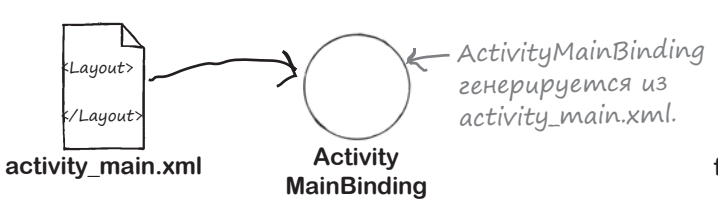
```
android {
    ...
    buildFeatures {
        viewBinding true
    }
}
```

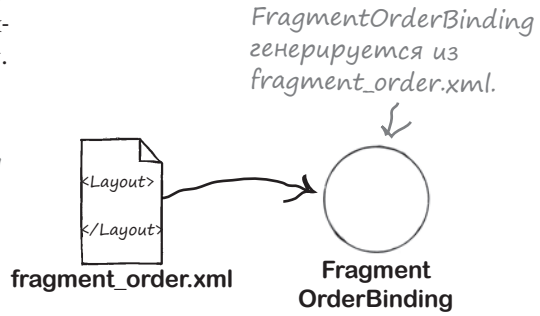


Затем выберите команду Sync Now, чтобы синхронизировать изменения с остальными частями проекта.

Классы связывания генерируются для каждого макета

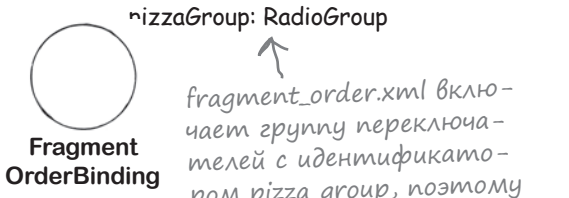
Как и в приложении Stopwatch, при включении связывания представлений в приложение Bits and Pizzas для каждого файла макета автоматически создается класс связывания. Приложение содержит два макета: *activity_main.xml* и *fragment_order.xml*, поэтому генерируются два класса связывания: `ActivityMainBinding` и `FragmentOrderBinding`.





Как и прежде, каждый класс связывания включает свойство для каждого представления с идентификатором в его макете. Например, макет *fragment_order.xml* включает группу переключателей с идентификатором `pizza_group`, поэтому класс связывания `FragmentOrderBinding` включает свойство с именем `pizzaGroup` и типом `RadioGroup`.

В приложении Bits and Pizzas с представлениями взаимодействует только код *OrderFragment.kt*, так что нам остается только обновить этот файл для использования связывания представлений.



Код активности
Код фрагмента

Код связывания представлений для фрагментов выглядит немного иначе

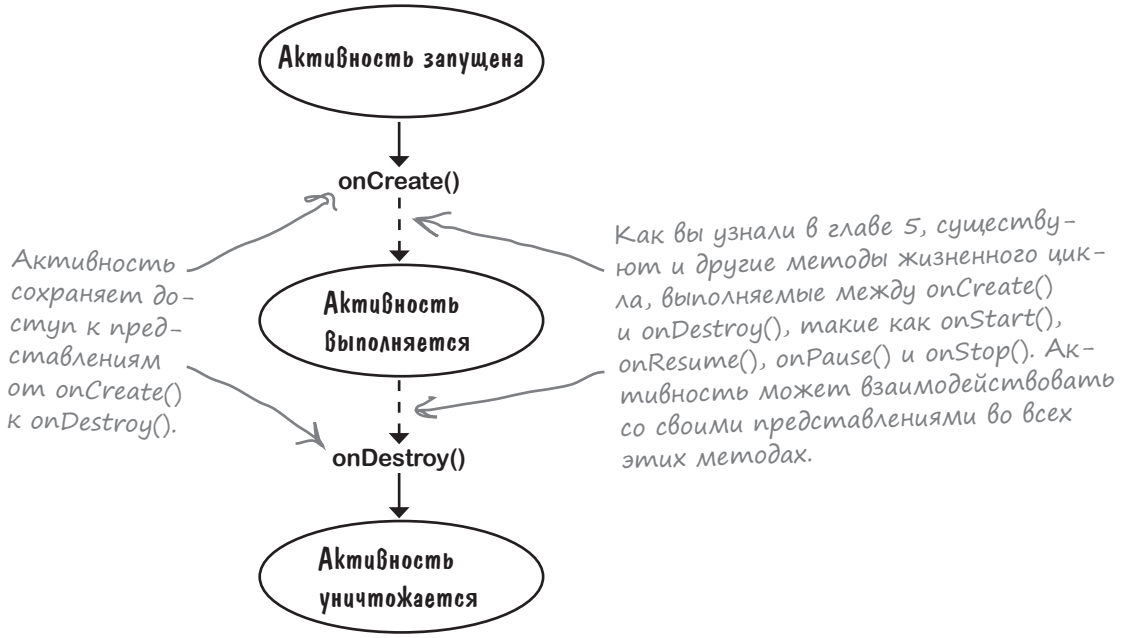
Как мы уже говорили, код связывания представлений, используемый во фрагментах, несколько отличается от кода, используемого в активностях. Но прежде чем показывать, как выглядит код связывания представлений во фрагментах, мы поглубже разберемся в теме и выясним, почему же код отличается.

Активности могут обращаться к представлениям из onCreate()

Как вам уже известно, активность впервые получает доступ к макету при выполнении ее метода `onCreate()`. Это первый метод жизненного цикла активности, и он используется для заполнения макета (или связывания его с новым представлением) и выполнения всей начальной настройки. Например, если кнопка должна реагировать на щелчки, активность назначает ей слушатель `OnClickListener` в своем методе `onCreate()`.

Активность сохраняет доступ к своему макету, пока не будет вызван ее метод `onDestroy()`. Это последний метод в жизненном цикле активности, и после того как он будет выполнен, активность уничтожается.

Так как активность сохраняет доступ к представлениям своего макета от `onCreate()` до `onDestroy()`, она может взаимодействовать с ними в любых своих методах:



Все это происходит с активностями, а с фрагментами дело обстоит немного иначе — давайте посмотрим, в чем именно.

Код активности
Код фрагмента

Фрагменты могут обращаться к представлениям от onCreateView() до onDestroyView()

Как вы знаете, фрагмент впервые получает доступ к своему макету при выполнении его метода `onCreateView()`. Он вызывается, когда активности потребуется доступ к макету фрагмента, поэтому он используется для заполнения макета и выполнения всей исходной настройки (например, назначения слушателей `OnClickListener`).

Тем не менее `onCreateView()` — не первый метод в жизненном цикле фрагмента. Существуют и другие методы, выполняемые до `onCreateView()` (например, `onCreate()`), и они не могут взаимодействовать с представлениями фрагмента.

После выполнения метода `onCreateView()` фрагмент сохраняет доступ к своим представлениям до того, как завершится выполнение его метода `onDestroyView()`. Метод вызывается, когда макет фрагмента становится ненужным для активности, например потому что активности нужно перейти к другому фрагменту или он уничтожается.

Однако `onDestroyView()` — не последний метод в жизненном цикле фрагмента. Существуют и другие методы, выполняемые после `onDestroyView()` (например, `onDestroy()`), и они тоже не могут взаимодействовать с представлениями фрагмента:

Активность может взаимодействовать со своими представлениями от onCreate() до onDestroy().

Фрагмент может взаимодействовать со своими представлениями только от onCreateView() до onDestroyView().

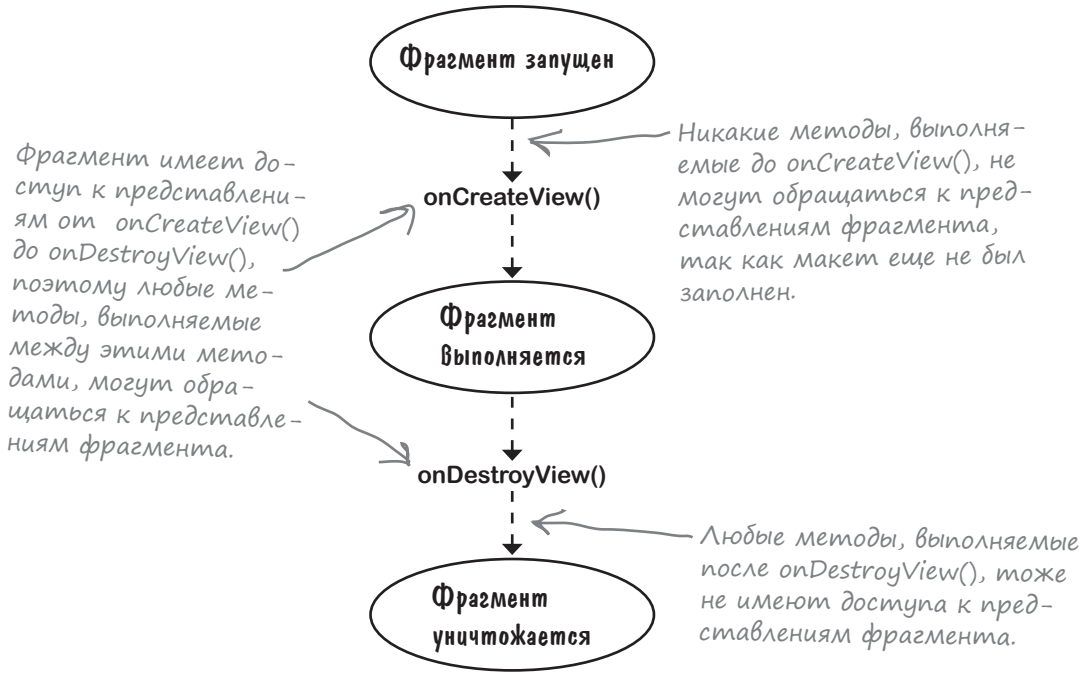
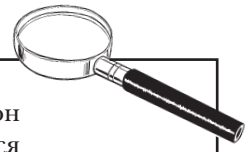


Схема жизненного цикла фрагмента приведена на следующей странице. Затем вы увидите, как выглядит код связывания представлений для фрагмента.

Методы жизненного цикла фрагмента под увеличительным стеклом



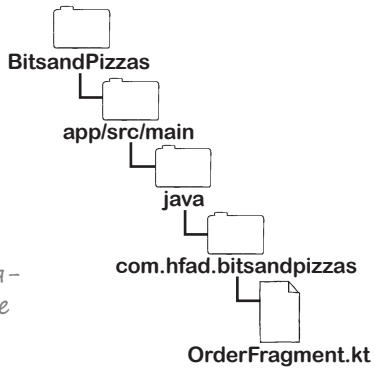
Жизненный цикл фрагмента очень похож на жизненный цикл активности, но он содержит ряд дополнительных фаз. На следующей схеме показано, когда вызываются методы жизненного цикла главного фрагмента, для чего они используются и какое место они занимают в состоянии активности, в которой отображается фрагмент:

Состояние активности	Метод жизненного цикла фрагмента	
Создается	onCreate()	**onCreate(Bundle)** Вызывается при создании фрагмента. Может использоваться для любой исходной настройки, в которой не задействованы представления.
	onCreateView()	**onCreateView(LayoutInflater, ViewGroup, Bundle)** Здесь фрагмент впервые получает доступ к своим представлениям.
Запускается	onStart()	**onStart()** Вызывается, когда фрагмент готовится стать видимым.
Продолжает выполнение	onResume()	**onResume()** Вызывается, когда фрагмент начинает активно выполняться.
Приостанавливается	onPause()	**onPause()** Вызывается, когда фрагмент перестает активно выполняться.
Останавливается	onStop()	**onStop()** Вызывается, когда фрагмент перестает быть видимым.
Уничтожается	onDestroyView()	**onDestroyView()** Вызывается, когда иерархия представлений фрагмента готовится к уничтожению, чтобы фрагмент мог освободить ресурсы.
	onDestroy()	**onDestroy()** Вызывается перед уничтожением фрагмента. В этой точке могут освобождаться любые дополнительные ресурсы.

Как выглядит код связывания представлений для фрагмента

Теперь вы готовы узнать, как выглядит код связывания представлений для фрагментов. Как и в случае с активностями, связывание представлений во фрагментах реализуется с использованием свойства связывания, но способ решения этой задачи немного другой.

Код выглядит так:




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.bitsandpizzas.databinding.FragmentOrderBinding

class OrderFragment : Fragment() {
    private var _binding: FragmentOrderBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_order, container, false)
        _binding = FragmentOrderBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Импортируем класс связывания, в данном случае FragmentOrderBinding.

Добавляем резервное свойство с именем _binding, которому присваивается null.

Для обращения к _binding используется свойство binding. Если значение _binding отлично от null, возвращается _binding. Если же _binding содержит null, при попытке использования выдается исключение null-указателя.

Удалите эту строку.

Задается значение свойства _binding.

Получает объект FragmentOrderBinding, связанный с макетом фрагмента.

В onDestroyView() свойству _binding присваивается null.

Как видите, в этом коде определяются два дополнительных свойства: `binding` и `_binding`. Давайте повнимательнее разберемся в том, что делают эти два свойства.

Расслабьтесь

Код связывания представлений для фрагментов остается почти неизменным для всех фрагментов.

После включения этих изменений вы сможете заменить все вызовы метода `findViewById()`, как это было сделано для активностей.

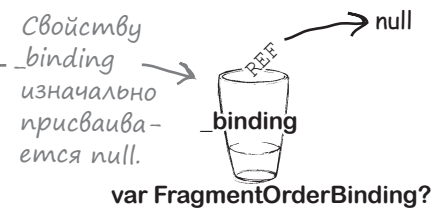
Код активности
Код фрагмента

_binding хранит ссылку на объект связывания...

Как вы уже видели, код связывания представлений для фрагментов определяет свойство _binding:

```
private var _binding: FragmentOrderBinding? = null
```

Свойству _binding изначально присваивается null.



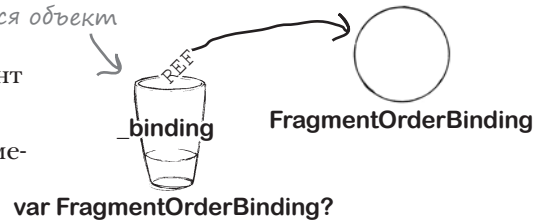
Оно имеет тип FragmentOrderBinding? и инициализируется значением null.

Свойству _binding присваивается экземпляр FragmentOrderBinding в методе onCreateView() фрагмента, для чего используется следующий код:

```
override fun onCreateView(...): View? {
    _binding = FragmentOrderBinding.inflate(inflater, container, false)
    ...
}
```

Когда фрагмент получает доступ к своему макету, _binding присваивается объект FragmentOrderBinding.

Оно задается в этом методе, потому что именно здесь фрагмент впервые получает доступ к своим представлениям.



Свойству _binding снова присваивается значение null в методе onDestroyView() фрагмента:

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

Фрагмент не может обращаться к своему макету после вызова onDestroyView(), поэтому _binding снова присваивается null.

Это происходит из-за того, что фрагмент не может обращаться к своим представлениям после вызова метода onDestroyView().

...а свойство binding предоставляет к нему доступ

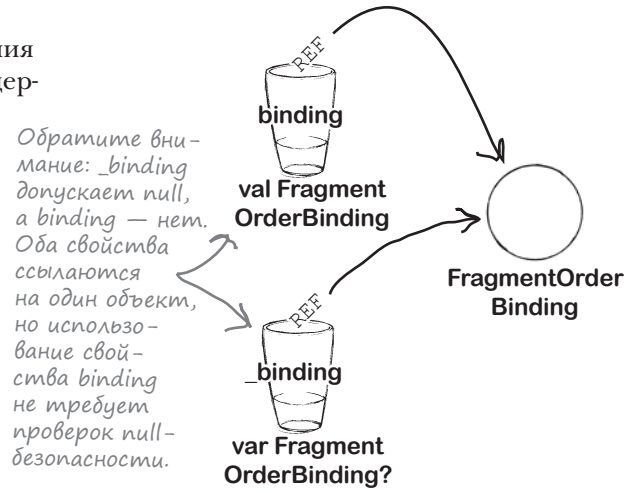
Свойство binding использует метод чтения для получения версии _binding, отличной от null. Если _binding содержит null, выдается исключение null-указателя:

```
private val binding get() = _binding!!
```

Это означает, что свойство binding можно использовать для взаимодействия с представлениями фрагмента без множества хлопотных проверок null-безопасности. Например, чтобы кнопка FAB фрагмента реагировала на щелчки, можно использовать простую команду:

```
binding.fab.setOnClickListener {
    //Код, который делает что-то полезное
}
```

Обратите внимание: _binding допускает null, а binding — нет. Оба свойства ссылаются на один объект, но использование свойства binding не требует проверок null-безопасности.



Код активности
Код фрагмента

Кажется, я поняла. Код связывания представлений для фрагмента отличается от кода связывания представлений для активности, потому что фрагмент не может получить доступ к своим представлениям, пока не будет вызван его метод onCreateView().

Активности и фрагменты получают доступ к своим представлениям в разных точках своих жизненных циклов.

Активность может взаимодействовать со своими представлениями, начиная с момента их создания и вызова их метода `onCreate()`. Представления остаются доступными до момента уничтожения активности, что происходит в конце жизненного цикла.

Однако *фрагмент* может взаимодействовать со своими представлениями только с момента вызова его метода `onCreateView()`. Это означает, что присваивание объекта связывания свойству `_binding` может выполняться *только* в этом методе.

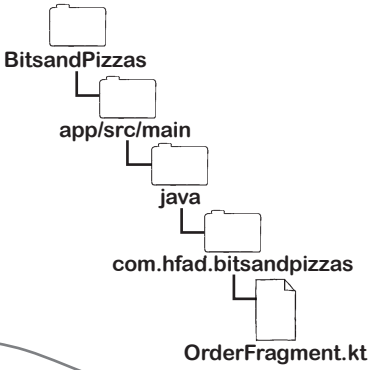
Фрагмент сохраняет доступ к своим представлениям до того момента, когда будет вызван его метод `onDestroyView()`. В этот момент макет фрагмента теряется, поэтому `_binding` необходимо присвоить null. Тем самым предотвращается риск того, что фрагмент попытается использовать объект связывания представлений, когда взаимодействие с представлениями уже невозможно.

Теперь вы знаете все, что необходимо знать для использования связывания представлений с фрагментами. Давайте обновим код `OrderFragment` в приложении Bits and Pizzas.

Код активности
Код фрагмента

Полный код OrderFragment.kt

Ниже приведен код для использования связывания представлений в `OrderFragment`; обновите свой код *OrderFragment.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.bitsandpizzas

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.app.AppCompatActivity
~~import com.google.android.material.appbar.MaterialToolbar~~
~~import android.widget.RadioGroup~~
~~import com.google.android.material.chip.Chip~~
~~import com.google.android.material.floatingactionbutton.FloatingActionButton~~
import android.widget.Toast
import com.google.android.material.snackbar.Snackbar
import com.hfad.bitsandpizzas.databinding.FragmentOrderBinding

class OrderFragment : Fragment() {
    private var _binding: FragmentOrderBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ~~val view = inflater.inflate(R.layout.fragment_order, container, false)~~
        _binding = FragmentOrderBinding.inflate(inflater, container, false)
        val view = binding.root
        ~~val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)~~
        (activity as AppCompatActivity).setSupportActionBar(binding.toolbar)
```

Эти команды импортирования не нужны, их можно удалить.

Импортирование класса связывания.

Объявление свойств _binding и binding.

Удалите эту строку.

Присваивание значения свойству _binding.

Используем связывание представлений вместо findViewById().

Продолжение на следующей странице.

Код активности
Код фрагмента

Полный код OrderFragment.kt (продолжение)

```
        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        binding.fab.setOnClickListener {
            val pizzaGroup = view.findViewById<RadioGroup>(R.id.pizza_group)
            val pizzaType = binding.pizzaGroup.checkedRadioButtonId
            if (pizzaType == -1) {
                val text = "You need to choose a pizza type"
                Toast.makeText(activity, text, Toast.LENGTH_LONG).show()
            } else {
                var text = (when (pizzaType) {
                    R.id.radio_diavolo -> "Diavolo pizza"
                    else -> "Funghi pizza"
                })
                val parmesan = view.findViewById<Chip>(R.id.parmesan)
                text += if (binding.parmesan.isChecked) ", extra parmesan" else ""
                val chiliOil = view.findViewById<Chip>(R.id.chili_oil)
                text += if (binding.chiliOil.isChecked) ", extra chili oil" else ""
                Snackbar.make(binding.fab, text, Snackbar.LENGTH_LONG).show()
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Вызов findViewById() заменяется связыванием представлений.

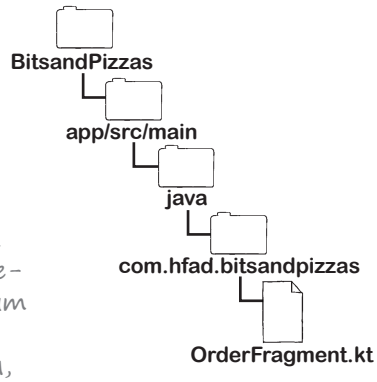
Здесь тоже используется связывание представлений.

Удалите эту строку.

Эти вызовы findViewById() заменяются связыванием представлений.

Обращение к FAB с использованием связывания.

Метод жизненного цикла onDestroyView() переопределяется: когда фрагмент не может обращаться к своим представлениям, _binding присваивается null.



И это весь код, необходимый для использования связывания представлений во фрагменте `OrderFragment`. Проведем тест-драйв приложения и убедимся в том, что оно по-прежнему нормально работает.

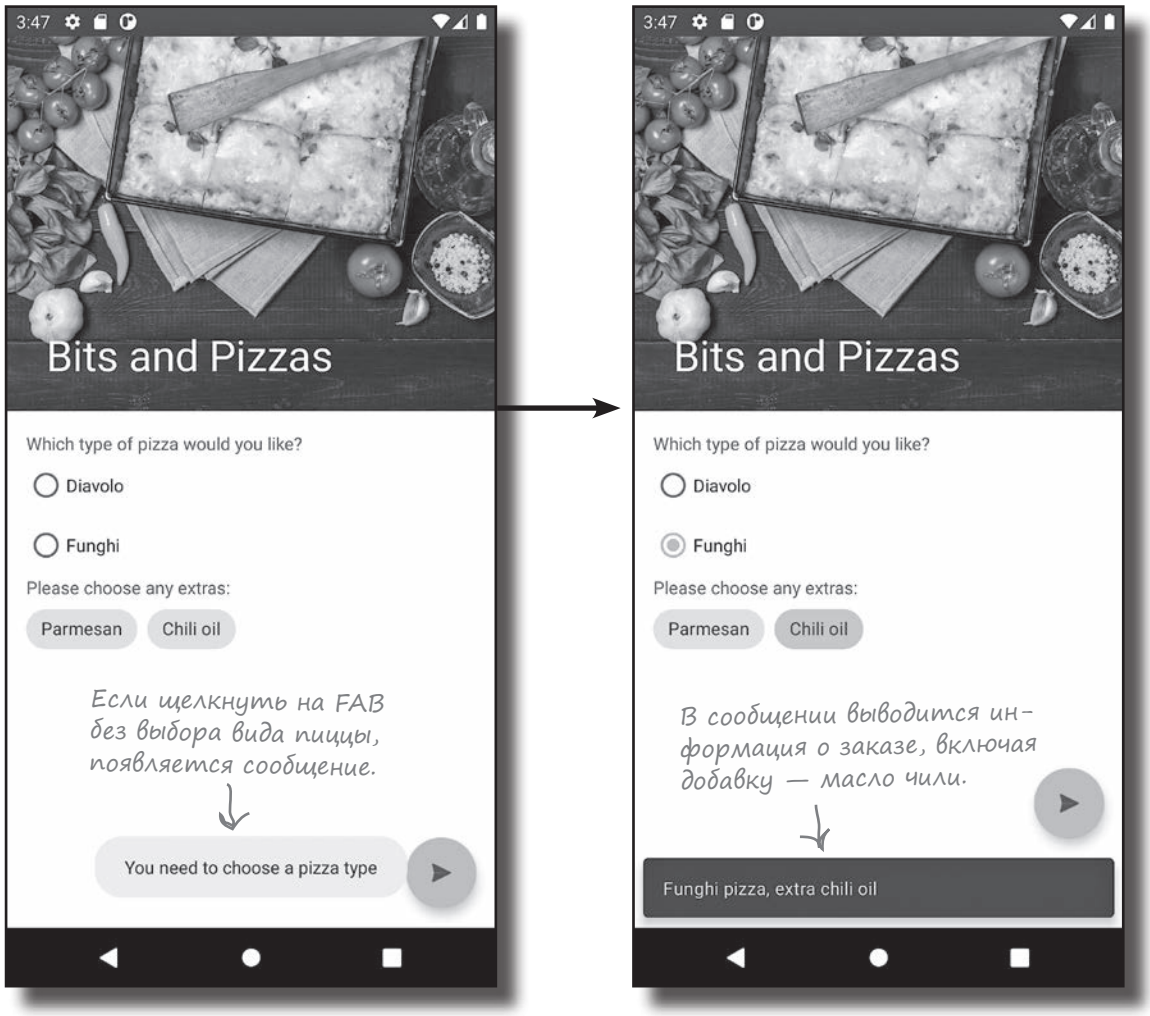
Тест-драйв

Код активности
Код фрагмента

Запущенное приложение работает точно так же, как прежде.

Если щелкнуть на кнопке FAB без выбора вида пиццы, появляется сообщение с предложением выбрать пиццу.

Если выбрать вид пиццы и снова щелкнуть на FAB, в нижней части экрана появляется сообщение Snackbar с подробной информацией о заказе.



Поздравляем! Теперь вы знаете, как реализовать связывание представлений в коде активностей и фрагментов. Мы продолжим использовать связывание представлений в оставшейся части книги.

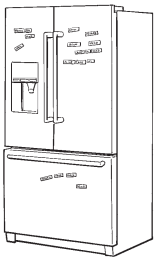
Упражнение

Следующий код фрагмента заставляет кнопку FAB (с идентификатором `fab`) реагировать на щелчки. По щелчку на кнопке на экране устройства появляется сообщение Snackbar. Оно включает действие, которое отображает сообщение Toast.

Удастся ли вам обновить фрагмент, чтобы в нем использовалось связывание представлений?

```
... ← Объявление пакета и команды импортирования опущены...
class MyFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_my, container, false)
        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        fab.setOnClickListener {
            Snackbar.make(fab, "Your order has been updated", Snackbar.LENGTH_SHORT)
                    .setAction("Undo") {
                        Toast.makeText(activity, "Undone!", Toast.LENGTH_SHORT)
                        .show()
                    }
                    .show()
        }
        return view
    }
}
```

Ответ на с. 470.



Развлечения с магнитами. Решение

Активность `MainActivity` использует файл *activity_main.xml* для определения макета. Макет включает кнопку `Button` (с идентификатором `pow_button`) и текстовое представление `TextView` (с идентификатором `pow_text`). По щелчку на кнопке в `TextView` должен выводиться текст «Pow!».

Код `MainActivity` был выложен из магнитов на холодильнике, но из-за неожиданной песчаной бури в кухне часть магнитов отвалилась. Сможете ли вы восстановить разрушенный код?

```
package com.hfad.myapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock

import com.hfad.myapp.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        binding.powButton.setOnClickListener {
            binding.powText.text = "Pow!"
        }
    }
}
```

Импортируем класс ActivityMainBinding.

Свойство binding должно иметь тип ActivityMainBinding.

Присваивание значения свойству binding.

Связывание представлений используется для назначения слушателя OnClickListener для кнопки.

Связывание представлений используется для назначения содержимого текстового представления.

Эти магниты не понадобились.

MainActivityBinding

view

MainActivityBinding

pow_text

view

pow_button

MainActivityBinding

Упражнение Решение

Следующий код фрагмента заставляет кнопку FAB (с идентификатором `fab`) реагировать на щелчки. По щелчку на кнопке на экране устройства появляется сообщение Snackbar. Оно включает действие, которое отображает сообщение Toast.

Удастся ли вам обновить фрагмент, чтобы в нем использовалось связывание представлений?

```
...
class MyFragment : Fragment() {
    private var _binding: FragmentMyBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = inflater.inflate(R.layout.fragment_my, container, false)
        _binding = FragmentMyBinding.inflate(inflater, container, false)
        val view = binding.root
        val fab = view.findViewById<FloatingActionButton>(R.id.fab)
        binding.fab.setOnClickListener {
            Snackbar.make(binding.fab, "Your order has been updated",
                          Snackbar.LENGTH_SHORT)
                .setAction("Undo") {
                    Toast.makeText(activity, "Undone!", Toast.LENGTH_SHORT)
                    .show()
                }
                .show()
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Объявляем _binding, присваиваем null и добавляем метод чтения binding.

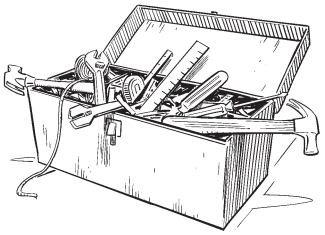
Эта строка удаляется, так как мы хотим использовать связывание представлений.

Присваиваем _binding.

Для обращения к FAB используется связывание представлений вместо findViewById().

Переопределяем onDestroyView().

Ваш инструментарий Android



Глава 10 осталась позади, а ваш инструментарий пополнился связыванием представлений.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

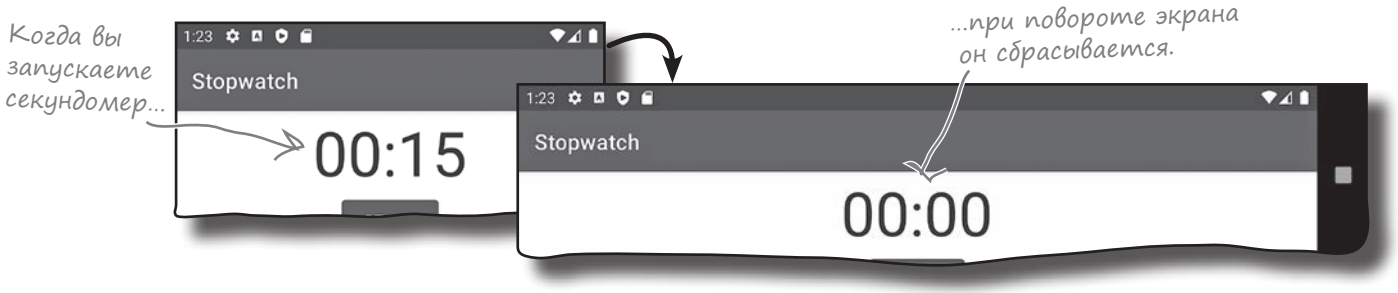
- Связывание представлений — механизм взаимодействия с представлениями, безопасный по отношению к типам. Он эффективнее `findViewById()` и с меньшей вероятностью приведет к ошибкам на стадии выполнения.
- Связывание представлений может использоваться в коде активностей и фрагментов.
- Связывание представлений включается в файле *build.gradle* приложения.
- Связывание представлений генерирует класс связывания для каждого файла макета приложения.
- Каждый класс связывания включает свойство для каждого представления в макете, обладающего идентификатором.
- Используя связывание представлений с активностью, вы задаете значение его свойства binding в методе `onCreate()` активности.
- Используя связывание представлений с фрагментом, вы задаете значение его свойства binding в методе `onCreateView()` фрагмента и возвращаете ему значение null в `onDestroyView()`.
- Фрагменты содержат методы жизненного цикла, связанные с состоянием активности.

Поведение модели

С ростом сложности приложений фрагментам приходится жонглировать все большим количеством объектов. И если не проявить осторожность, это может привести к **разбуханию** кода, который пытается делать все сразу. Бизнес-логика, навигация, управление пользовательским интерфейсом, обработка изменений в конфигурации... что ни возьми, оно здесь. В этой главе вы узнаете, как действовать в подобных ситуациях с использованием **моделей представлений.** Вы узнаете, **как они упрощают код активности и фрагментов** и как они **справляются с изменениями конфигурации**, поддерживая целостное состояние приложения. Наконец, мы покажем, как построить **фабрику моделей представлений** и когда может возникнуть такая необходимость.

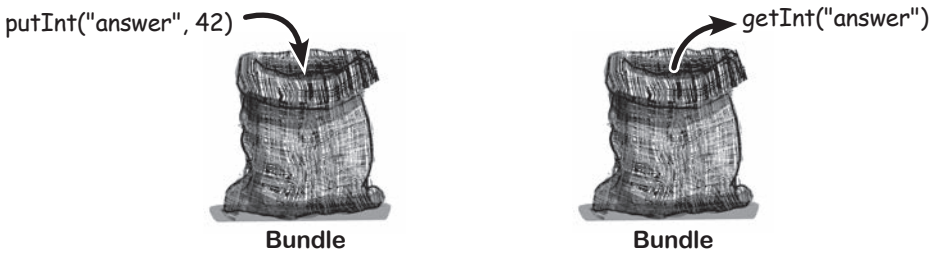
Снова об изменениях конфигурации

Как вы узнали в главе 5, при повороте экрана во время работы приложения могут происходить Всякие Неприятности. Изменение ориентации экрана является изменением конфигурации, которое заставляет Android уничтожить и заново создать текущую активность. В результате представления и свойства могут потерять свое состояние и вернуться к исходному значению:



В главе 5 вы научились решать эту проблему с использованием метода `onSaveInstanceState` активности. Этот метод вызывается перед уничтожением активности, а для сохранения всех значений, которые могут быть потеряны, в нем обычно используется объект `Bundle`. При воссоздании активности сохраненные значения могут использоваться для восстановления состояния представлений и свойств активности.

У фрагментов тоже есть метод onSaveInstanceState. И хотя мы продемонстрировали использование этого подхода только с активностями, он также работает и с фрагментами.



Сохранение состояния в объектах `Bundle` неплохо работает в относительно простых приложениях, но в более сложных ситуациях такое решение не идеально. Дело в том, что объекты `Bundles` предназначены для небольших объемов данных и работают с ограниченным набором типов.

Существуют и другие проблемы

Существует и другая проблема: код активностей и фрагментов быстро разрастается. Коду приходится управлять навигацией, обновлять интерфейс, сохранять состояние, а иногда он также включает более общую бизнес-логику для управления поведением приложения. Решение всех этих задач в одном месте удлиняет код, а также усложняет его чтение и сопровождение.

В этой главе вы научитесь решать все перечисленные проблемы с использованием **модели представления**.

Знакомство с моделями представлений

Модель представления — отдельный класс, который существует параллельно с кодом активности или фрагмента. Он отвечает за все данные, которые должны отображаться на экране, а также может содержать любую бизнес-логику. Каждый раз, когда фрагменту требуется обновить свой макет, он запрашивает у модели представления новейшие значения, которые ему нужно отобразить, а если ему потребуется доступ к бизнес-логике, он вызывает методы, содержащиеся в модели представления.

Модели представлений также являются частью Android Jetpack.

Для чего нужны модели представлений?

Существует пара причин для использования модели представления.

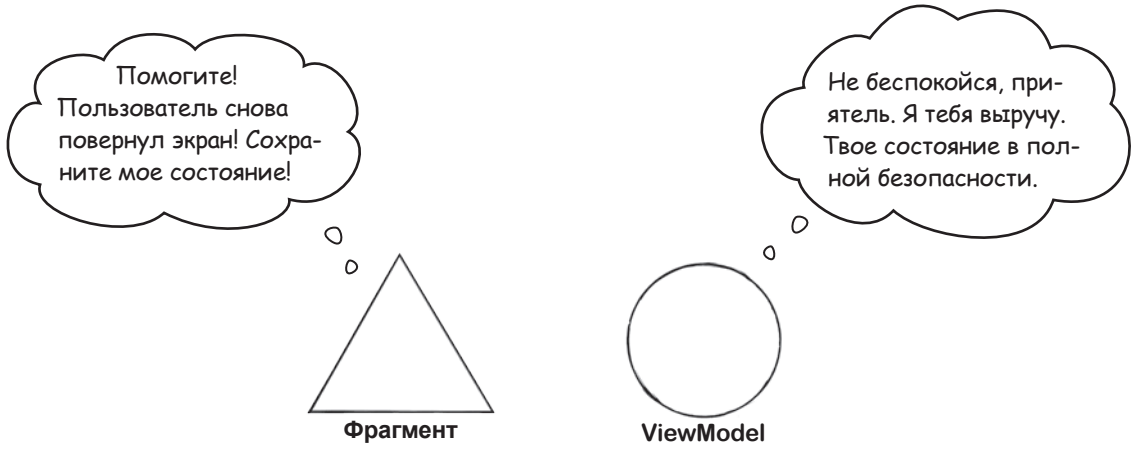
Использование модели представления упрощает код активности или фрагмента. Во фрагмент уже не нужно включать код, относящийся к бизнес-логике приложения, так как все хранится в отдельном классе. Вместо этого можно сосредоточиться на таких аспектах, как обновление экрана или навигация.

Другая причина заключается в том, что **модель представления может пережить изменения конфигурации.** Она не уничтожается, когда пользователь поворачивает экран устройства, так что состояние отдельных переменных не будет потеряно. Это позволит вам восстановить состояние приложения без сохранения значений в `Bundle`.

В: Вы сказали, что объекты `Bundle` рассчитаны на хранение ограниченного набора типов данных. Означает ли это, что сохранить другие типы не удастся?

О: Строго говоря, вы можете добавить дополнительные типы в `Bundle` при помощи интерфейса `Parcelable`. Мы не будем подробно рассматривать этот подход, так как, на наш взгляд, лучше использовать модель представления.

Впрочем, если эта тема вас заинтересует, дополнительную информацию о `Parcelable` можно найти в документации Android.



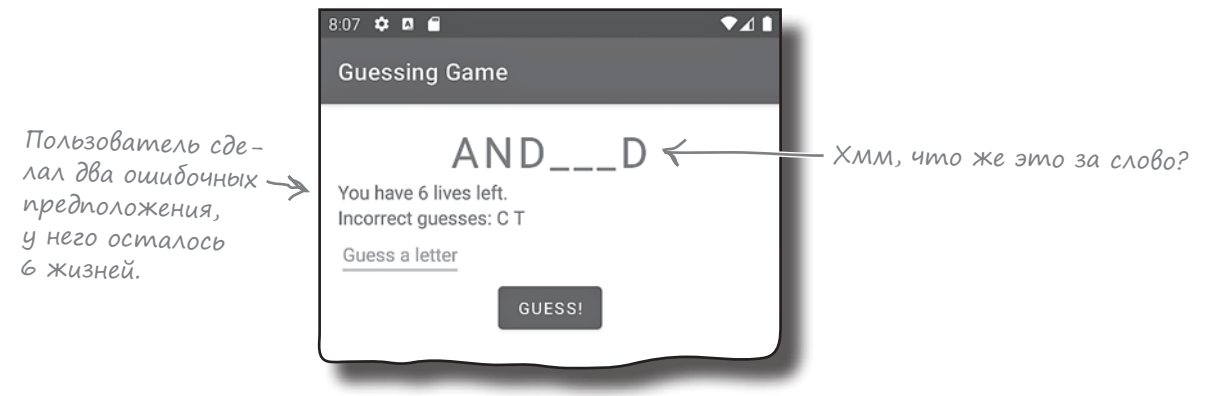
Чтобы вы лучше поняли, как пользоваться моделями представлений, мы построим игру с угадыванием слов. Но прежде чем переходить к написанию кода, разберемся, как будет работать эта игра.

Как будет работать игра

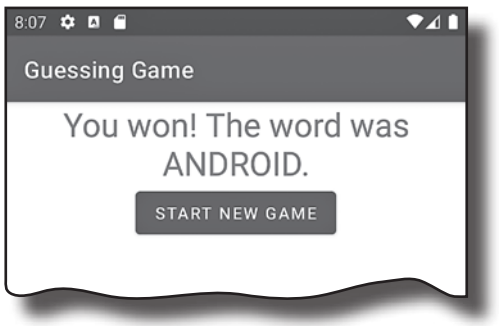
В игре Guessing Game пользователь пытается угадать загаданное слово.

При запуске приложение выбирает случайное слово из массива и выводит пробел для каждой буквы слова:

Пользователь вводит букву, которая, по его предположению, может входить в загаданное слово. Если предположение было правильным, игра выводит букву в той позиции, в какой она встречается в загаданном слове. Если предположение было ошибочным, игра выводит сообщение, а игрок теряет одну жизнь.



Пользователь продолжает делать предположения, пока не угадает все буквы или не потеряет все жизни. Когда это произойдет, появляется новый экран, на котором выводится загаданное слово, а также сообщается результат игры. На этом экране пользователю также предлагается начать новую игру.

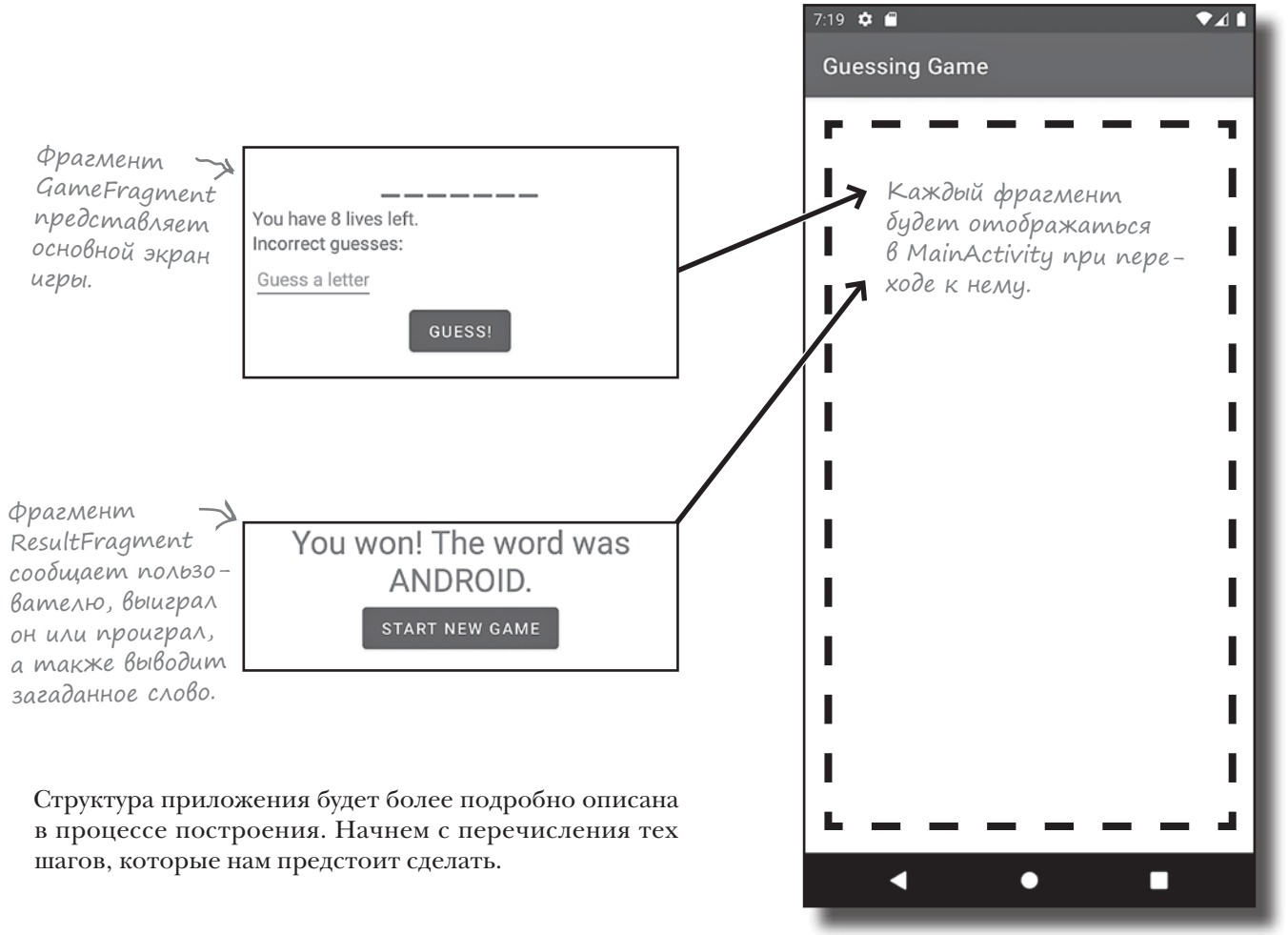


Структура приложения

Приложение будет содержать одну активность (с именем `MainActivity`), которая используется для отображения двух фрагментов игры: `GameFragment` и `ResultFragment`.

`GameFragment` представляет главный экран приложения. На нем выводятся пустые ячейки, обозначающие буквы загаданного слова, пользователю предоставляется возможность ввести букву, а также отображается количество ошибочных предположений и оставшихся жизней.

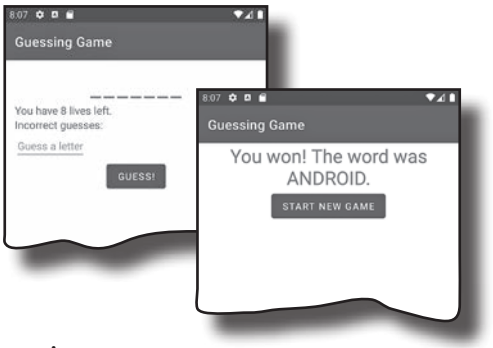
При завершении игры `GameFragment` использует компонент Navigation для перехода к `ResultFragment`, при этом передается строка с результатом. `ResultFragment` отображает строку и кнопку, при помощи которой пользователь может начать новую игру.



Структура приложения будет более подробно описана в процессе построения. Начнем с перечисления тех шагов, которые нам предстоит сделать.

Что мы собираемся сделать

Ниже перечислены основные этапы построения приложения.

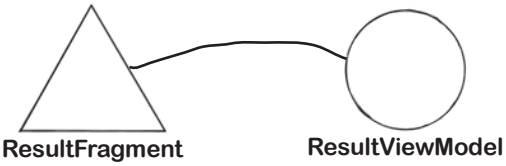


1. **Написание базовой игры.**
Мы создадим фрагменты `GameFragment` и `ResultFragment`, напишем игровую логику и воспользуемся компонентом Navigation для переходов между двумя фрагментами.

Построение базовой игры займет несколько страниц. Можете считать это повторением того, что вам уже известно.

2. **Добавление модели представления для GameFragment.**
Мы создадим модель представления `GameViewModel`для хранения игровой логики и данных `GameFragment`. Такой подход упрощает код `GameFragment` и гарантирует, что игра переживет изменения конфигурации.

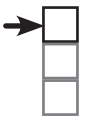
3. **Добавление модели представления для ResultFragment.**
Мы добавим вторую модель представления `ResultViewModel`, которая должна использоваться `ResultFragment`. В этой модели представления будет храниться результат игры, только что завершенной пользователем.



Создание проекта Guessing Game

Для приложения Guessing Game нужно создать новый проект. Создайте его по той же схеме, которая использовалась в предыдущих главах. Выберите вариант Empty Activity, введите имя приложения «Guessing Game» и имя пакета «com.hfad.guessinggame» и подтвердите каталог для сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin с минимальным уровнем SDK API 21, чтобы приложение работало на большинстве устройств Android.

Приложение Guessing Game будет использовать механизм связывания представлений для обращения к своим представлениям и компонент Navigation для перехода между своими фрагментами. Обновим файлы *build.gradle* проекта и приложения и включим в них эти библиотеки.

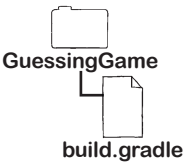


Базовая игра
GameFragment
ResultFragment

Обновление файла build.gradle проекта...

В файл *build.gradle* проекта будет добавлена новая переменная, определяющая используемую версию компонента Navigation, а также путь к классам Safe Args.

Откройте файл *GuessingGame/build.gradle* и добавьте следующие строки (выделенные жирным шрифтом) в указанные разделы:




```
buildscript {
    ext.nav_version = "2.3.5"
    ...
    dependencies {
        classpath "androidx.navigation:navigation-safe-args-gradle-plug-in:$nav_version"
        ...
    }
}
```

Номер версии для удобства сохраняется в nav_version.

Добавляем путь к классам Safe Args.

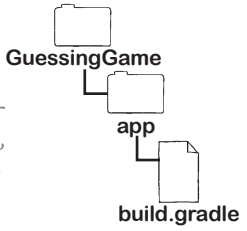
...файл build.gradle приложения тоже нужно обновить

В файле *build.gradle* приложения необходимо включить связывание представлений, зависимость для библиотеки компонента Navigation, применить плагин Safe Args для передачи аргументов фрагментам.

Откройте файл *GuessingGame/app/build.gradle* и добавьте следующие строки (выделенные жирным шрифтом) в соответствующие разделы:

```
plug-ins {
    ...
    id 'androidx.navigation.safeargs.kotlin'
}

android {
    ...
    buildFeatures {
        viewBinding true
    }
}

dependencies {
    ...
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    ...
}
```

Применяет плагин.



Включает связывание представлений, чтобы нам не приходилось избегать findViewById().

Добавляет зависимость для компонента Navigation.

После внесения изменений щелкните на ссылке Sync Now, чтобы синхронизировать изменения с остальными частями проекта.

Следующим шагом станет создание фрагментов приложения.

В приложении Guessing Game используются два фрагмента

Приложению Guessing Game необходимы два фрагмента: `GameFragment` и `ResultFragment`. `GameFragment` представляет главный экран приложения, а `ResultFragment` будет использоваться для отображения результата:

Сделаем следующий шаг и добавим два фрагмента в проект.

Создание GameFragment...

Сначала добавим `GameFragment` в проект.

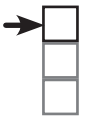
Выделите пакет *com.hfad.guessinggame* в папке *app/src/main/java*, после чего выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «GameFragment» и имя макета «fragment_game» и убедитесь в том, что выбран язык Kotlin.

Пока не беспокойтесь о коде этих фрагментов. Мы обновим его после того, как фрагменты будут добавлены в граф навигации.

...и создание ResultFragment

Затем добавьте фрагмент `ResultFragment` — выделите пакет *com.hfad.guessinggame* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «ResultFragment» и имя макета «fragment_result» и убедитесь в том, что выбран язык Kotlin.

Код этих двух фрагментов будет обновлен через несколько страниц. Но сначала мы добавим в проект граф навигации, который сообщит приложению, как должны происходить переходы между двумя фрагментами.



Базовая игра
GameFragment
ResultFragment

Как должна работать навигация

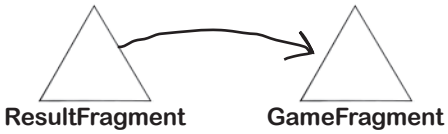
Как вы уже знаете, граф навигации сообщает Android информацию о возможных целях проекта и о переходах между ними.

В приложении Guessing Game навигация должна работать так:

1. **При запуске приложения отображается GameFragment.**

2. **При выигрыше или проигрыше GameFragment переходит к фрагменту ResultFragment, передавая ему результат.**

3. **Когда пользователь щелкает на кнопке New Game, приложение переходит к GameFragment.**



Чтобы реализовать эту схему, мы добавим `GameFragment` и `ResultFragment` в новый граф навигации (который вскоре будет создан) и укажем, что каждый фрагмент может переходить к другому.

Также следует указать, что `GameFragment` будет передавать строковый аргумент фрагменту `ResultFragment`. Аргумент должен содержать сообщение, указывающее, выиграл или проиграл пользователь только что сыгранную игру. Фрагмент `ResultFragment` будет выводить сообщение при переходе к нему.

Создание графа навигации

Чтобы создать граф навигации, выделите папку *GuessingGame/app/src/main/res* на панели проекта, а затем выберите команду File→New→Android Resource File. Введите имя файла «nav_graph», выберите тип ресурса «Navigation» и щелкните на кнопке OK. В результате будет создан граф навигации с именем *nav_graph.xml*.

Граф навигации необходимо обновить и включить в него фрагменты `GameFragment` и `ResultFragment` вместе с действиями навигации, приводящими к ним. Мы сделаем это на следующем шаге.

Граф навигации содержит подробную информацию обо всех возможных целях приложения и о том, как перейти к ним.

Базовая игра
GameFragment
ResultFragment

Обновление графа навигации

Чтобы обновить граф навигации, откройте файл *nav_graph.xml* (если он не был открыт ранее), переключитесь в режим Code и обновите код (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/gameFragment">
    <fragment
        android:id="@+id/gameFragment"
        android:name="com.hfad.guessinggame.GameFragment"
        android:label="fragment_game"
        tools:layout="@layout/fragment_game" >
        <action
            android:id="@+id/action_gameFragment_to_resultFragment"
            app:destination="@id/resultFragment"
            app:popUpTo="@id/gameFragment"
            app:popUpToInclusive="true" />
    </fragment>
    <fragment
        android:id="@+id/resultFragment"
        android:name="com.hfad.guessinggame.ResultFragment"
        android:label="fragment_result"
        tools:layout="@layout/fragment_result" >
        <argument
            android:name="result"
            app:argType="string" />
        <action
            android:id="@+id/action_resultFragment_to_gameFragment"
            app:destination="@id/gameFragment" />
    </fragment>
</navigation>
```

Проследите за тем, чтобы имя файла включало это пространство имен.

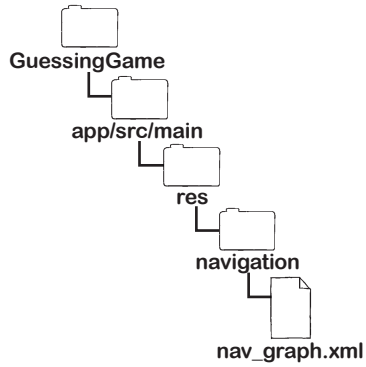
Игра начинается с GameFragment.

Дейтсвие позволяет GameFragment перейти к ResultFragment.

Это действие извлекает фрагменты из стека возврата до GameFragment включительно.

ResultFragment получает строковый аргумент.

Это действие позволяет ResultFragment перейти к GameFragment.

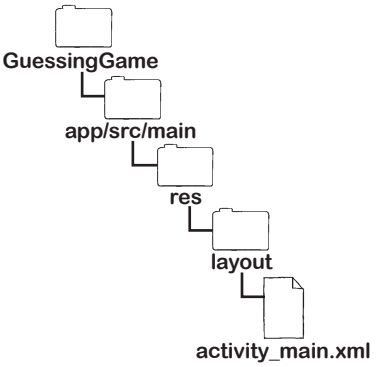


После обновления графа навигации свяжем его с активностью `MainActivity`, чтобы она отображала каждый фрагмент при переходе к нему.

Вывод текущего фрагмента в макете MainActivity

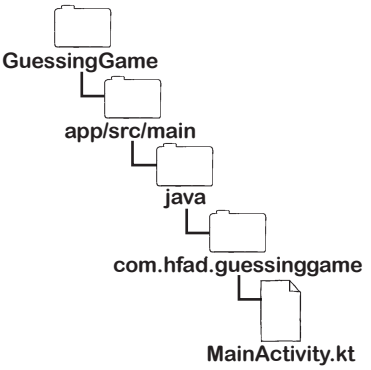
Для отображения фрагментов необходимо включить в макет `MainActivity` хост навигации, связанный с только что созданным графом навигации. Для этого мы воспользуемся представлением `FragmentContainerView`, как и в предыдущих главах.

Код для решения этой задачи уже знаком вам по предыдущим главам. Обновите файл *activity_main.xml*, чтобы он соответствовал приведенному ниже:



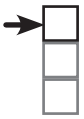

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true" />
```

После обновления макета откройте файл *MainActivity.kt* и убедитесь в том, что он содержит следующий код:




```
package com.hfad.guessinggame

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

И это все, что необходимо сделать с `MainActivity`. На следующем шаге мы обновим код двух фрагментов, начиная с `GameFragment`.



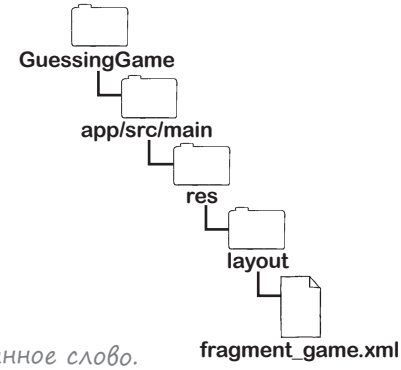
Базовая игра
GameFragment
ResultFragment

Обновление макета GameFragment

`GameFragment` — главный экран приложения, на котором пользователь взаимодействует с игрой. В его макет необходимо добавить несколько представлений: три текстовых представления для угадываемого слова, количество оставшихся жизней и сделанных ошибочных предположений, текстовое поле для ввода буквы и кнопки для обработки предположения.

Чтобы включить эти представления в макет, обновите код *fragment_game.xml*, чтобы он совпадал с приведенным ниже кодом:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".GameFragment">

    <TextView
        android:id="@+id/word"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textSize="36sp"
        android:letterSpacing="0.1" />

    <TextView
        android:id="@+id/lives"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp" />

    <TextView
        android:id="@+id/incorrect_guesses"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp" />
```



Загаданное слово.

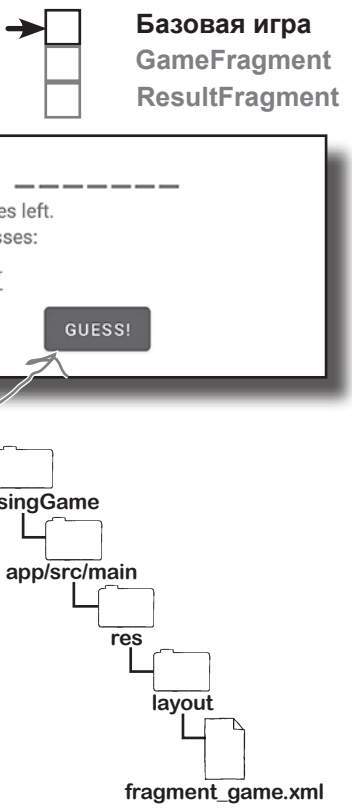
Количество оставшихся жизней.

Ошибочные предположения, сделанные пользователем.

Продолжение на следующей странице.

fragment_game.xml (продолжение)

```
    <EditText
        android:id="@+id/guess"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:hint="Guess a letter"
        android:inputType="text"
        android:maxLength="1" />

    <Button
        android:id="@+id/guess_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Guess!"/>
</LinearLayout>
```



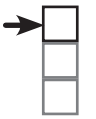
И это все, что необходимо включить в макет `GameFragment`. Теперь нужно определить поведение игры, для чего мы внесем изменения в ее код Kotlin.

Что должен делать фрагмент GameFragment

В первой версии приложения файл *GameFragment.kt* должен включать весь код, необходимый для игры, а также для обновления экрана и обеспечения навигации. Он должен:

- **Выбрать случайное слово.**
  Мы определим список, из которого будет выбираться загаданное слово.
- **Дать пользователю возможность ввести букву.**
- **Отреагировать на предположение.**
  Если пользователь угадал правильно, фрагмент `GameFragment` должен добавить букву в список правильных предположений и показать, где в загаданном слове встречается эта буква. Если буква в слове не встречается, она добавляется в строку ошибочных предположений, а количество оставшихся жизней уменьшается.
- **Перейти к ResultFragment при завершении игры.**

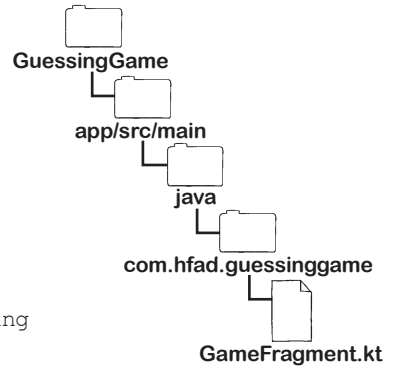
Реализация состоит из кода Kotlin и кода Android, который уже встречался вам ранее. Полный код приводится на нескольких ближайших страницах.



Базовая игра
GameFragment
ResultFragment

Код GameFragment.kt

Ниже приводится код `GameFragment`; обновите свою версию файла *GameFragment.kt*, чтобы она соответствовала приведенной ниже:




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!

    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = ""
    var correctGuesses = ""
    var incorrectGuesses = ""
    var livesLeft = 8

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root

        secretWordDisplay = deriveSecretWordDisplay()
        updateScreen()
```

Эти свойства используются для связывания представлений.

Список возможных слов. В реальном приложении список должен быть более длинным, чтобы игра не была слишком простой.

Слово, которое нужно угадать.

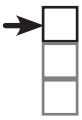
Вид, в котором слово отображается на экране.

Количество правильных и неправильных предположений.

Количество оставшихся жизней.

Определить, в каком виде должно отображаться загаданное слово, и обновить экран.

Продолжение на следующей странице.



Базовая игра
GameFragment
ResultFragment

GameFragment.kt (продолжение)

```
        binding.guessButton.setOnClickListener() {
            makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            updateScreen()
            if (isWon() || isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    fun updateScreen() {
        binding.word.text = secretWordDisplay
        binding.lives.text = "You have $livesLeft lives left."
        binding.incorrectGuesses.text = "Incorrect guesses: $incorrectGuesses"
    }

    fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }
```

Вызываем makeGuess, чтобы обработать предположение пользователя.

Очистить текстовое поле и обновить экран.

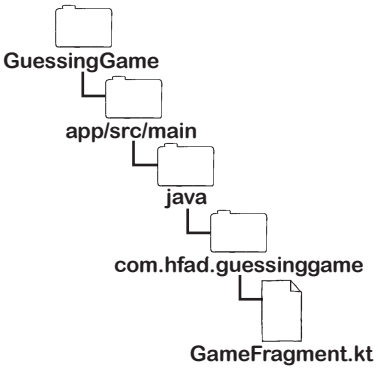
Если пользователь выиграл или проиграл, перейти к ResultFragment с передачей возвращаемого значения wonLostMessage().

Если макет стал недоступным для его фрагмента, свойству _binding присваивается null.

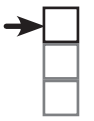
Задаем содержимое текстовых представлений макета.

Здесь создается строка с формой, в которой загаданное слово должно отображаться на экране.

Вызываем checkLetter для каждой буквы secretWord и присоединяем возвращаемое значение к переменной display.



Продолжение на следующей странице.



Базовая игра
GameFragment
ResultFragment

GameFragment.kt (продолжение)

```
    fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
                                        true -> str
                                        false -> "_"
                                    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                secretWordDisplay = deriveSecretWordDisplay()
            } else {
                incorrectGuesses += "$guess "
                livesLeft--
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay, true)

    fun isLost() = livesLeft <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

Проверяет, содержит ли загаданное слово букву, введенную пользователем. Если содержит, то возвращается буква, а если нет, возвращается «_».

Вызывается каждый раз, когда пользователь вводит предположение.

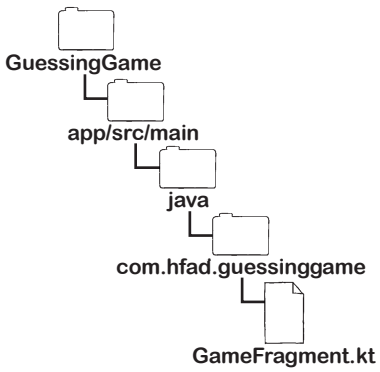
Для каждого правильного предположения обновить correctGuesses и secretWordDisplay.

Для каждого ошибочного предположения обновить incorrectGuesses и livesLeft.

Пользователь выиграл, если загаданное слово совпадает с secretWordDisplay.

Пользователь проиграл, если у него кончились жизни.

wonLostMessage() возвращает строку, которая сообщает, выиграл или проиграл пользователь и какое слово было загадано.



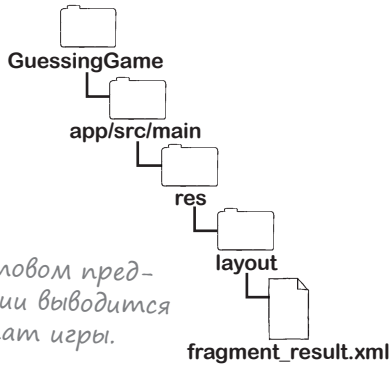
И это весь код, необходимый для `GameFragment`. Перейдем к коду `ResultFragment`.

Обновление макета ResultFragment

`ResultFragment` использует текстовое представление для вывода информации о том, выиграл он или проиграл, и кнопку для запуска следующей игры. Оба представления необходимо добавить в макет фрагмента.

Этот код вам уже знаком. Откройте файл *fragment_result.xml* и обновите его содержимое, чтобы оно совпадало с приведенным ниже:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".ResultFragment">

    <TextView
        android:id="@+id/won_lost"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="28sp" />

    <Button
        android:id="@+id/new_game_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Start new game"/>
</LinearLayout>
```



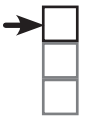
В текстовом представлении выводится результат игры.

Кнопка позволяет начать игру заново.

Также необходимо обновить ResultFragment.kt

После того как текстовое представление и кнопка будут добавлены в макет `ResultFragment`, необходимо определить их поведение в коде фрагмента, написанном на Kotlin. Содержимое текстового представления должно обновляться результатом, а по щелчку на кнопке приложение должно возвращаться к `GameFragment`.

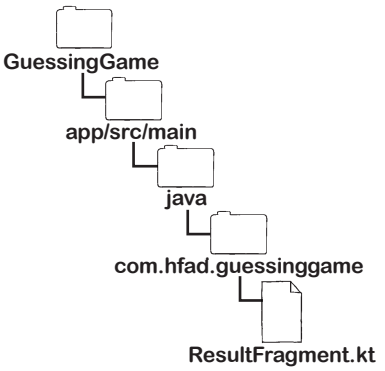
Код *ResultFragment.kt* приведен на следующей странице.



Базовая игра
GameFragment
ResultFragment

Код ResultFragment.kt

Ниже приведен код `ResultFragment`; обновите содержимое файла *ResultFragment.kt*, чтобы оно соответствовало приведенному ниже:



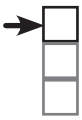

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentResultBinding
import androidx.navigation.findNavController

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        val view = binding.root

        binding.wonLost.text = ResultFragmentArgs.fromBundle(requireArguments()).result

        binding.newGameButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

В этом фрагменте также будет использоваться связывание представлений.

Текстовое представление заполняется строкой, переданной из GameFragment.

По щелчку на кнопке происходит переход к GameFragment.

И это весь код, необходимый для этой версии приложения Guessing Game. Давайте разберемся, что происходит при выполнении кода, и проведем тест-драйв приложения.

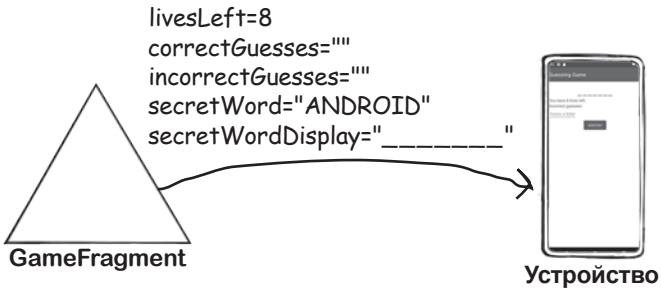


Базовая игра
GameFragment
ResultFragment

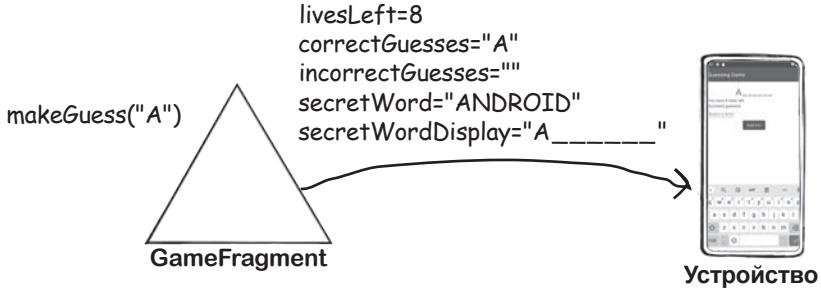
Что происходит при выполнении приложения

Во время выполнения приложения происходят следующие события:

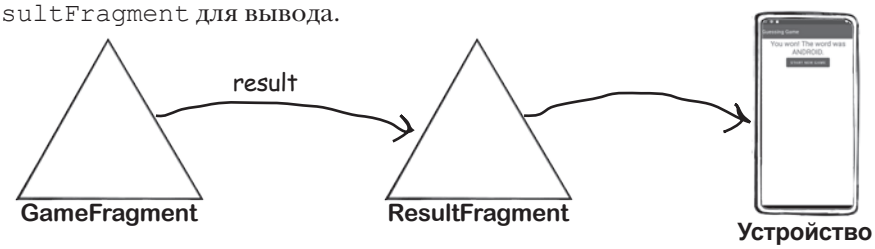
1. **Приложение запускается, и фрагмент GameFragment отображается в MainActivity.**
`GameFragment` присваивает `livesLeft` значение 8, `correctGuesses` и `incorrectGuesses` — пустую строку `""`, `secretWord` — случайно выбранное слово, а `secretWordDisplay` — значение `deriveSecretWordDisplay()`. Затем вызывается метод `updateScreen()`, который выводит значения `livesLeft`, `incorrectGuesses` и `secretWordDisplay`.



2. **Когда пользователь вводит предположение, GameFragment вызывает свой метод makeGuess().**
Метод проверяет, содержит ли `secretWord` букву, введенную пользователем. Если буква найдена, то `makeGuess()` добавляет букву в `correctGuesses` и обновляет `secretWordDisplay`. Если буква отсутствует, метод добавлят букву в `incorrectGuesses` и уменьшает `livesLeft` на 1. Затем метод `updateScreen()` снова вызывается для отображения новых значений.



3. **После каждого предположения GameFragment проверяет результаты isWon() и isLost() на истинность.**
Эти методы проверяют, угадал ли пользователь все буквы слова или у него кончились жизни. Если один из двух методов возвращает *true*, то `GameFragment` передает результат фрагменту `ResultFragment` для вывода.

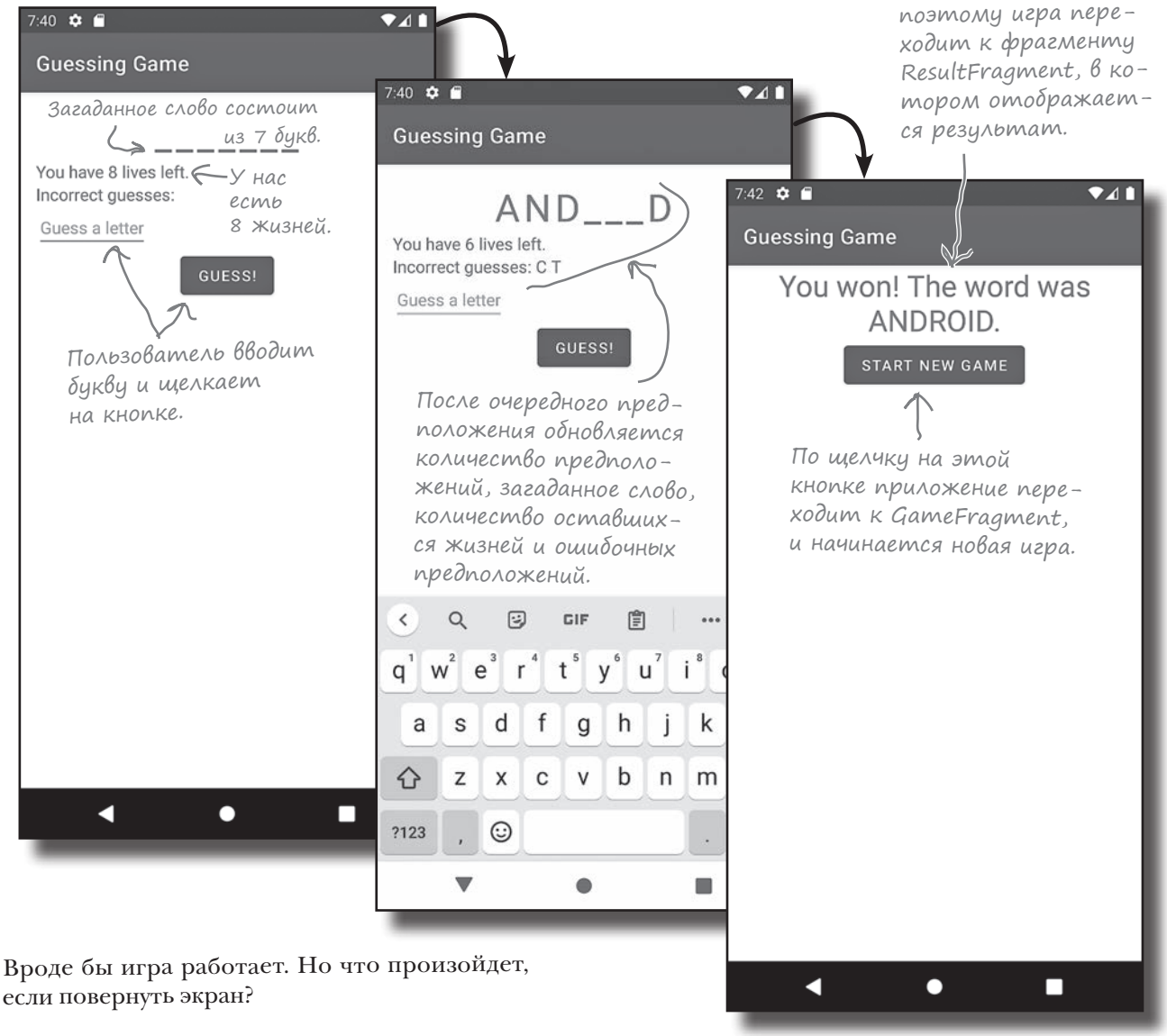


Тест-драйв

Базовая игра
GameFragment
ResultFragment

При запуске приложения отображается фрагмент `GameFragment`. Он показывает, сколько букв содержит загаданное слово и сколько жизней осталось у пользователя.

Чтобы сделать предположение, пользователь вводит букву в текстовом поле и щелкает на кнопке. Если предположение оказалось правильным, буква включается в загаданное слово, но в случае ошибки теряется одна жизнь. Если пользователь угадал все буквы или у него кончились жизни, в `ResultFragment` выводится сообщение о выигрыше или проигрыше.

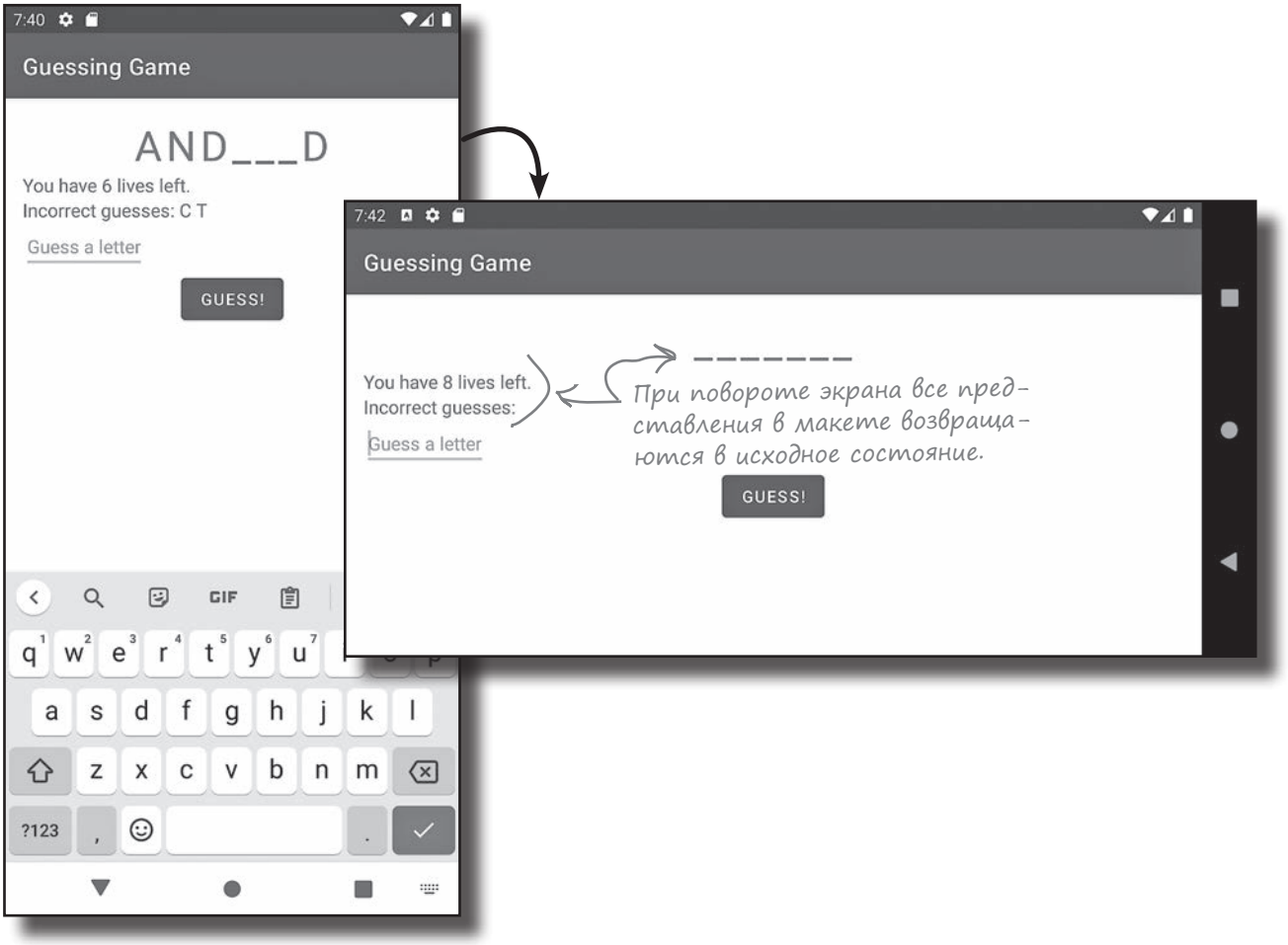


Вроде бы игра работает. Но что произойдет, если повернуть экран?

Базовая игра
GameFragment
ResultFragment

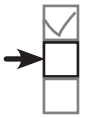
При повороте экрана состояние теряется

Однако в игре есть проблема. Если повернуть экран во время игры, приложение теряет свое состояние и игра начинается заново.



Игра теряет состояние, потому что поворот экрана изменяет конфигурацию приложения; Android уничтожает активность (и отображаемый ею фрагмент) и немедленно создает ее заново. При этом представления и свойства игры возвращаются к исходным значениям.

Проблему можно решить сохранением состояния всех свойств в методе `onSaveInstanceState` фрагмента, как делалось ранее в этой книге. Однако на этот раз мы воспользуемся другим решением и **реализуем модель представления**.



Базовая игра
GameFragment
ResultFragment

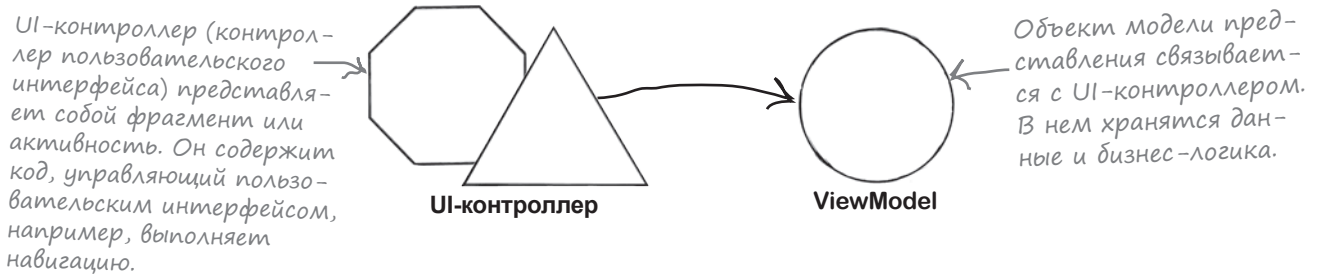
Модель представления содержит бизнес-логику

Как было сказано выше, модель представления — это отдельный класс, который существует параллельно с кодом активности или фрагмента. Он отвечает за данные, которые должны отображаться на экране, а также может включать любую бизнес-логику. Так, в приложении Guessing Game это означает, что модель представления должна хранить свойства игры (например, слово, которое пользователь пытается отгадать, и количество оставшихся жизней) и любые методы, управляющие ходом игры.

Модель представления используется для упрощения кода активностей и фрагментов, а также для сохранения состояния свойств, чтобы они нормально пережили изменения конфигурации.

Когда вы реализуете модель представления, весь код, относящийся к данным приложения или бизнес-логике, перемещается из активности или фрагмента в модель представления. Весь код, управляющий пользовательским интерфейсом (например, вывод текста или получение данных от пользователя), остается в коде активности или фрагмента.

Этот способ формирования структуры приложений Android следует принципу проектирования, известному как *принцип разделения обязанностей*. Приложение разбивается на классы, при этом каждый класс предназначен для выполнения некоторой отдельной обязанности. Контроллер пользовательского интерфейса — код активности или фрагмента — обеспечивает работу пользовательского интерфейса, а модель представления отвечает за бизнес-логику и данные:

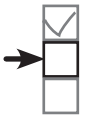


Такая архитектура приложения имеет два ключевых преимущества.

- **Она упрощает код активностей и фрагментов.**
  Выделение данных и бизнес-логики приложения в модель представления означает уменьшение объема кода активностей и фрагментов, который вам придется сопровождать.

- **Ваше приложение справляется с изменениями конфигурации.**
  Модель представления — это отдельный класс, который существует наряду с кодом активностей или фрагментов. Она не уничтожается при повороте экрана устройства, так что состояние свойств, хранящихся в модели представления, переживет изменения конфигурации, и вам не придется добавлять их значения в `Bundle`.

Теперь вы знаете, что вам даст использование модели представления. Посмотрим, как добавить модель представления в приложение Guessing Game.



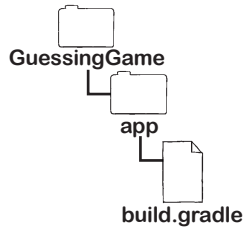
Включение зависимости для модели представления в файл build.gradle приложения...

Библиотека моделей представлений является частью Android Jetpack, поэтому вам придется обновить файл *build.gradle* приложения и включить ее как зависимость.

Откройте файл *GuessingGame/app/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел dependencies:

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
    ...
}
```

Используемая версия библиотеки.



Синхронизируйте изменения, когда вам будет предложено это сделать. Теперь все готово к созданию модели представления.

...и создание модели представления

Мы создадим модель представления с именем `GameViewModel`, которая будет использоваться фрагментом `GameFragment` для хранения его логики и данных.

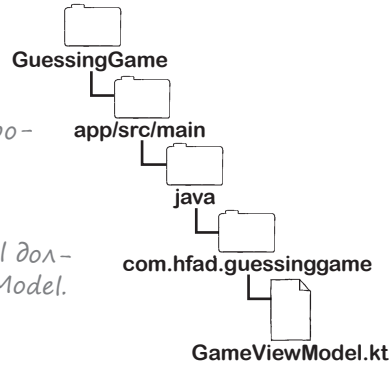
Выделите пакет *com.hfad.guessinggame* в папке *app/src/main/java* folder и выберите команду File→New→Kotlin Class/File. Введите имя файла «GameViewModel» и выберите вариант создания класса.

Когда файл *GameViewModel.kt* будет создан, обновите его код, чтобы он соответствовал приведенному ниже (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {

}
```

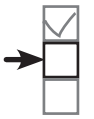
Необходимо импортировать этот класс.

Класс GameViewModel должен расширять ViewModel.



Как видите, класс `GameViewModel` расширяет `androidx.lifecycle.ViewModel`. `ViewModel` — абстрактный класс, который используется для преобразования обычного традиционного класса в полноценную модель представления.

После того как мы добавили класс `GameViewModel` в проект Guessing и преобразовали его в модель представления, можно переходить к написанию оставшегося кода.



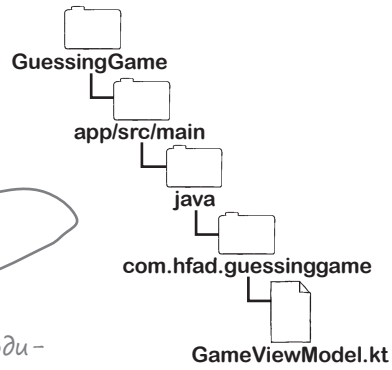
Базовая игра
GameFragment
ResultFragment

Полный код GameViewModel.kt

Как вы уже узнали, модель представления отвечает за бизнес-логику и данные активности или фрагмента. Для игры Guessing Game это означает, что мы должны определить все свойства и методы `GameFragment`, относящиеся к игровому процессу, и переместить их в `GameViewModel`. Весь код, относящийся к навигации или пользовательскому интерфейсу, останется в `GameFragment`.

Ниже приведен полный код *GameViewModel.kt*; обновите код и включите в него изменения (выделенные жирным шрифтом):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {
    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = ""
    var correctGuesses = ""
    var incorrectGuesses = ""
    var livesLeft = 8

    init {
        secretWordDisplay = deriveSecretWordDisplay()
    }

    fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }
```

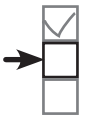


Свойства, необходимые для игры.

Этот метод вызывается в блоке init, который выполняется при инициализации класса.

Вид, в котором загаданное слово отображается на экране.

Продолжение на следующей странице.



Базовая игра
GameFragment
ResultFragment

Полный код GameViewModel.kt (продолжение)

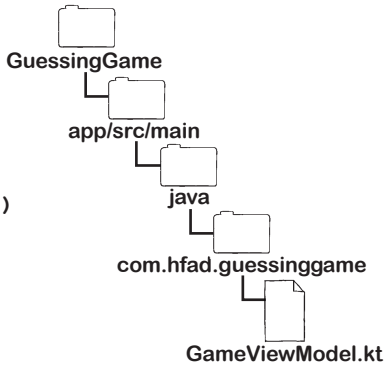
```
    fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                secretWordDisplay = deriveSecretWordDisplay()
            } else {
                incorrectGuesses += "$guess "
                livesLeft--
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay, true)

    fun isLost() = livesLeft <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

Возвращает str, если буква встречается в загаданном слове, или "_", если буква отсутствует.

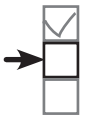
Вызывается, когда пользователь вводит предположение.

Проверяет выигрыш или проигрыш пользователя.

Генерирует строку при завершении игры.



Вот и все, что необходимо сделать для `GameViewModel`. На следующем шаге мы свяжем класс с `GameFragment` и обновим код этого фрагмента.



Базовая игра
GameFragment
ResultFragment

Создание объекта GameViewModel

Чтобы связать модель представления с активностью или фрагментом, следует добавить в код свойство `ViewModel` и инициализировать его объектом `ViewModel`, который необходимо создать. Код выглядит так:

```
class GameFragment : Fragment() {
    ...
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
        ...
    }
}
```

Определяем свойство viewModel, которое будет инициализировано позднее в коде.

Эта строка получает объект GameViewModel и присваивает его свойству viewModel.

:: в этом коде используется для получения ссылки на класс GameViewModel. Необходимость использования этого синтаксиса объясняется тем, что метод get() класса ViewModelProvider требует ссылки на класс GameViewModel вместо объекта GameViewModel.

В этом коде для создания объекта `ViewModel` используется **провайдер модели представления**. Что же происходит при использовании этого класса?

Использование класса ViewModelProvider для создания моделей представлений

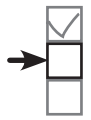
Как подсказывает имя, `ViewModelProvider` — специальный класс, который должен предоставлять модели представлений активностям и фрагментам. Он гарантирует, что **новый объект модели представления создается только в том случае, если он не существует**.

Как вы уже знаете, при повороте экрана любые фрагменты, отображаемые на экране, уничтожаются и создаются заново. Когда это происходит, провайдер модели представления следит за тем, чтобы использовался тот же объект модели представления. Модель представления сохраняет свое состояние, так что любые свойства, используемые фрагментом, не сбрасываются.

Провайдер модели представления хранит модель представления, пока активность или фрагмент продолжают существовать. Когда фрагмент отсоединяется или удаляется из своей активности, провайдер разрывает связь с моделью представления фрагмента. Когда провайдер модели представления получает следующий запрос на объект модели представления, он создает новый объект.

Итак, теперь вы знаете, как связать модель представления с фрагментом, и мы можем перейти к обновлению кода `GameFragment`.

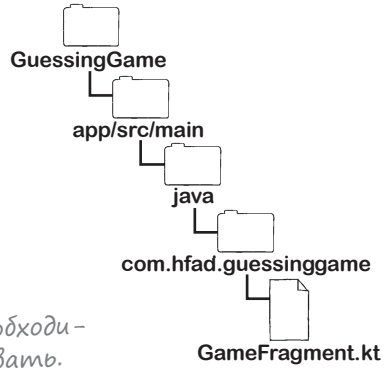
ViewModelProvider предоставляет фрагменту GameFragment экземпляр GameViewModel. Новый объект GameViewModel создается только в том случае, если он не существует.



Обновленный код GameFragment.kt

Из `GameFragment` следует удалить свойства и методы, перемещенные в `GameViewModel`, и заставить фрагмент использовать модель представления.

Ниже приведен обновленный код `GameFragment`; убедитесь в том, что файл *GameFragment.kt* включает эти изменения (выделенные жирным шрифтом):




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = ""
    var correctGuesses = ""
    var incorrectGuesses = ""
    var livesLeft = 8

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

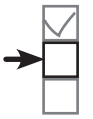
Этот класс необходимо импортировать.

Добавляем свойство viewModel.

Эти свойства перемещены в GameViewModel, из GameFragment их следует удалить.

Задаем свойство viewModel.

Продолжение на следующей странице.



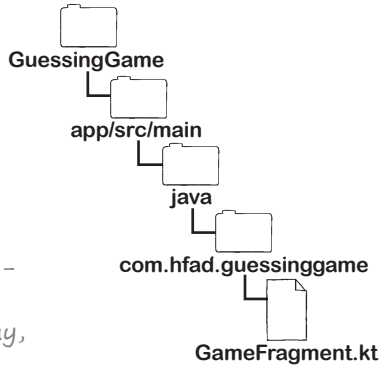
Базовая игра
GameFragment
ResultFragment

Обновленный код GameFragment.kt (продолжение)

```
        ~~secretWordDisplay = deriveSecretWordDisplay()~~
        updateScreen()

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            updateScreen()
            if (viewModel.isWon() || viewModel.isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    fun updateScreen() {
        binding.word.text = viewModel.secretWordDisplay
        binding.lives.text = "You have ${viewModel.livesLeft} lives left."
        binding.incorrectGuesses.text = "Incorrect guesses: ${viewModel.incorrectGuesses}"
    }

    ~~fun deriveSecretWordDisplay()...~~
    ~~fun checkLetter(str: String)...~~
    ~~fun makeGuess(guess: String)...~~
    ~~fun isWon()...~~
    ~~fun isLost()...~~
    ~~fun wonLostMessage()...~~
}
```

Удалите эту строку.

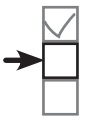
К методам makeGuess(), isWon(), isLost() и wonLostMessage() необходимо обращаться через свойство viewModel.



Свойство viewModel будет использоваться для обращения к свойствам secretWordDisplay, livesLeft и incorrectGuesses.

Эти методы перемещены в GameViewModel, поэтому их следует удалить из GameFragment.

И это все изменения, которые необходимо внести в `GameFragment`. Давайте разберемся, что происходит при его выполнении, и проведем тест-драйв.

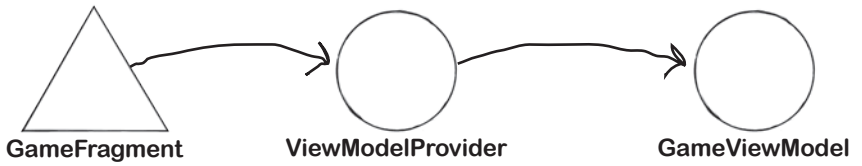


Базовая игра
GameFragment
ResultFragment

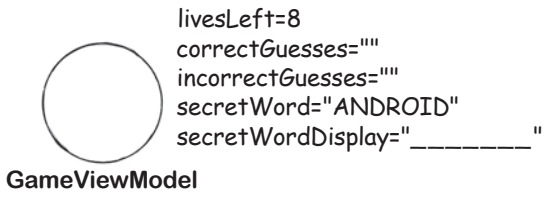
Что происходит при выполнении приложения

Во время выполнения приложения происходят следующие события:

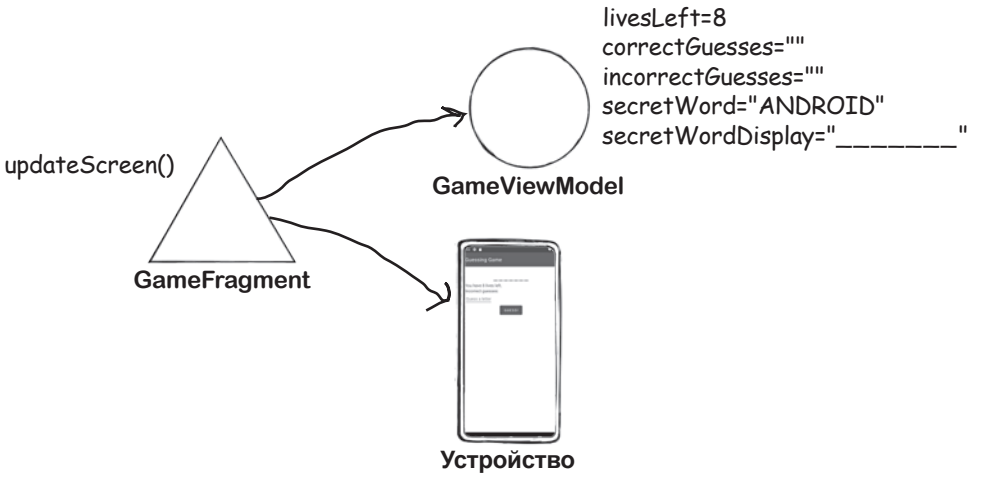
1. **GameFragment запрашивает у класса ViewModelProvider экземпляр GameViewModel.** Провайдер модели представления видит, что в настоящий момент объект `GameViewModel`, связанный с фрагментом, не существует, поэтому он создает новый экземпляр.

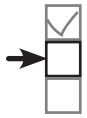


2. **Объект GameViewModel инициализируется.** Свойству `livesLeft` присваивается 8, `correctGuesses` и `incorrectGuesses` присваивается `""`, `secretWord` — случайно выбранное слово, а `secretWordDisplay` — результат `deriveSecretWordDisplay()`.



3. **GameFragment вызывает метод updateScreen().** Метод обращается к свойствам `secretWordDisplay`, `livesLeft` и `incorrectGueses` объекта `GameViewModel` и выводит их на экран.



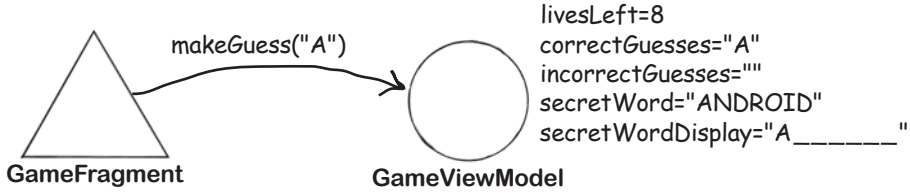


Базовая игра
GameFragment
ResultFragment

История продолжается

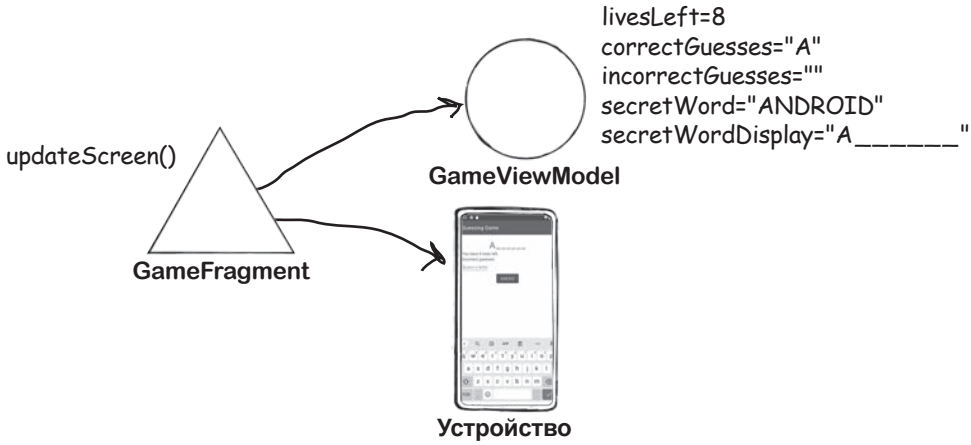
4 **Когда пользователь вводит предположение, GameFragment вызывает метод makeGuess() объекта GameViewModel.**

Метод проверяет, содержит ли `secretWord` введенную букву. Если буква встречается в слове, то она добавляется в `correctGuesses`, а значение `secretWordDisplay` обновляется. Если буква отсутствует в слове, она добавляется в `incorrectGuesses`, а значение `livesLeft` уменьшается на 1.



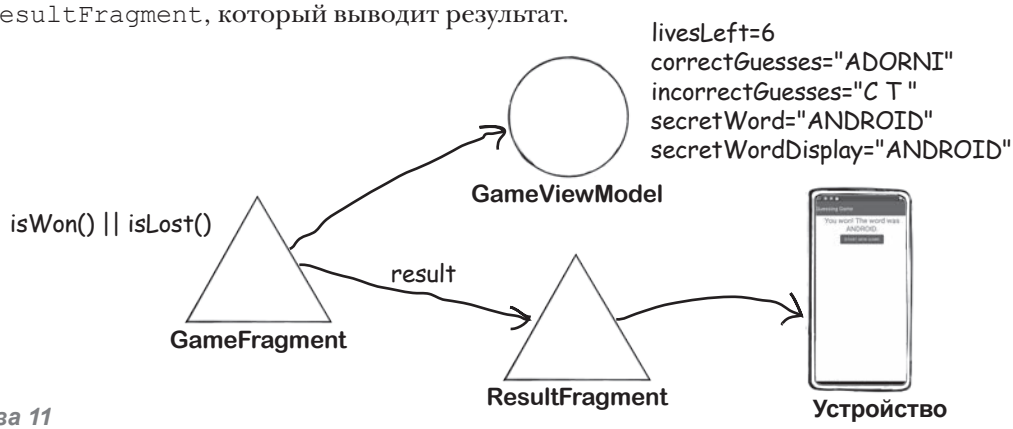
5 **GameFragment снова вызывает свой метод updateScreen().**

Метод получает обновленные значения свойств из объекта `GameViewModel` и обновляет экран.



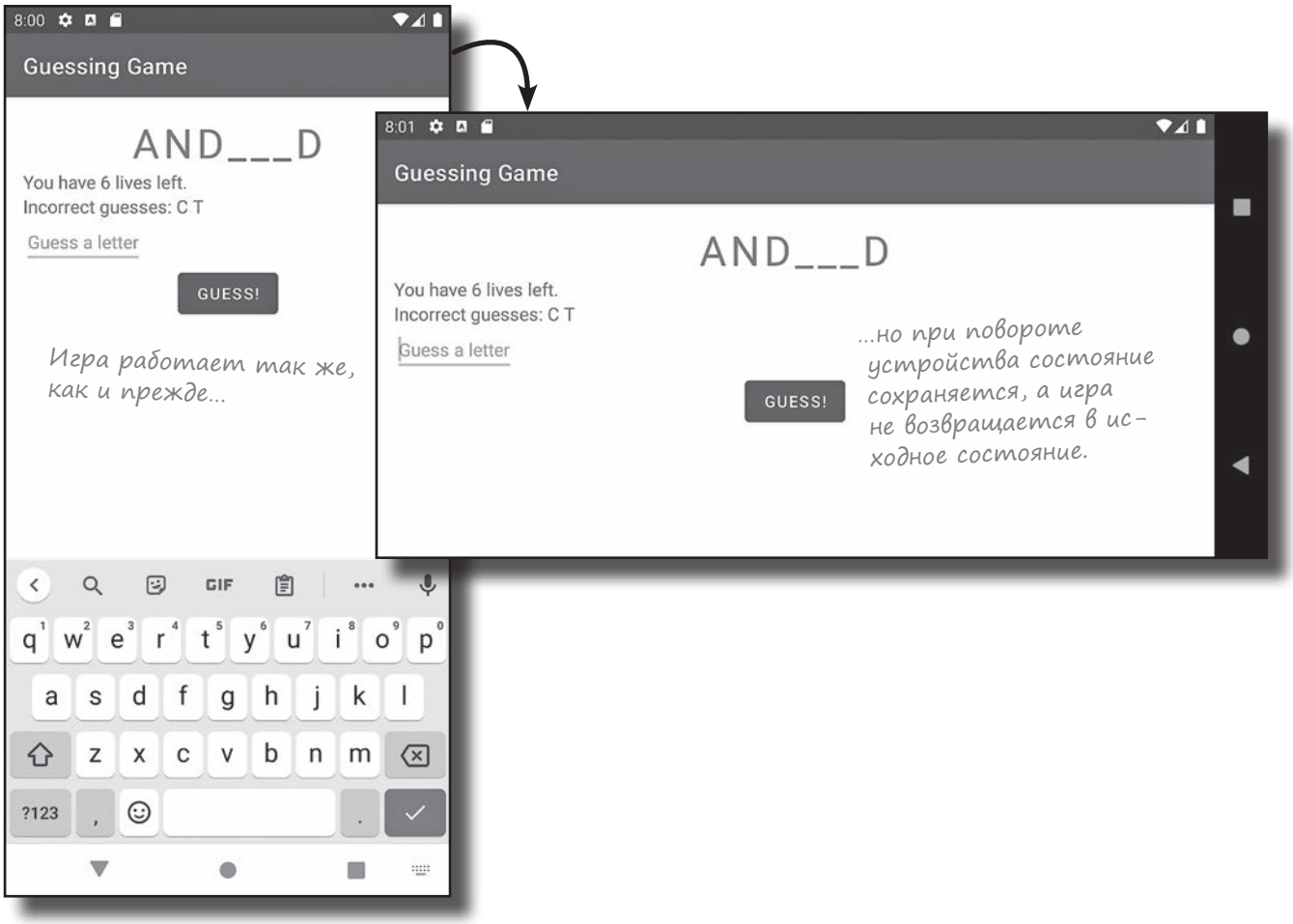
6 **После каждого предположения GameFragment проверяет, возвращает ли значение true один из методов isWon() или isLost() модели представления.**

Если хотя бы один из методов возвращает *true*, `GameFragment` передает результат фрагменту `ResultFragment`, который выводит результат.



Тест-драйв

При запуске приложения отображается фрагмент `GameFragment`, как и прежде. Если начать игру и повернуть экран, игра сохраняет свое состояние.



Вы узнали, как добавить модель представления в ваши приложения и как использовать ее для предотвращения проблем, которые могут возникнуть при повороте экрана устройства. Но прежде чем двигаться дальше, нужно поближе познакомиться с моделями представлений.

Модели представлений под увеличительным стеклом

Модель представления — это отдельный класс, ответственный за бизнес-логику и данные UI-контроллера. Этот класс расширяет `androidx.lifecycle.ViewModel` и дополнительно может переопределять метод **`onCleared()`**. Этот метод вызывается непосредственно перед уничтожением модели и дает возможность освободить любые ресурсы:

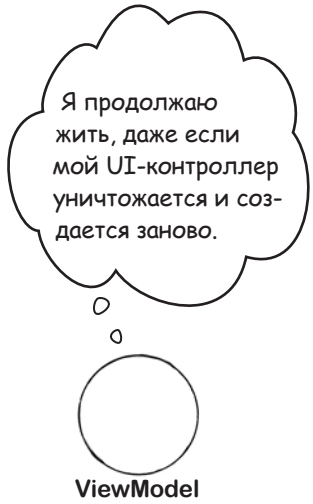
UI-контроллером может быть активность или фрагмент.

```
import androidx.lifecycle.ViewModel

class MyViewModel : ViewModel() {
    //Код инициализации модели представления

    override fun onCleared() {
        //Код, выполняемый перед очисткой модели представления
    }
}
```

Например, объект модели представления создается с использованием класса `ViewModelProvider`:

```
viewModel = ViewModelProvider(this).get(MyViewModel::class.java)
```

Приведенная команда создает модель представления в одной области видимости с UI-контроллером. Она гарантирует, что новая модель представления будет создаваться *только* в том случае, если в области видимости UI-контроллера еще нет существующей модели. Это означает, что модель представления переживет изменения конфигурации и вам не придется сохранять состояние переменных экземпляров в отдельном объекте `Bundle`.



Модель представления продолжает существовать до тех пор, пока ее UI-контроллер не уйдет навсегда. Если она связана с активностью, это происходит при завершении активности, то есть когда она уничтожается и не будет создаваться заново. Если она связана с фрагментом, это происходит при удалении фрагмента или его отсоединения от родительской активности.

Для отслеживания моментов создания или уничтожения модели представления можно воспользоваться классом **`android.util.Log`**. Этот класс позволяет добавлять сообщения в журнал Android (или logcat), который можно просмотреть в Android Studio. Например, следующий код регистрирует сообщение при уничтожении модели представления:

```
override fun onCleared() {
    Log.i("MyViewModel", "ViewModel cleared")
}
```

Регистрирует сообщение «ViewModel cleared» и снабжает его меткой «MyViewModel».

Класс `Log` включает несколько методов, которые могут использоваться для регистрации различных типов сообщений. Например, метод `Log.i()` регистрирует журнальную информацию, а метод `Log.e()` используется для регистрации ошибок.

Ниже перечислены различные методы `Log` и их предназначение:

`Log.v(String tag, String message)`	Регистрирует подробное сообщение.
`Log.d(String tag, String message)`	Регистрирует отладочное сообщение.
`Log.i(String tag, String message)`	Регистрирует информационное сообщение.
`Log.w(String tag, String message)`	Регистрирует предупреждающее сообщение.
`Log.e(String tag, String message)`	Регистрирует сообщение об ошибке.

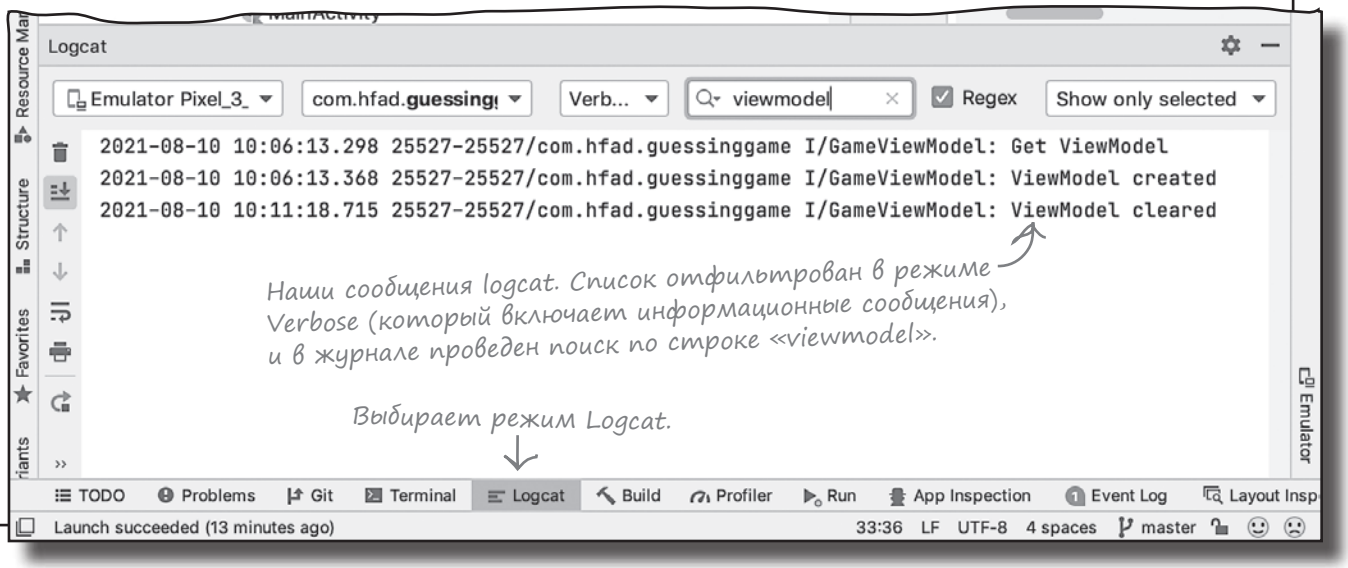
Также существует метод Log.wtf(), который предназначен для оповещения о ситуациях, которые никогда не должны возникать. Согласно документации Android, сокращение wtf означает «What a Terrible Failure», то есть «Что за ужасная ошибка».

Каждое сообщение состоит из строковой метки, которая используется для идентификации источника сообщений, и самого сообщения. Таким образом, код:

```
Log.i("MyViewModel", "ViewModel cleared")
```

регистрирует информационное сообщение, поступающее от `MyViewModel`.

Чтобы просмотреть журнал Android Studio, выберите вкладку Logcat в нижней части экрана. Содержимое журнала можно фильтровать по типу сообщений, а также проводить поиск текста:

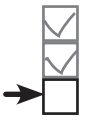


СТАНЬ моделью представления

Приведенный ниже код описывает класс модели представления с именем MyViewModel. Представьте себя на месте модели представления и скажите, какие проблемы встречаются в этом коде и как их исправить.

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log
import android.widget.TextView

class MyViewModel {
    val num = 2

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * num
    }

    fun joinTogether(text1: TextView, text2: TextView): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1.text} ${text2.text}")
    }
}
```

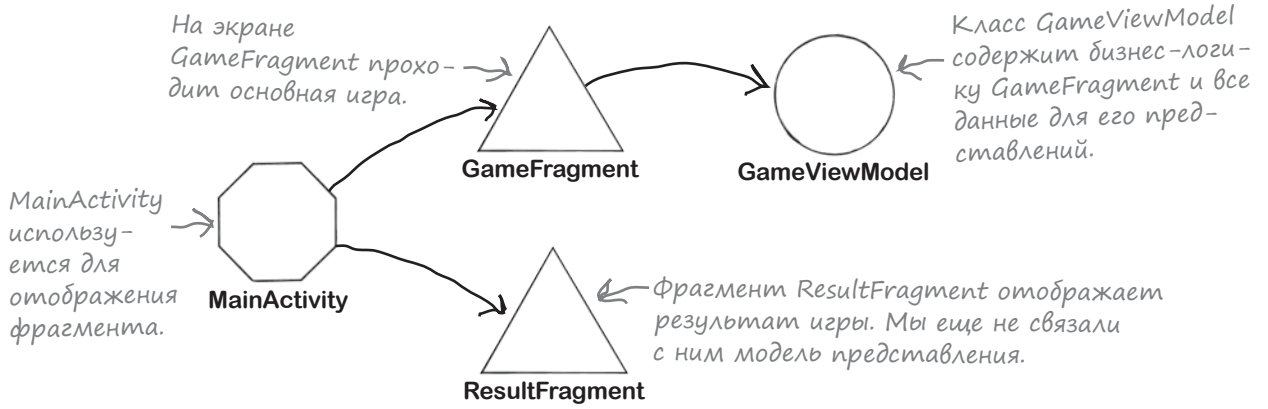
Ответ на с. 519.



Базовая игра
GameFragment
ResultFragment

Мы добавили модель представления для GameFragment

К настоящему моменту мы обновили приложение Guessing Game, чтобы фрагмент `GameFragment` использовал модель представления с именем `GameViewModel`, которая отвечает за всю бизнес-логику и данные фрагмента. Такое использование модели представления упрощает код *GameFragment.kt* и означает, что приложение не будет терять состояние при повороте экрана.



ResultFragment тоже нужна модель представления

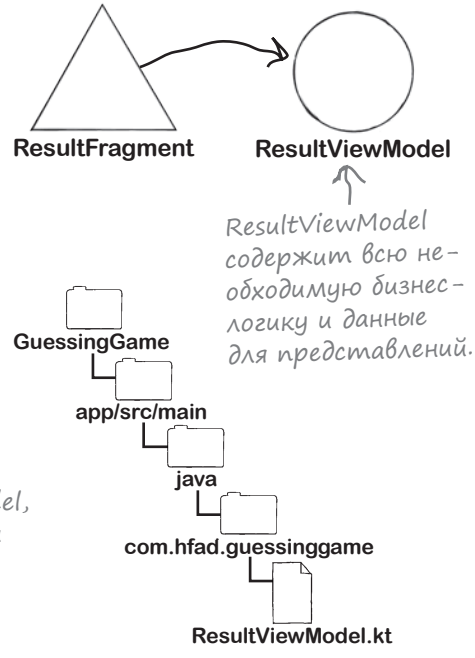
Каждая модель представления, которую вы создаете, связывается с одним UI-контроллером — активностью или фрагментом. Таким образом, если вы хотите, чтобы фрагмент `ResultFragment` использовал модель представления, необходимо создать новую модель для этого фрагмента.

Давайте сделаем это. Выделите пакет *com.hfad.guessinggame* в папке *app/src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя файла «ResultViewModel» и выберите вариант создания класса.

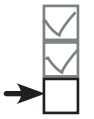
Когда файл *ResultViewModel.kt* будет создан, обновите код и приведите его к следующему виду (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class ResultViewModel : ViewModel(){
}
```

Новый класс должен расширять ViewModel, поэтому нужны эти изменения.



Это базовый код, необходимый для определения модели представления. Какой еще код следует добавить?



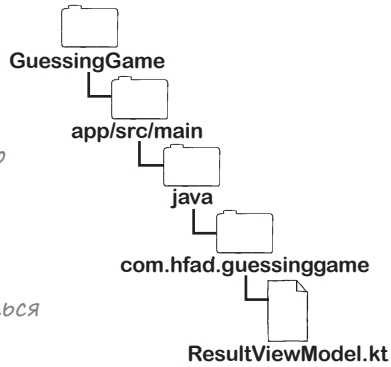
Базовая игра
GameFragment
ResultFragment

ResultViewModel необходимо хранить результат

Как говорилось ранее, `ResultFragment` выводит в своем макете сообщение о том, выиграл он или проиграл текущую игру. Это сообщение передается `ResultFragment` фрагментом `GameFragment` при завершении игры.

В новой версии приложения `ResultViewModel` отвечает за игровую логику и данные `ResultFragment`, поэтому в `ResultViewModel` необходимо включить свойство для хранения результата. Мы также используем конструктор с параметром `String`, чтобы значение свойства задавалось сразу же после создания `ResultViewModel`.

Ниже приведен полный код *ResultViewModel.kt*; обновите свою версию файла (изменения выделены жирным шрифтом):




```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel

class ResultViewModel(finalResult: String) : ViewModel(){
    val result = finalResult
}
```

Добавляем конструктор с параметром String...

...который будет использоваться для задания свойства result.

Затем обновим фрагмент `ResultFragment`, чтобы в нем использовалась новая модель представления.

Связываем ResultViewModel с ResultFragment

Ранее для добавления ссылки `GameViewModel` в `GameFragment` использовался следующий код:

```
viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

В этом коде используется класс ViewModelProvider для получения ссылки на объект GameViewModel.

Он отдает команду провайдеру модели представления получить объект `GameViewModel`, связанный с фрагментом, или создать новый объект, если он еще не существует.

Однако этот способ не может использоваться для включения ссылки на `ResultViewModel` в `ResultFragment`. Дело в том, что **он подходит только для моделей представлений с конструктором без аргументов**.

Такое решение работало для `GameViewModel`, потому что объект можно было сконструировать для передачи аргументов. Но конструктор класса `ResultViewModel` должен получать строку, поэтому приведенный выше код работать не будет.

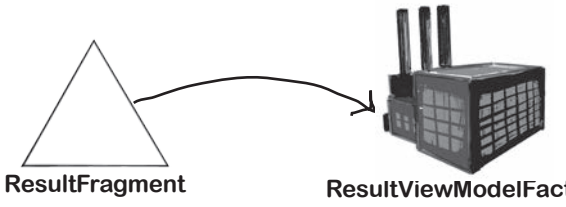
Фабрика модели представления создает модели представлений

Альтернативный способ создания модели представления основан на передаче провайдеру модели представления **фабрики модели представлений**: отдельного класса, единственным назначением которого является создание и инициализация моделей представлений. Такой подход означает, что провайдеру модели представления не придется беспокоиться о создании модели представления своими силами. Вместо этого он использует фабрику модели представления.

Хотя фабрики моделей представлений могут использоваться для любых разновидностей моделей представлений, чаще всего они используются для моделей, конструкторы которых должны получать аргументы. Дело в том, что провайдер модели представления не может передавать аргументы конструктору сам по себе: для этого он должен использовать фабрику модели представления.

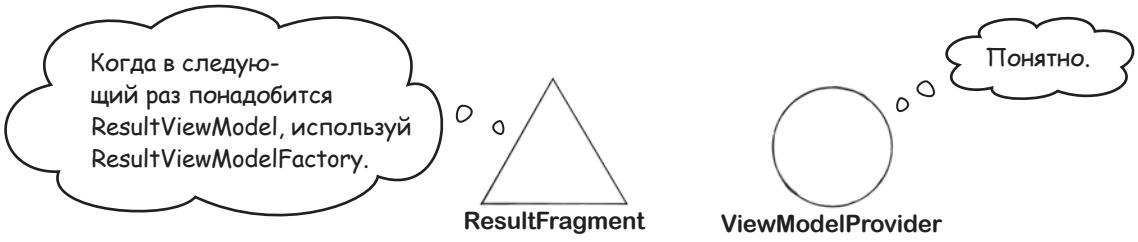
Код использования фабрики модели представления в приложении Guessing Game выглядит так:

1 **Мы определяем класс ResultViewModelFactory, который будет использоваться ResultFragment для создания объекта фабрики.**

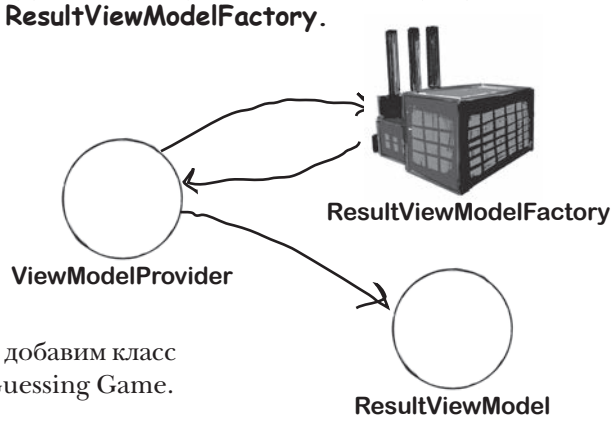


Фабрика модели представления используется для создания модели представления, не имеющей конструктора без аргументов.

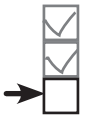
2 **ResultFragment отдает команду провайдеру модели представления использовать объект фабрики.**



3 **Когда провайдеру модели представления потребуется новый объект ResultViewModel, он использует ResultViewModelFactory.**



Сделаем следующий шаг и добавим класс фабрики в приложение Guessing Game.

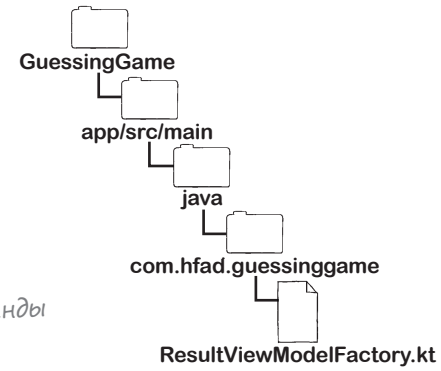


Базовая игра
GameFragment
ResultFragment

Создание класса ResultViewModelFactory

Мы добавим в приложение Guessing Game класс фабрики модели представления с именем `ResultViewModelFactory`. Этот класс будет использоваться провайдером модели представления для создания объекта `ResultViewModel`.

Чтобы создать класс, выделите пакет *com.hfad.guessinggame* в папке */src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя файла «ResultViewModelFactory» и выберите вариант создания класса.



После того как файл *ResultViewModelFactory.kt* будет создан, приведите его к следующему виду (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import java.lang.IllegalArgumentException

class ResultViewModelFactory(private val finalResult: String)
        : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ResultViewModel::class.java))
            return ResultViewModel(finalResult) as T
        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Добавьте эти команды импортирования.

Параметр String в конструкторе необходим для создания объекта ResultViewModel.

Переопределите этот метод, используемый провайдером модели представления для создания объектов модели представления.

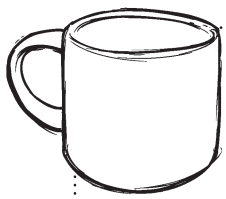
Класс должен реализовать этот интерфейс.

Проверяет, что модель представления, которую пытается создать провайдер модели представления, имеет правильный тип. Если проверка проходит успешно, то модель возвращается.

Если модель представления имеет неправильный тип, выдается исключение.

Как видно из кода, класс `ResultViewModelFactory` реализует интерфейс с именем `ViewModelProvider.Factory` и переопределяет его метод `create()`. Тем самым класс преобразуется в фабрику модели представления, которая может использоваться провайдером модели представления для создания объекта `ResultViewModel`.

Приведенный выше код — все, что необходимо для работы `ResultViewModelFactory`. Посмотрим, как использовать его в коде `ResultFragment`.



Расслабьтесь

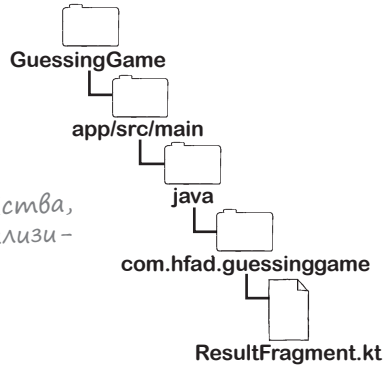
Весь код фабрики модели представления остается практически неизменным.

Не стоит беспокоиться о подробностях. Все, что действительно необходимо знать, — код создает модель представления определенного типа и передает аргумент ее конструктору.

Использование фабрики для создания модели представления

Как говорилось ранее, фабрика модели представления используется для создания модели представления, для чего провайдеру модели представления передается фабрика. Провайдер модели представления решает, нужно ли создать новый объект модели представления, и при необходимости использует фабрику для его создания.

Код, обеспечивающий использование фабрики провайдером модели представления, будет практически идентичным для всех создаваемых вами моделей представлений. Он выглядит примерно так:




```
class ResultFragment : Fragment() {
    ...
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...
        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
                                    .get(ResultViewModel::class.java)
        ...
    }
}
```

Добавьте эти два свойства, которые будут инициализированы позднее в коде.

Создает объект фабрики модели представления.

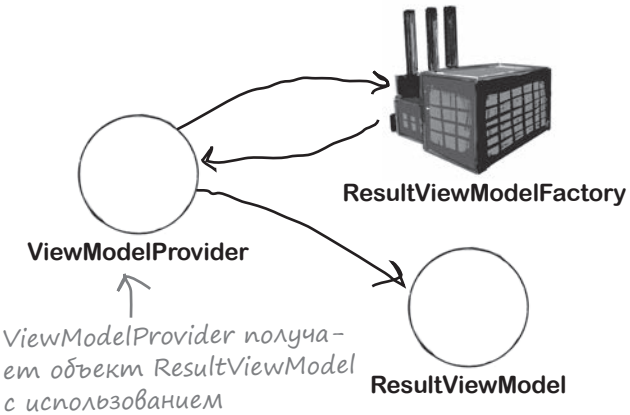
Передает фабрику модели представления провайдеру модели представления.

Провайдер модели представления использует фабрику модели представления для создания нового объекта модели представления, если он еще не существует.

Как видите, в этом коде определяются два свойства: `viewModel` и `viewModelFactory`. Они задаются в методе `onCreateView()` фрагмента.

В `onCreateView()` код использует строку result, переданную ему из `GameFragment`, для создания нового объекта `ResultViewModelFactory`. Он передает фабрику провайдеру модели представления, который использует ее для получения объекта `ResultViewModel`.

Итак, теперь вы знаете, как использовать фабрику для связывания модели представления с фрагментом, и мы можем перейти к обновлению кода `ResultFragment`.

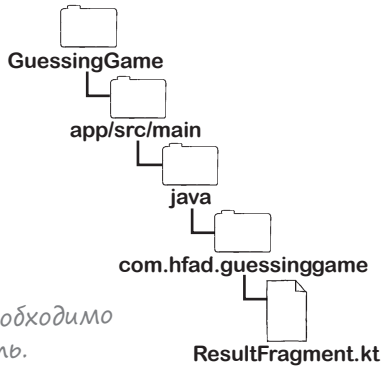


ViewModelProvider получает объект ResultViewModel с использованием ResultViewModelFactory.

Базовая игра
GameFragment
ResultFragment

Обновленный код ResultFragment.kt

Ниже приведен полный код `ResultFragment`. Обновите код в файле *ResultFragment.kt*, чтобы он включал изменения, приведенные ниже (выделены жирным шрифтом):




```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentResultBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        val view = binding.root

        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
                                    .get(ResultViewModel::class.java)
        binding.wonLost.text = ResultFragmentArgs.fromBundle(requireArguments()).result
        binding.wonLost.text = viewModel.result

        binding.newGameButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
        return view
    }
```

Этот класс необходимо импортировать.

Добавьте эти свойства.

Получение строки result, переданной ResultFragment.

Создание фабрики.

Получение модели представления.

Удалите эту строку.

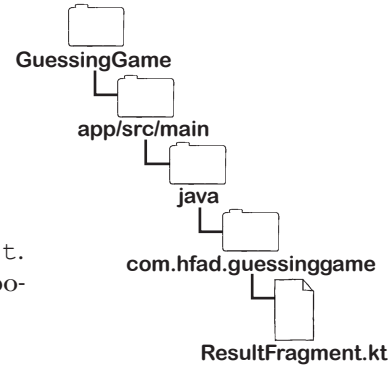
Вывод result из модели представления.

Продолжение на следующей странице.

Обновленный код ResultFragment.kt (продолжение)

```
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Изменять этот метод не нужно.



И это все изменения, которые необходимо внести в `ResultFragment`. После ответов на некоторые вопросы мы разберемся в том, что же происходит во время выполнения приложения.

Часто Задаваемые Вопросы

В: Фрагменту `ResultFragment` действительно нужна собственная модель представления?

О: Мы решили использовать ее для отделения игровой логики и данных, необходимых для `ResultFragment`. В текущей версии приложения она состоит только из строки result, но модель полезно создать на случай, если в будущем приложение станет более сложным. И признаемся честно: это был хороший повод рассказать вам об использовании фабрики модели представления.

В: А почему мы не использовали фабрику при создании `GameViewModel`?

О: Фабрики необходимы только при создании моделей представлений, не имеющих конструктора без аргументов. Так как `GameViewModel` использует конструктор по умолчанию, не имеющий аргументов, использовать фабрику не нужно.

В: Но при желании мы могли использовать фабрику для `GameViewModel`?

О: Да. Фабрики можно использовать для создания любых разновидностей моделей представлений, но если у модели представления есть конструктор по умолчанию, использовать их необязательно.

В: Зачем понадобилось добавлять конструктор в `ResultViewModel`?

О: Чтобы мы могли задать строку result сразу же после создания модели представления. Это гарантирует, что значение строки result будет определено в любой момент, когда `ResultFragment` использует модель представления.

В: Обязательно ли создавать модели представлений с использованием класса `ViewModelProvider`? А что произойдет, если этого не сделать?

О: Да, для создания объекта модели представления всегда следует использовать провайдера модели представления. Провайдер модели представления создает новый объект модели представления *только* в том случае, если он еще не существует в области видимости UI-контроллера; это означает, что модель представления будет нормально переносить изменения конфигурации.

Если вы создадите экземпляр модели представления без использования провайдера, он будет заново создаваться каждый раз, когда UI-контроллер уничтожается и создается заново. Модель представления будет терять свое состояние при каждом повороте экрана устройства.

В: Можно ли проводить модульное тестирование моделей представлений?

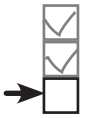
О: Да! Модели представлений не обращаются к экранным представлениям, что позволяет проводить их модульное тестирование независимо от активностей и фрагментов. Для их тестирования не обязательно запускать приложение.

В: Почему мы не создали модель представления для `MainActivity`?

О: В приложении Guessing Game `MainActivity` используется только для отображения фрагментов игры. Так как активность `MainActivity` не включает никакую игровую логику или данные, создавать для нее модель представления не обязательно.

В: Могут ли два фрагмента использовать одну модель представления?

О: Если активность использует два фрагмента, которые должны взаимодействовать друг с другом, можно добавить в активность одну модель представления, которая будет доступной для обоих фрагментов. Дополнительную информацию можно найти по адресу *https://developer.android.com/topic/libraries/architecture/viewmodel*

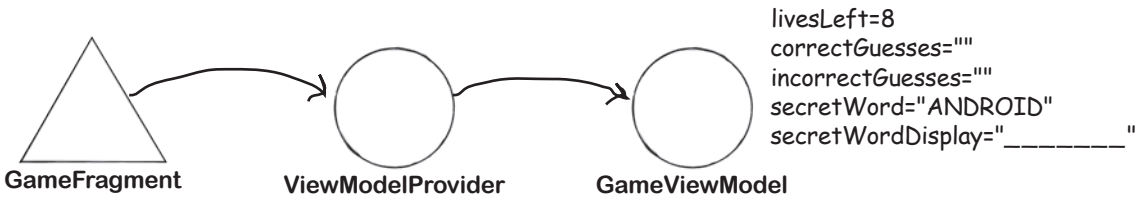


Базовая игра
GameFragment
ResultFragment

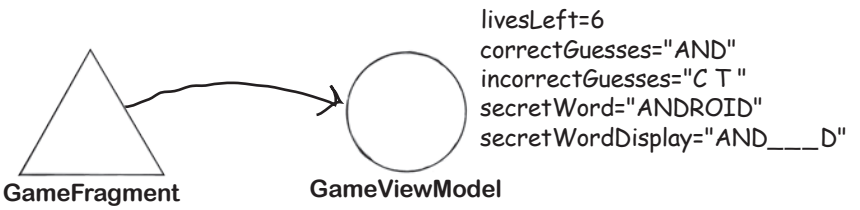
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

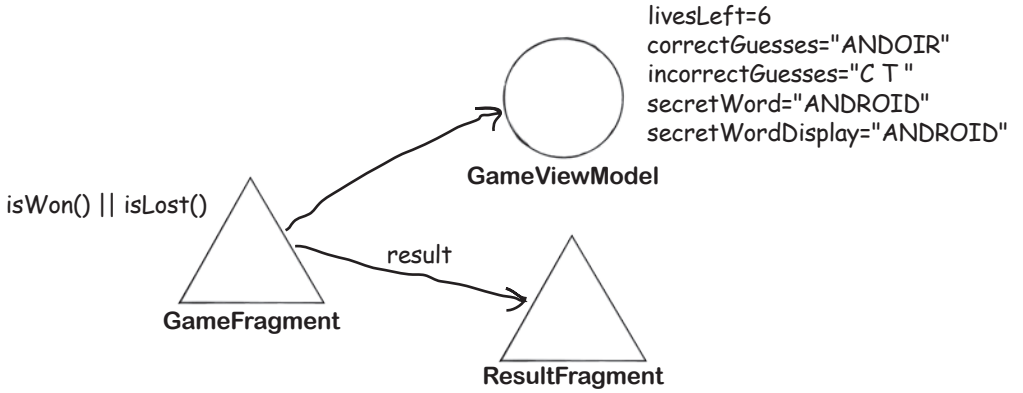
1. **GameFragment запрашивает у класса ViewModelProvider экземпляр GameViewModel.**
Объект `GameViewModel` инициализируется и выбирает случайное слово.

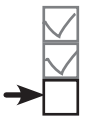


2. **GameFragment взаимодействует с объектом GameViewModel.**
Объект `GameViewModel` сохраняет все предположения, сделанные пользователем, и отслеживает количество оставшихся жизней.



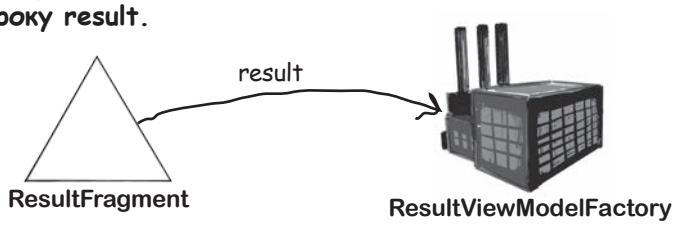
3. **После каждого предположения GameFragment проверяет, возвращает ли true один из методов isWon() или isLost() модели представления.**
Если один из методов возвращает *true*, `GameFragment` переходит к `ResultFragment` с передачей result.



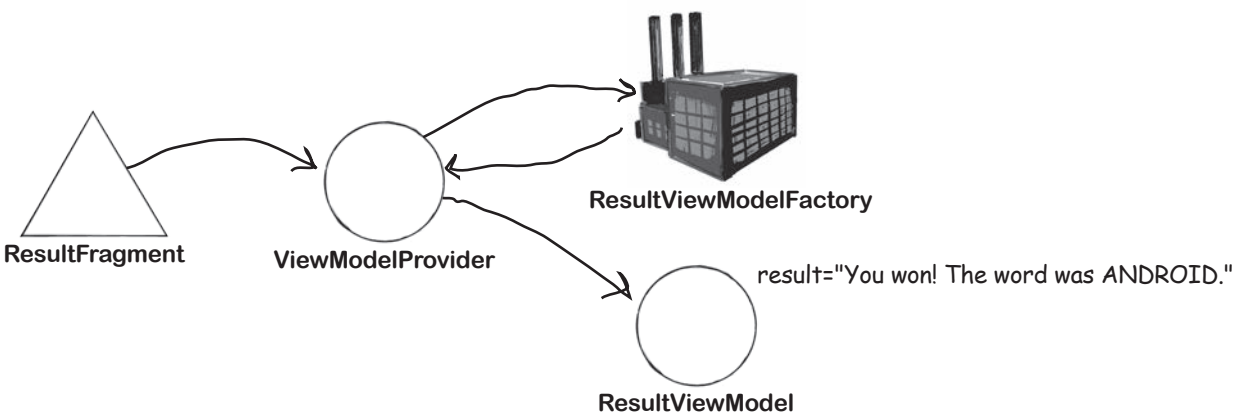


История продолжается

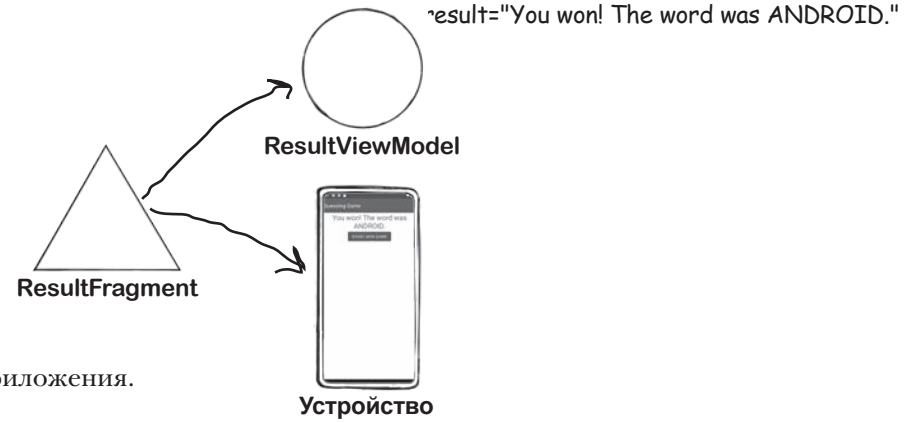
4 **ResultFragment создает объект ResultViewModelFactory и передает ему строку result.**



5 **ResultFragment запрашивает у класса ViewModelProvider экземпляр ResultViewModel.** Класс `ViewModelProvider` видит, что существующего объекта `ResultViewModel` нет, поэтому он создает его при помощи `ResultViewModelFactory`. Свойство `result` объекта `ResultViewModel` инициализируется строкой result.



6 **ResultFragment получает строку result из объекта ResultViewModel, и выводит ее на экран.**



Проведем тест-драйв приложения.

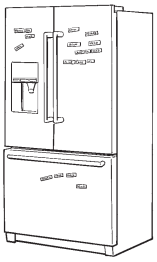
Тест-драйв

- [x] Базовая игра
- [x] GameFragment
- [x] **ResultFragment**

При запуске приложения, как и прежде, отображается `GameFragment`.

Если пользователь угадал все буквы или потратил все жизни, приложение переходит к `ResultFragment`. Выводится сообщение с информацией о том, выиграл или проиграл пользователь и какое слово было загадано.

На первый взгляд игра работает точно так же, как и прежде, но теперь для игровой логики и данных в ее новой версии используются модели представлений.



Развлечения с магнитами

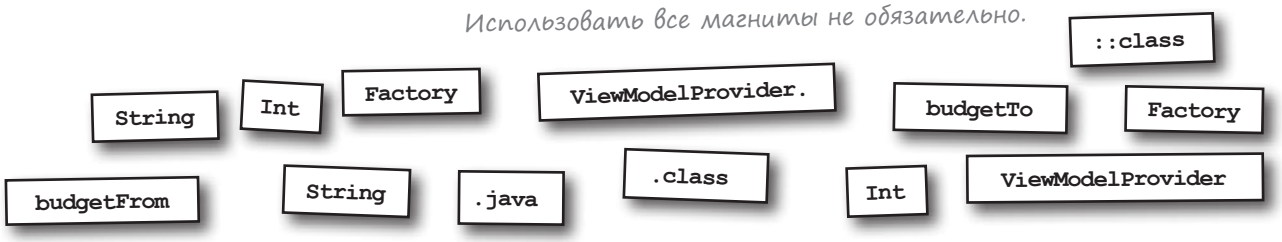
Ниже приведен код определения модели представления с именем `GiftViewModel`:

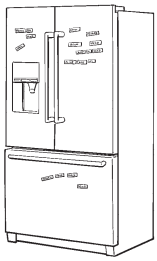
```
import androidx.lifecycle.ViewModel

class GiftViewModel(budgetFrom: Int, budgetTo: Int) : ViewModel(){
    val from = budgetFrom
    val to = budgetTo
}
```

Удастся ли вам восстановить код класса фабрики модели представления, которая будет использоваться для создания объектов `GiftViewModel`?

```
import androidx.lifecycle.ViewModel

import androidx.lifecycle.  ..........
import java.lang.IllegalArgumentException

class GiftViewModelFactory(private val budgetFrom: ..........., private val budgetTo: ..........)

                           :  ..........{.....

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {

        if (modelClass.isAssignableFrom(GiftViewModel ..........  ))

            return GiftViewModel(..........., ..........) as T

        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Ниже приведен код определения модели представления с именем `GiftViewModel`:

```
import androidx.lifecycle.ViewModel

class GiftViewModel(budgetFrom: Int, budgetTo: Int) : ViewModel(){
    val from = budgetFrom
    val to = budgetTo
}
```

Удастся ли вам восстановить код класса фабрики модели представления, которая будет использоваться для создания объектов `GiftViewModel`?

```
import androidx.lifecycle.ViewModel

import androidx.lifecycle.  [ViewModelProvider]
import java.lang.IllegalArgumentException

class GiftViewModelFactory(private val budgetFrom: [Int], private val budgetTo: [Int])
    : [ViewModelProvider.][Factory]{

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {

        if (modelClass.isAssignableFrom(GiftViewModel[::class][.java])){

            return GiftViewModel([budgetFrom][budgetTo]) as T

        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Импортируем ViewModelProvider.

Для создания GiftViewModel фабрике необходимы два значения Int.

Класс должен расширять ViewModelProvider.Factory.

isAssignableFrom() требует ссылки на класс GiftViewModel.

Конструирует объект GiftViewModel.

[String] [Factory] [String] [.class]

Эти магниты остались неиспользованными.

СТАНЬ моделью представления. Решение

Приведенный ниже код описывает класс модели представления с именем MyViewModel. Представьте себя на месте модели представления и скажите, какие проблемы встречаются в этом коде и как их исправить.

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log
import android.widget.TextView

class MyViewModel : ViewModel() {
    val num = 2

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * num
    }

    fun joinTogether(text1: TextView String, text2: TextView String): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1.text} ${text2.text}")
    }
}
```

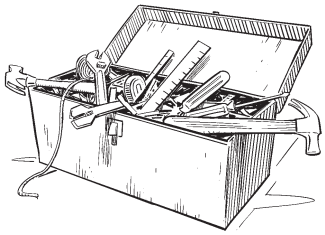
Модели представлений не должны обращаться к каким-либо представлениям, поэтому эта команда импортирования не нужна.

Класс должен расширять ViewModel.

Модели представлений не должны обращаться к каким-либо представлениям, поэтому этот метод переводится на использование значений String.

Ваш инструментарий Android

ГЛАВА 11



Глава 11 осталась позади, а ваш инструментарий пополнился моделями представлений.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Модель представления упрощает код активности или фрагмента за счет отделения кода, относящегося к бизнес-логике и данным.
- Модели представлений могут нормально переносить изменения конфигурации.
- Модель представления обычно связывается с отдельным UI-контроллером.
- Модель представления расширяет класс `androidx.lifecycle.ViewModel`.
- Модель представления создается вызовом метода `get()` класса `ViewModelProvider`. Это гарантирует, что модель представления будет создаваться только в том случае, если она еще не существует.
- Модель представления продолжает существовать до тех пор, пока UI-контроллер не будет уничтожен навсегда. Это происходит при завершении активности или удаления фрагмента из его активности.
- Если конструктору модели представления необходимы аргументы, вам придется написать для него дополнительный класс — фабрику. Класс `ViewModelProvider` использует фабрику каждый раз, когда ему потребуется создать экземпляр модели представления.
- Класс фабрики модели представления должен реализовать интерфейс с именем `ViewModelProvider.Factory`.

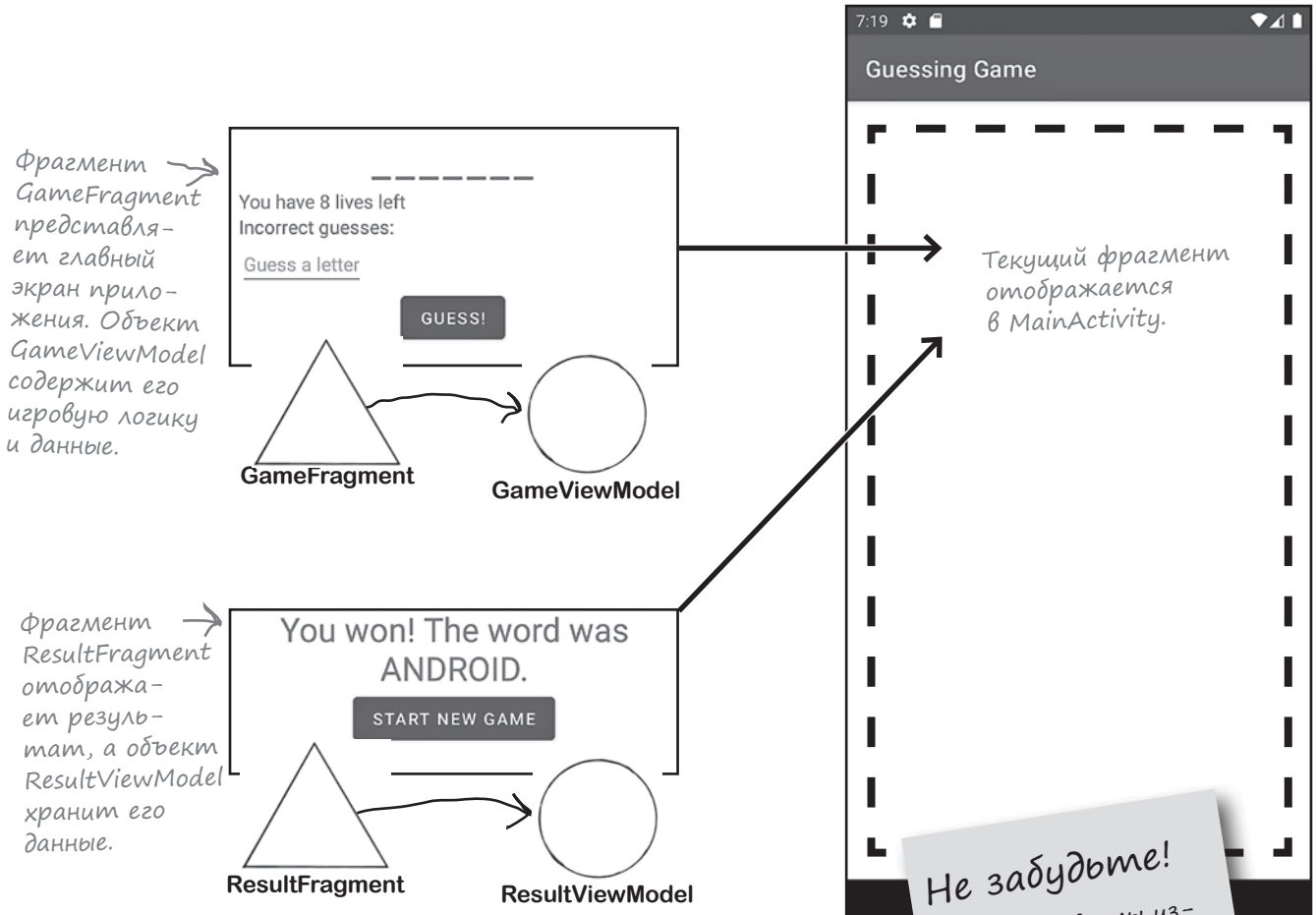
В самой гуще событий

Вашему коду часто приходится реагировать на изменения значений свойств. Например, если свойство модели представления меняет значение, **фрагмент может отреагировать** обновлением своих представлений или переходом. Но **как фрагмент узнает о том, что свойство было обновлено**? В этой главе вы узнаете о **Live Data**: механизме **оповещения заинтересованных сторон о каких-либо изменениях**. Вы узнаете о ***MutableLiveData*** и о том, как **заставить ваш фрагмент наблюдать за изменениями свойств этого типа. Мы покажем, как тип *LiveData*** помогает обеспечить целостность данных вашего приложения. Вскоре вы будете писать приложения, которые реагируют на происходящее быстрее, чем когда-либо.

Снова о приложении Guessing Game

В предыдущей главе мы построили приложение Guessing Game, в котором пользователь пытается отгадать буквы, входящие в загаданное слово. Когда пользователь отгадывает все буквы или у него кончаются жизни, игра завершается.

Чтобы код фрагмента не разрастался, а состояние приложения не терялось при повороте экрана устройства, мы использовали модели представлений для игровой логики и данных приложения. `GameFragment` использует `GameViewModel` для своей логики и данных, а `ResultViewModel` хранит результат игры, необходимый для `ResultFragment`:



Когда каждый фрагмент отображается на экране или пользователь вводит предположение, фрагменты получают новейшие значения от модели представления и отображают их на экране.

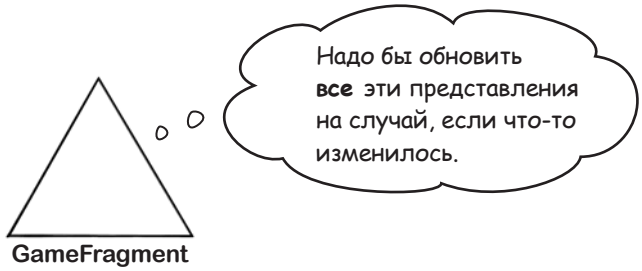
Такое решение работает, но у него есть ряд недостатков.

Не забудьте!

В этой главе мы изменим приложение Guessing Game, созданное в главе 11. Откройте проект этого приложения.

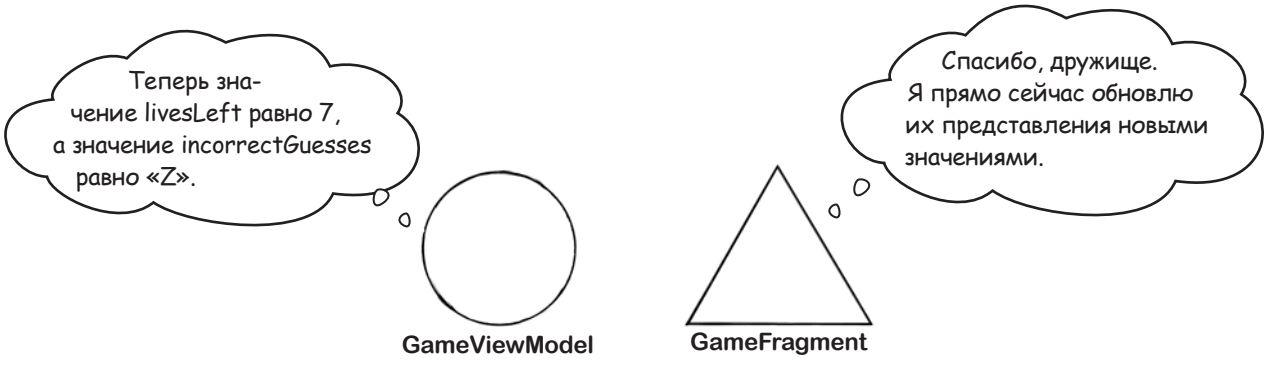
Фрагменты решают, когда обновлять представления

Недостаток такого решения состоит в том, что каждый фрагмент решает, когда следует получить новейшие значения свойств из модели представления и обновить ее представления. Однако эти значения изменяются не всегда. Например, если пользователь вводит правильное предположение, `GameFragment` обновляет выводимый текст с количеством оставшихся жизней и ошибочных предположений, несмотря на то что эти значения не изменялись.



Пусть модель представления сама сообщает об изменении значений

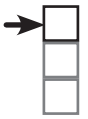
Возможно альтернативное решение: пусть `GameViewModel` сообщает `GameFragment` об изменении каждого из своих свойств. Если фрагмент будет оповещаться об этих изменениях, он уже не сможет самостоятельно решать, когда следует получить новейшие значения свойств от модели представления и обновить ее представления. Вместо этого представления будут обновляться только тогда, когда фрагмент получит оповещения об изменении используемых свойств.



Мы реализуем это изменение в приложении Guessing Game при помощи библиотеки Android **Live Data**, которая является частью Android Jetpack. Live Data позволяет модели представления оповещать заинтересованные стороны (например, фрагменты и активности) об обновлении значений свойств. Они могут отреагировать на изменения обновлением представлений или вызовом других методов.

В оставшейся части этой главы вы научитесь использовать Live Data. Но сначала рассмотрим основные этапы обновления приложения.

Библиотека Live Data является частью Android Jetpack.

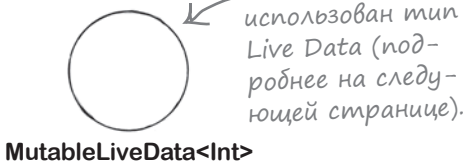


Использование Live Data
Защита свойств
Добавление gameOver

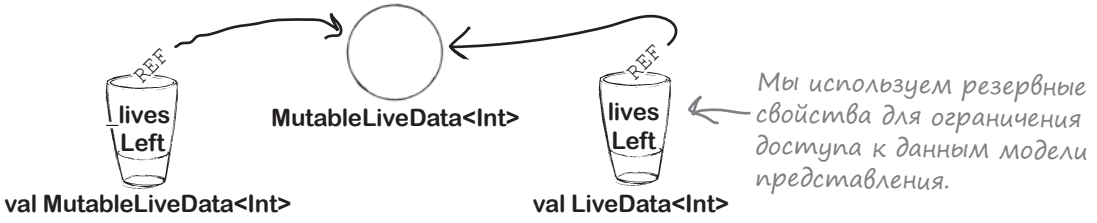
Что нам предстоит сделать

Основные этапы построения приложения:

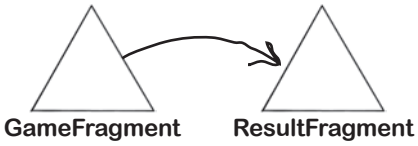
1. **Перевод приложения Guessing Game на использование Live Data.**
Мы обновим `GameViewModel`, чтобы для свойств `livesLeft`, `incorrectGuesses` и `secretWordDisplay` использовался механизм Live Data. Затем мы позаботимся о том, чтобы фрагмент `GameFragment` обновлял свои представления при изменении значений этих свойств.



2. **Защита свойств и методов GameViewModel.**
Мы ограничим доступ к свойствам `GameViewModel`, чтобы они могли обновляться только из `GameViewModel`. Также проследим за тем, чтобы фрагменту `GameFragment` были доступны только методы, необходимые для выполнения его работы.



3. **Добавление свойства gameOver.**
Для принятия решения о том, завершена ли игра, класс `GameViewModel` будет использовать новое свойство `gameOver`. `GameFragment` будет переходить к `ResultFragment` при изменении значения этого свойства.



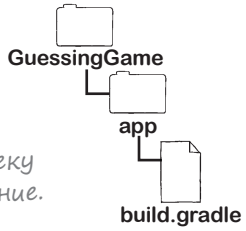
Добавление зависимости для Live Data в файл build.gradle приложения

Так как мы собираемся использовать библиотеку Live Data, начнем с добавления зависимости Live Data в файл *build.gradle* приложения.

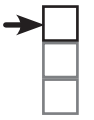
Откройте проект приложения Guessing Game (если это не было сделано ранее), откройте файл *GuessingGame/app/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `dependencies`:

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.3.1'
    ...
}
```

Добавляет библиотеку Live Data в приложение.



Синхронизируйте изменения, когда среда предложит вам это сделать.



Использование Live Data
Защита свойств
Добавление gameOver

GameViewModel и GameFragment должны использовать Live Data

Мы хотим использовать Live Data в приложении Guessing Game, чтобы объект `GameViewModel` оповещал `GameFragment` об изменении значений свойств. Тогда `GameFragment` сможет реагировать на эти изменения. Эта задача будет решаться в два этапа:

1. **Определение изменений свойств GameViewModel, о которых должен знать GameFragment.**
2. **Определение того, как фрагмент GameFragment должен реагировать на каждое изменение.**

Пока сосредоточимся на изменениях кода `GameViewModel`.

Какие свойства модели представления должны использовать Live Data?

`GameViewModel` включает три свойства: `secretWordDisplay`, `incorrectGuesses` и `livesLeft`, которые `GameFragment` использует для обновления своих представлений. Мы укажем, что эти три свойства используют Live Data, чтобы `GameFragment` оповещался об изменении значений их свойств.

Чтобы указать, что свойство использует Live Data, следует изменить его тип на **`MutableLiveData<Type>`**, где `Type` — тип данных, которые должны храниться в свойстве. Например, свойство `livesLeft` в настоящее время определяется в коде с типом `Int`:

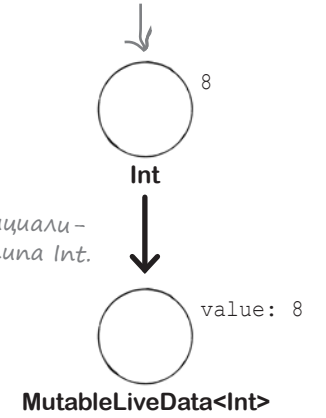
```
var livesLeft = 8
```

Чтобы свойство использовало механизм Live Data, замените его тип на `MutableLiveData<Int>`, чтобы оно выглядело так:

```
var livesLeft = MutableLiveData<Int>(8)
```

MutableLiveData<Int> инициализируется значением 8 типа Int.

Свойство livesLeft преобразуется из Int в MutableLiveData<Int>.



Эта запись означает, что `livesLeft` теперь относится к типу `MutableLiveData<Int>` и имеет исходное значение 8.

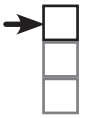
Аналогичным образом можно определить свойства `incorrectGuesses` и `secretWordDisplay`, для чего используется код следующего вида:

```
var incorrectGuesses = MutableLiveData<String>("")
var secretWordDisplay = MutableLiveData<String>()
```

Значение secretWordDisplay здесь не присваивается, потому что мы сделаем это в init-блоке модели представления.

Здесь каждому свойству назначается тип `MutableLiveData<String>`. Свойству `incorrectGuesses` присваивается значение `""`, тогда как значение `secretWordDisplay` будет задаваться в init-блоке `GameViewModel`.

Так определяются свойства Live Data. На следующем шаге мы займемся обновлением их значений.



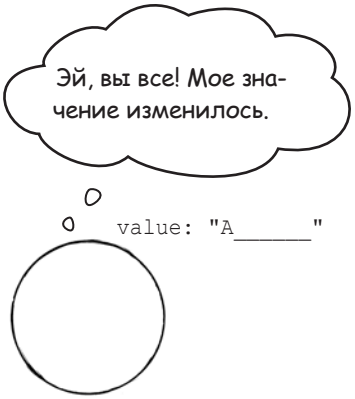
Объекты Live Data используют свойство value

При использовании свойств `MutableLiveData` вы обновляете их значения при помощи свойства с именем **`value`**. Например, для обновления свойства `secretWordDisplay` возвращаемым значением метода `deriveSecretWordDisplay()` не используется код следующего вида:

```
secretWordDisplay = deriveSecretWordDisplay()
```

как это делалось ранее. Вместо этого используется следующий код:

```
secretWordDisplay.value = deriveSecretWordDisplay()
```



Изменение значения свойства именно таким способом важно, потому что при этом заинтересованные стороны — в данном случае `GameFragment` — оповещаются о любых изменениях. Каждый раз, когда значение свойства `value` у `secretWordDisplay` обновляется, `GameFragment` оповещается об этих изменениях и может отреагировать обновлением своих представлений.

Свойство value может содержать null

При использовании Live Data необходимо учитывать еще одно обстоятельство: тип значения *допускает null*. Это означает, что при использовании значений Live Data в коде необходимо выполнять проверки null-безопасности, в противном случае ваш код компилироваться не будет.

Например, свойство `livesLeft` определяется следующим кодом:

```
var livesLeft = MutableLiveData<Int>(8)
```

Означает, что свойству livesLeft может присваиваться Int или null.

Свойство имеет тип `MutableLiveData<Int>`, поэтому его свойство `value` может принимать значение `Int` или содержать `null`.

Так как свойство `value` может содержать `null`, для уменьшения его на 1 не удастся использовать следующий код:

```
livesLeft.value--
```

Вместо этого приходится использовать команду:

```
livesLeft.value = livesLeft.value?.minus(1)
```

Уменьшает value на 1 при условии, что значение отлично от null.

которая вычитает 1 из value при условии, что значение отлично от `null`.

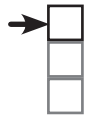
Аналогичным образом следующий метод `isLost()` не компилируется, потому что `liveLeft.value` может содержать `null`:

```
fun isLost() = livesLeft.value <= 0
```

Однако его можно изменить с использованием «элвис-оператора» языка Kotlin:

```
fun isLost() = livesLeft.value ?: 0 <= 0
```

livesLeft.value ?: ноль возвращает ноль, если value содержит null. Здесь isLost() возвращает true, если value содержит null или <= 0.



Использование Live Data
Защита свойств
Добавление gameOver

Я тут подумала... Вы говорите, что каждый раз, когда потребуется обновить свойство Live Data, новое значение присваивается свойству value существующего объекта. Сам объект Live Data при этом не заменяется. Означает ли это, что свойства Live Data можно определять с val вместо var?

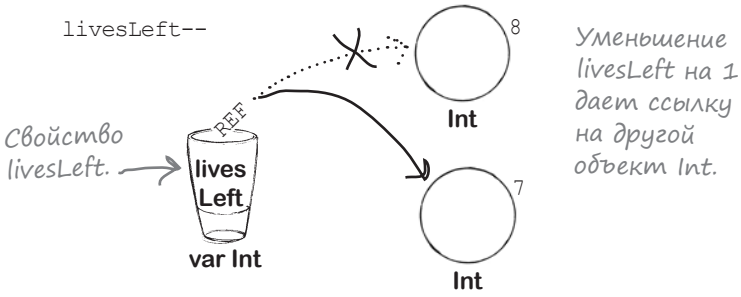
Свойства Live Data могут определяться с val.

Как вы уже знаете, ключевые слова `val` и `var` используются в Kotlin для определения того, возможно ли для свойства присваивание ссылки на новый объект.

Когда мы изначально определяли каждое свойство, мы использовали `var`, чтобы его можно было обновить. Например, для свойства `livesLeft` используется следующее определение:

```
var livesLeft = 8
```

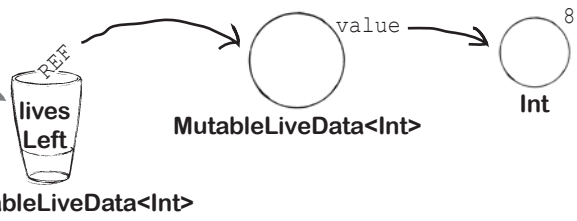
которое инициализирует `livesLeft` объектом `Int` со значением 8. Каждый раз, когда пользователь делает ошибочное предположение, мы уменьшаем `livesLeft` на 1, что дает ссылку на новый объект `Int`:

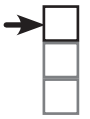


С Live Data вы обновляете свойство `value` *существующего* объекта, вместо того чтобы заменять его другим объектом, и заинтересованные стороны оповещаются об этом изменении. Так как объект более не заменяется, свойство можно определить с `val` вместо `var`, как в следующем примере:

```
val livesLeft = MutableLiveData<Int>(8)
```

Здесь уже не нужно предоставлять livesLeft ссылку на новый объект MutableLiveData; мы просто обновляем его свойство value. А следовательно, его можно определить с val вместо var.



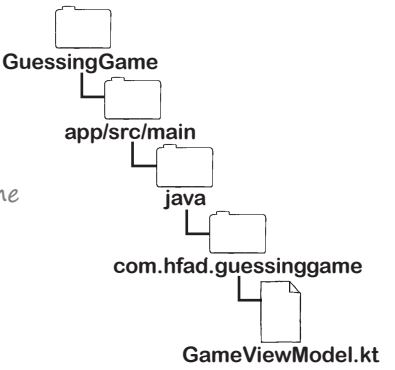


Использование Live Data
Защита свойств
Add gameOver

Полный код GameViewModel.kt

Теперь вы знаете, как работает Live Data. Давайте обновим класс `GameViewModel`, чтобы все свойства `livesLeft`, `incorrectGuesses` и `secretWordDisplay` использовали Live Data.

Ниже приведен полный код *GameViewModel.kt*; обновите свою версию кода (изменения выделены жирным шрифтом):




```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData

class GameViewModel : ViewModel() {
    val words = listOf("Android", "Activity", "Fragment")
    val secretWord = words.random().uppercase()
    var secretWordDisplay = MutableLiveData<String>()
    var correctGuesses = ""
    var incorrectGuesses = MutableLiveData<String>("")
    var livesLeft = MutableLiveData<Int>(8)

    init {
        secretWordDisplay.value = deriveSecretWordDisplay()
    }

    fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }

    fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }
```

Импортируйте этот класс.

Свойства Live Data определяются с val вместо var.

Измените типы этих свойств, чтобы они использовали Live Data.

Необходимо использовать свойство value свойства secretWordDisplay.

Продолжение на следующей странице.

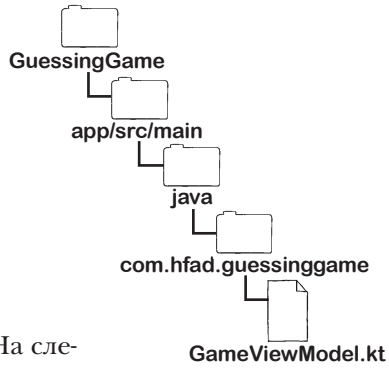
GameViewModel.kt (продолжение)

```
    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                incorrectGuesses.value += "$guess "
                livesLeft--
                livesLeft.value = livesLeft.value?.minus(1)
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

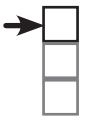
Необходимо использовать свойство value для secretWordDisplay и incorrectGuesses.

Уменьшает значение livesLeft на единицу способом, безопасным по отношению к null.

Если value равно null, «элвис-оператор» возвращает ноль. Это означает, что если значение livesLeft равно null или оно меньше либо равно нулю, isLost() возвращает true.



И это все, что было необходимо сделать для `GameViewModel`. На следующем шаге мы заставим `GameFragment` реагировать на обновление свойств `livesLeft`, `incorrectGuesses` и `secretWordDisplay`.



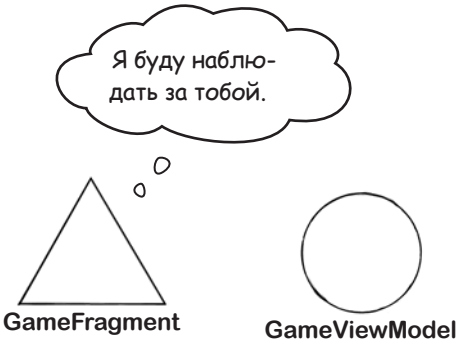
Использование Live Data
Защита свойств
Добавление gameOver

Фрагмент наблюдает за свойствами модели представления и реагирует на изменения

Чтобы фрагмент реагировал на изменения `value` в свойстве `MutableLiveData` модели представления, следует вызвать метод **`observe()`** свойства. Например, если добавить в `GameFragment` следующий код, он будет наблюдать за свойством `livesLeft` модели представления и выполнять заданное действие при его изменении:

Добавление этого кода во фрагмент позволяет ему реагировать на изменения значений свойств.

```
viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
    //Код, использующий новое значение
})
```



Как видите, в этом коде методу `observe()` передаются аргументы `viewLifecycleOwner` и `Observer`.

`viewLifecycleOwner` относится к жизненному циклу представлений фрагмента. Он привязан к промежутку времени, в котором фрагмент имеет доступ к своему пользовательскому интерфейсу: от момента его создания в методе `onCreateView()` фрагмента до его уничтожения и вызова метода `onDestroyView()`.

`Observer` — класс, который может получать данные Live Data. Он связан с `viewLifecycleOwner`, и поэтому он активен (и может получать оповещения Live Data), только когда фрагмент имеет доступ к своим представлениям. Если значение свойства Live Data изменяется, когда фрагмент не имеет доступа к своему пользовательскому интерфейсу, наблюдатель не получит оповещения, а фрагмент не среагирует. Тем самым предотвращаются попытки фрагмента обновить представления в то время, когда они недоступны, что может привести к фатальному сбою приложения.

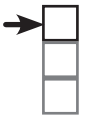
Это также упрощает работу программиста, потому что вам не нужно проверять доступность представления. Механизм Live Data делает все за вас.

Класс `Observer` получает параметр с лямбда-выражением, которое определяет, как должно использоваться новое значение свойства. Например, в приложении Guessing Game фрагмент `GameFragment` должен обновлять текст `lives` при каждом обновлении свойства `livesLeft` модели представления. Для этого можно воспользоваться следующим кодом:

```
viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
    binding.lives.text = "You have $newValue lives left"
})
```

Сообщает пользователю, сколько жизней у него осталось.

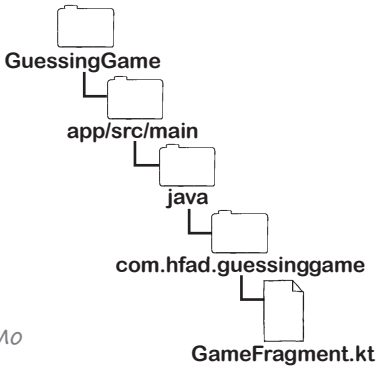
Вот и все, что необходимо знать для того, чтобы фрагмент `GameFragment` обновлял свои представления при изменении значений свойств модели представления. Давайте внесем изменения в код.



Использование Live Data
Защита свойств
Добавление gameOver

Полный код GameFragment.kt

Ниже приведен обновленный код `GameFragment`; включите в файл *GameFragment.kt* изменения, выделенные жирным шрифтом:



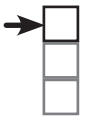

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
        updateScreen()

        viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
            binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
        })

        viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
            binding.lives.text = "You have $newValue lives left"
        })
```

Этот класс необходимо импортировать.

Этот метод теперь не нужен, так как для обновления значений, отображаемых на экране, используется Live Data.

При каждом изменении свойств incorrectGuesses и livesLeft в модели представления происходит обновление текста в макете.

Продолжение на следующей странице.



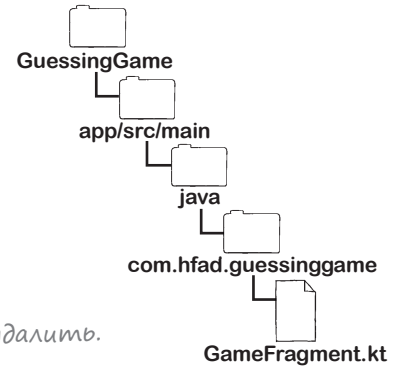
Полный код GameFragment.kt (продолжение)

```
        viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
            binding.word.text = newValue
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            ~~updateScreen()~~
            if (viewModel.isWon() || viewModel.isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    ~~fun updateScreen() {~~
        ~~binding.word.text = viewModel.secretWordDisplay~~
        ~~binding.lives.text = "You have ${viewModel.livesLeft} lives left"~~
        ~~binding.incorrectGuesses.text = "Incorrect guesses:${viewModel.incorrectGuesses}"~~
    ~~}~~
}
```

Обновляет текст представления загаданного слова при обновлении свойства secretWordDisplay в модели представления.

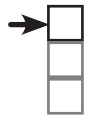
Удаляет эту строку, так как мы используем Live Data для обновления экрана.

Этот метод не нужен, его можно удалить.



И это все изменения, которые необходимо внести в `GameFragment`, чтобы представления обновлялись каждый раз, когда будут обнаружены изменения в свойствах `incorrectGuesses`, `livesLeft` и `secretWordDisplay`.

Мы реализовали поддержку Live Data в приложении Guessing Game. Давайте посмотрим, что происходит во время выполнения приложения.



Модели ResultViewModel не нужно использовать Live Data, поэтому и обновлять ее не придется.

Как вы помните, `ResultViewModel` содержит одно свойство (с именем `result`), значение которого задается при создании модели представления. Код выглядит примерно так:

```
class ResultViewModel(finalResult: String) : ViewModel(){
    val result = finalResult
}
```

Как видите, `result` определяется с ключевым словом `val`, поэтому после инициализации обновить ее другим значением не удастся. `ResultFragment` не нуждается в оповещениях об изменениях `result`, потому что **после присваивания `result` не может измениться**. Заставлять `ResultFragment` реагировать на изменения не нужно, потому что изменений быть не может.

Разберемся, что происходит при выполнении кода, затем проведем тест-драйв приложения.

Часто Задаваемые Вопросы

В: Что произойдет, если фрагмент `GameFragment` не будет активен при обновлении одного из свойств Live Data объекта `GameViewModel`? Не приведет ли это к фатальному сбою приложения?

О: Нет. Механизм Live Data принимает во внимание жизненный цикл, поэтому при изменении значений свойств оповещаются только наблюдатели с активным жизненным циклом. Если фрагмент `GameFragment` неактивен при обновлении свойств `GameViewModel`, то `GameFragment` не будет оповещаться и вам не придется беспокоиться о сбое приложения.

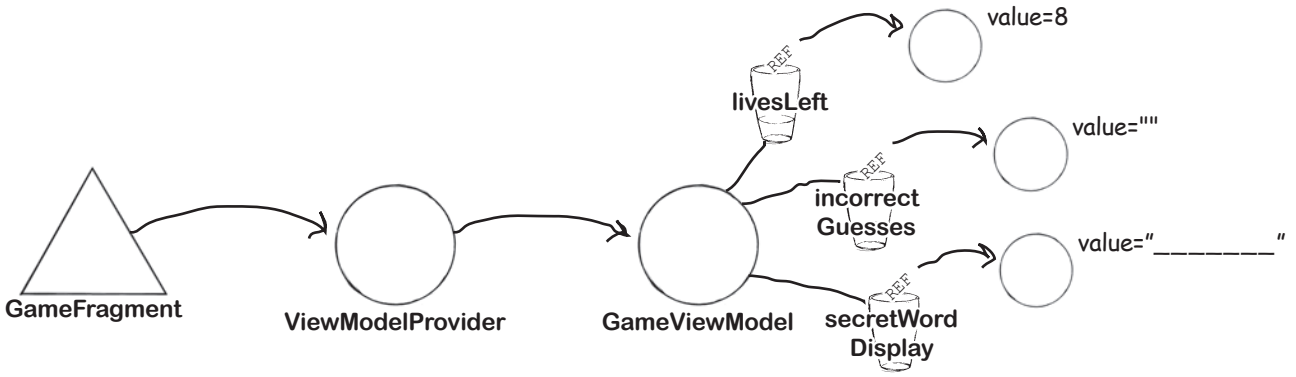
В: Мы знаем тип `MutableLiveData`. Существует ли тип с именем `LiveData`?

О: Да! Тип `LiveData` очень похож на `MutableLiveData`, не считая того, что его свойство `value` обновляться не может. О том, как используется тип `LiveData`, вы узнаете через несколько страниц.

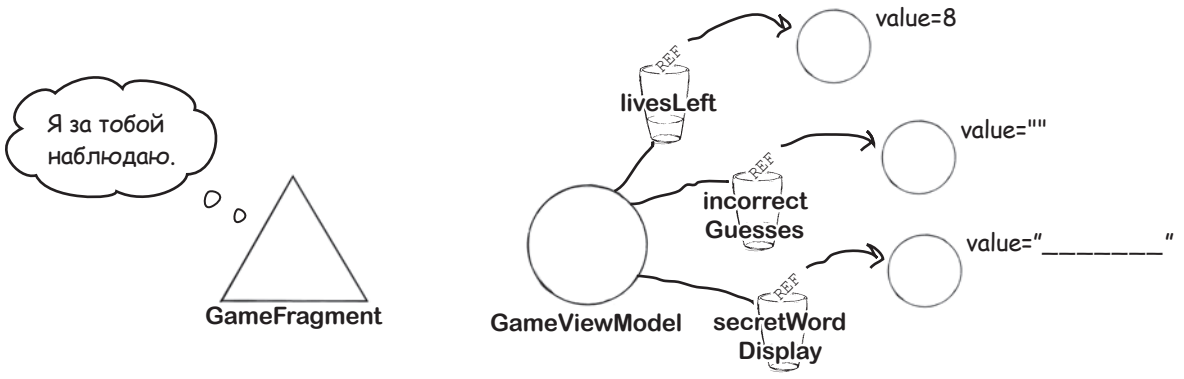
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

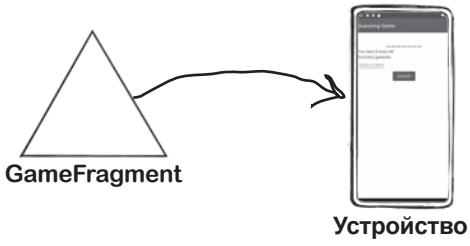
1 **GameFragment запрашивает у класса ViewModelProvider экземпляр GameViewModel.** Объект `GameViewModel` инициализируется, и задаются значения трех свойств `MutableLiveData` — `livesLeft`, `incorrectGuesses` и `secretWordDisplay`.



2 **GameFragment наблюдает за свойствами livesLeft, incorrectGuesses и secretWordDisplay объекта GameViewModel.**

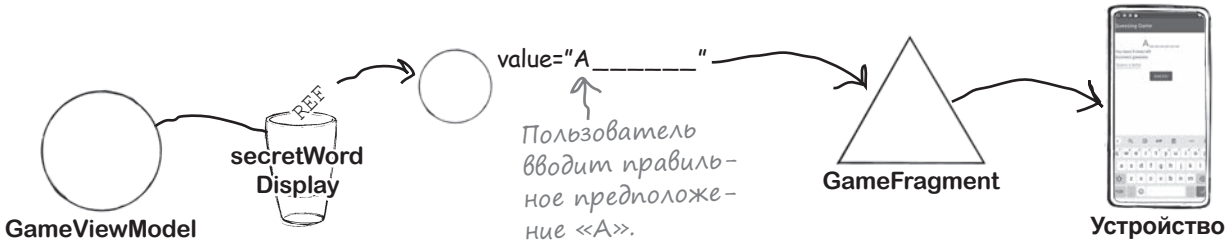


3 **GameFragment обновляет свои представления значениями свойств, за которыми он наблюдает.**

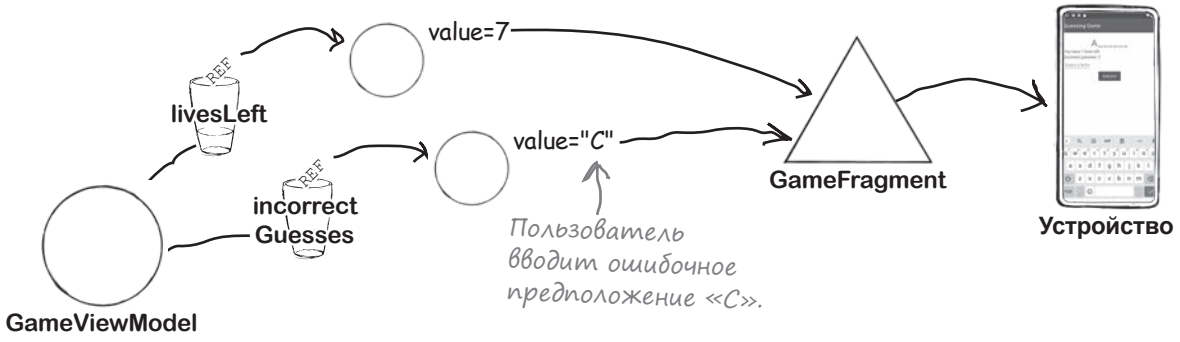


История продолжается

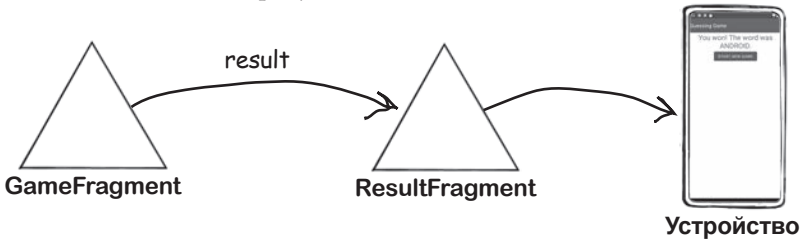
4 **Когда пользователь вводит правильную букву, значение secretWordDisplay обновляется и новое значение передается GameFragment.**
`GameFragment` реагирует обновлением экранного представления загаданного слова.



5 **Когда пользователь вводит ошибочное предположение, значения incorrectGuesses и livesLeft обновляются и передаются GameFragment.**
`GameFragment` реагирует обновлением своих представлений.

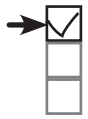


6 **Когда isWon() или isLost() возвращает true, GameFragment переходит к ResultFragment и передает result.**
`ResultFragment` выводит результат.



Проведем тест-драйв приложения.

Тест-драйв

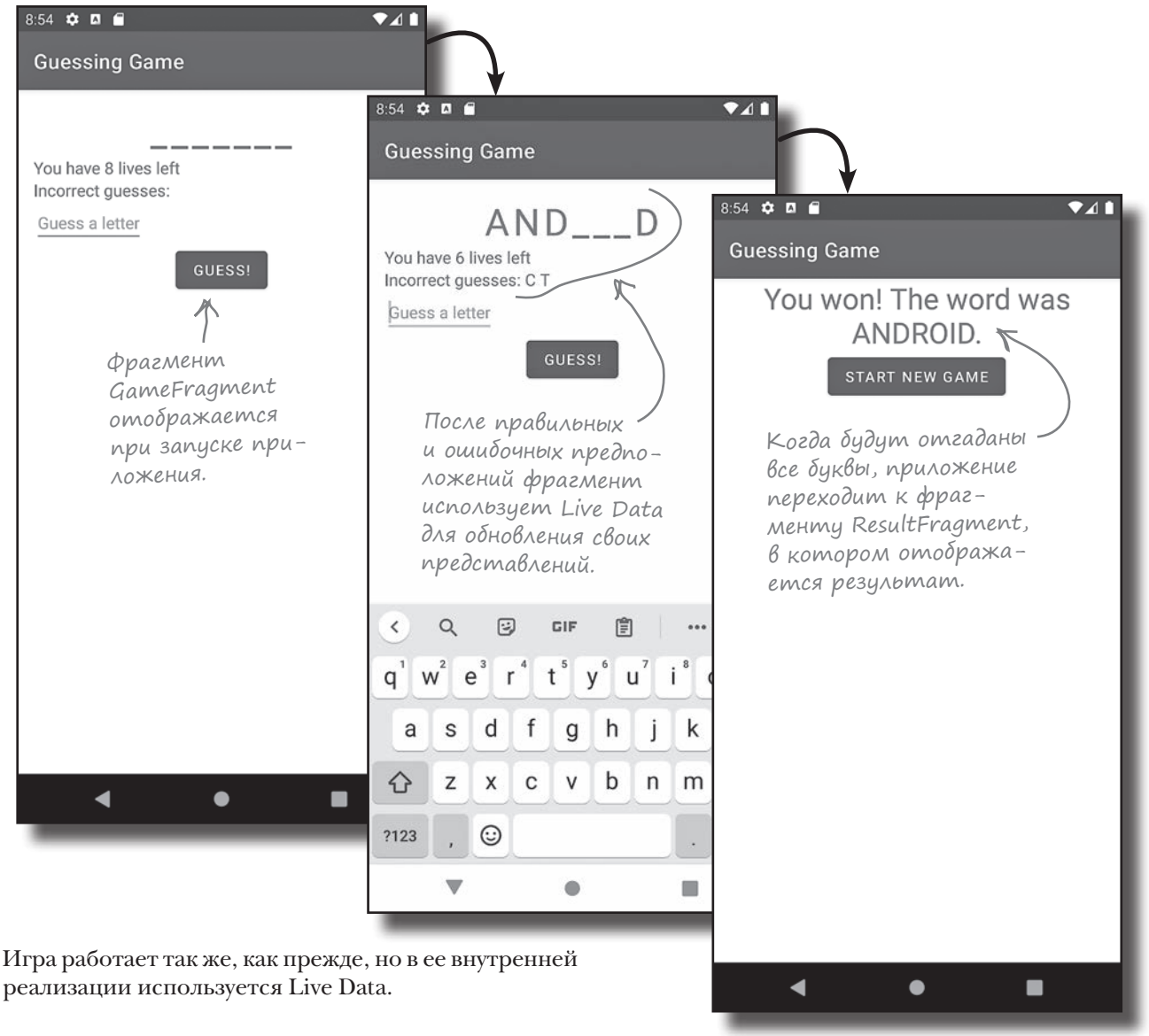


Использование Live Data
Защита свойств
Добавление gameOver

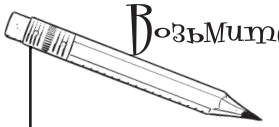
При запуске приложения, как и прежде, отображается фрагмент `GameFragment`.

При вводе правильного предположения экранное представление загаданного слова обновляется. При вводе ошибочного предположения обновляется количество оставшихся жизней, а предположение включается в выводимое количество неправильных предположений.

Если пользователь правильно угадал все буквы или потерял все жизни, приложение переходит к фрагменту `ResultFragment`, который отображает результат.



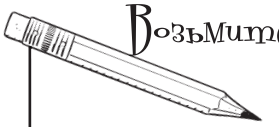
Игра работает так же, как прежде, но в ее внутренней реализации используется Live Data.



Возьмите в руку карандаш

Следующий код описывает модель представления с именем `MyViewModel`. Удастся ли вам изменить код так, чтобы он использовал Live Data для реагирования фрагментов на изменение свойства `rate`?

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log

class MyViewModel : ViewModel() {
    var rate = 2

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun updateRate(newRate: Int) {
        rate = newRate
    }

    fun Calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * rate
    }

    fun JoinTogether(text1: String, text2: String): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1} ${text2}")
    }
}
```



Возьмите в руку карандаш
Решение

Следующий код описывает модель представления с именем `MyViewModel`. Удастся ли вам изменить код так, чтобы он использовал Live Data для реагирования фрагментов на изменение свойства `rate`?

```
package com.hfad.myapp

import androidx.lifecycle.ViewModel
import android.util.Log
import androidx.lifecycle.MutableLiveData   <- Этот класс необходимо импортировать.

class MyViewModel : ViewModel() {
    val ~~var~~ rate = MutableLiveData<Int>(2)   <- Свойство rate должно использовать val вместо var и относиться к типу MutableLiveData<Int>.

    init {
        Log.i("MyViewModel", "ViewModel created")
    }

    override fun onCleared() {
        Log.i("MyViewModel", "ViewModel cleared")
    }

    fun updateRate(newRate: Int) {
        rate.value = newRate   <- Необходимо правильно присвоить значение rate.
    }

    fun Calculation(val1: Int, val2: Int): Int {
        Log.i("MyViewModel", "Called Calculation")
        return (val1 + val2) * rate.value ?: 0   <- Необходимо правильно использовать значение rate с «элвис-оператором» на случай, если value содержит null.
    }

    fun JoinTogether(text1: String, text2: String): String {
        Log.i("MyViewModel", "Called JoinTogether")
        return ("${text1} ${text2}")
    }
}
```

Фрагменты могут обновлять свойства GameViewModel

До настоящего момента мы обновили коды GameViewModel и GameFragment, чтобы в них использовался механизм Live Data. Кажый раз, когда значение свойства MutableLiveData из GameViewModel обновляется, GameFragment реагирует обновлением своих представлений.

Тем не менее в этом коде кроется небольшая проблема. GameFragment обладает полным доступом к свойствам и методам GameViewModel, поэтому при желании фрагмент может использовать их некорректно. Ничто не помешает фрагменту, скажем, присвоить свойству livesLeft значение 100, чтобы у пользователя было намного больше предположений и он легко выигрывал каждую партию.

Для предотвращения этой проблемы мы ограничим прямой доступ к свойствам GameViewModel, чтобы они **могли обновляться только методами модели представления**.

Чтобы закрыть прямой доступ к свойству модели представления, можно объявить его с модификатором private и предоставить возможность обращения только для чтения через get-метод.

Приватные свойства

Чтобы защитить свойства GameViewModel, мы пометим их ключевым словом private, чтобы их значения могли обновляться только кодом GameViewModel. После этого мы предоставим доступ к версии, *доступной только для чтения*, каждого свойства MutableLiveData, за которым должен наблюдать GameFragment. Вместо того чтобы определять свойство livesLeft кодом следующего вида:

```
val livesLeft = MutableLiveData<Int>(8)
```

мы будем использовать следующую запись:

```
private val _livesLeft = MutableLiveData<Int>(8)
val livesLeft: LiveData<Int>
    get() = _livesLeft
```

Свойство MutableLiveData помечено ключевым словом private, чтобы другие классы не могли обращаться к нему.

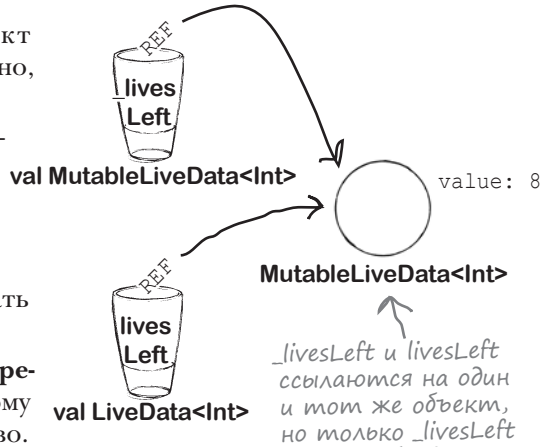
Возвращает версию свойства MutableLiveData, доступную только для чтения.

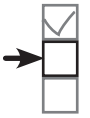
Свойство _livesLeft содержит ссылку на объект MutableLiveData. Для GameFragment это свойство недоступно, потому что оно помечено ключевым словом private.

Однако GameFragment может обратиться к значению этого свойства через метод чтения livesLeft. Свойство livesLeft относится к типу **LiveData**, который имеет много общего с MutableLiveData, за исключением того, что оно не может использоваться для обновления свойства value используемого объекта: GamesFragment может читать значение, но не может обновить его.

При такой структуре кода private-свойство иногда называется **резервным** (backing). Оно содержит ссылку на объект, к которому другие классы могут обращаться только через другое свойство.

Обновим код GameViewModel.



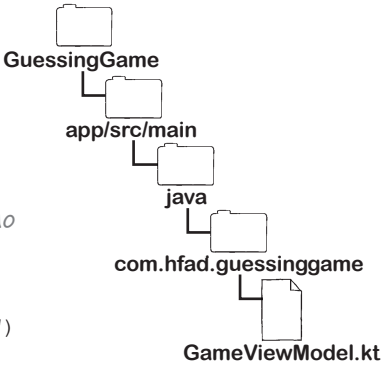


Использование Live Data
Защита свойств
Добавление gameOver

Полный код GameViewModel.kt

Обновим код `GameViewModel`, чтобы в нем использовались резервные свойства для ограничения прямого доступа к свойствам Live Data. Также мы пометим ключевым словом private все свойства и методы, которые не должны использоваться `GameFragment`.

Ниже приведен полный код *GameViewModel.kt*; обновите свою версию кода (изменения выделены жирным шрифтом):




```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    private val words = listOf("Android", "Activity", "Fragment")
    private val secretWord = words.random().uppercase()
    private val _secretWordDisplay = MutableLiveData<String>()
    val secretWordDisplay: LiveData<String>
        get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
        get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
        get() = _livesLeft

    init {
        _secretWordDisplay.value = deriveSecretWordDisplay()
    }
```

Этот класс необходимо импортировать.

Помечается private.

secretWordDisplay предоставляет доступ только для чтения к резервному свойству _secretWordDisplay.

Должны иметь пометку private.

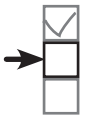
incorrectGuesses предоставляет доступ только для чтения к своему резервному свойству _incorrectGuesses.

Тоже private.

livesLeft возвращает версию резервного свойства _livesLeft, доступную только для чтения.

Мы обновляем свойство value, поэтому необходимо использовать _secretWordDisplay вместо secretWordDisplay.

Продолжение на следующей странице.



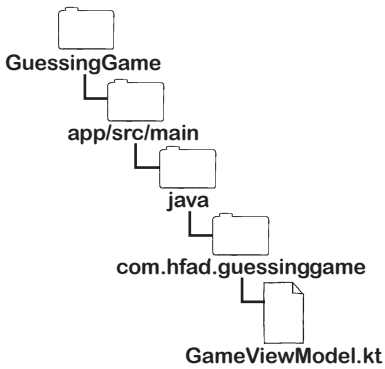
GameViewModel.kt (продолжение)

Эти методы должны вызываться только из GameViewModel, поэтому они помечаются ключевым словом private.

```
    private fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }

    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                _secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                _incorrectGuesses.value += "$guess "
                _livesLeft.value = _livesLeft.value?.minus(1)
            }
        }
    }

    fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

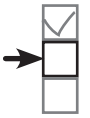
Необходимо использовать изменяемые версии этих свойств, поэтому префиксы «_».

Также не забудьте добавить «_».



Посмотрим, что происходит при выполнении кода.

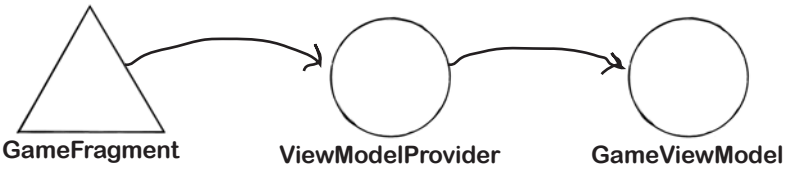
Что происходит при выполнении приложения



Использование Live Data
Protect properties
Добавление gameOver

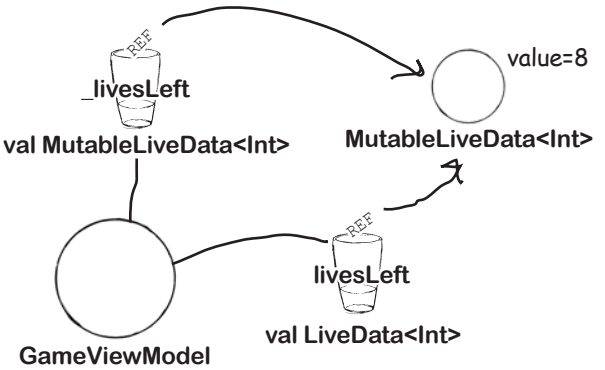
Во время выполнения приложения происходят следующие события:

1 **GameFragment запрашивает у класса ViewModelProvider экземпляр GameViewModel.**

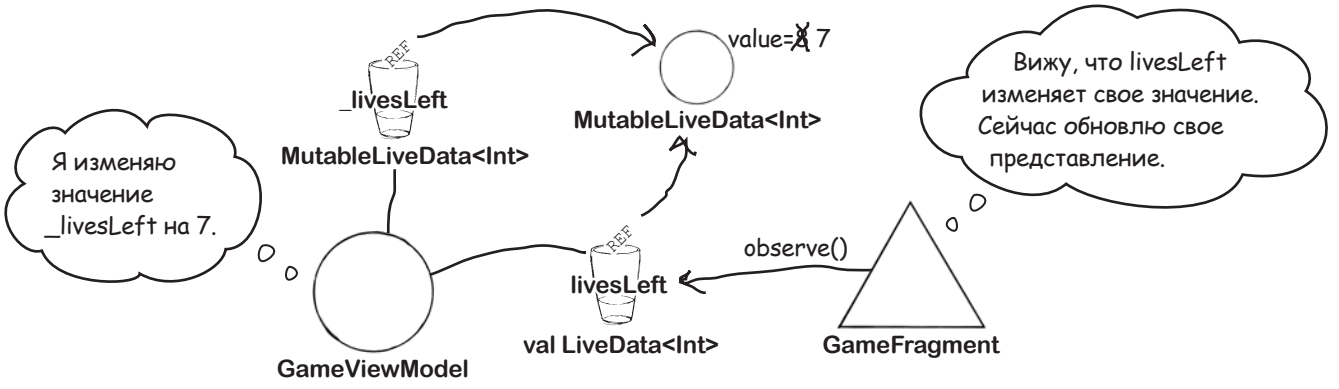


2 **Инициализируются свойства GameViewModel.**
`livesLeft`, `incorrectGuesses` и `secretWordDisplay` — свойства `Live Data`, ссылающиеся на те же объекты, что и их резервные свойства `MutableLiveData`.

Здесь показаны только свойства _livesLeft и livesLeft для упрощения диаграммы.

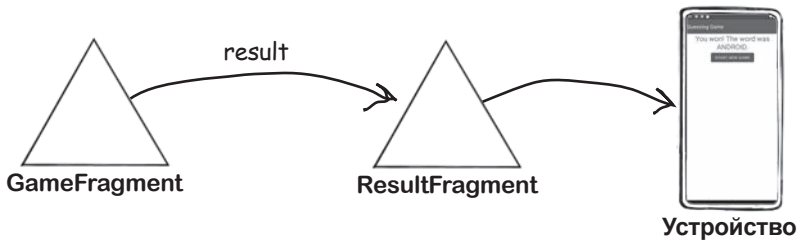


3 **GameFragment наблюдает за свойствами livesLeft, incorrectGuesses и secretWordDisplay.**
`GameFragment` не может обновлять эти свойства, но реагирует, когда `GameViewModel` обновляет какие-либо из резервных свойств, потому что они ссылаются на один и тот же используемый объект.



История продолжается

4 **GameFragment продолжает реагировать на изменения значений до того момента, когда isWon() или isLost() вернет true.** `GameFragment` переходит к фрагменту `ResultFragment` и передает ему результат. `ResultFragment` отображает результат на экране.

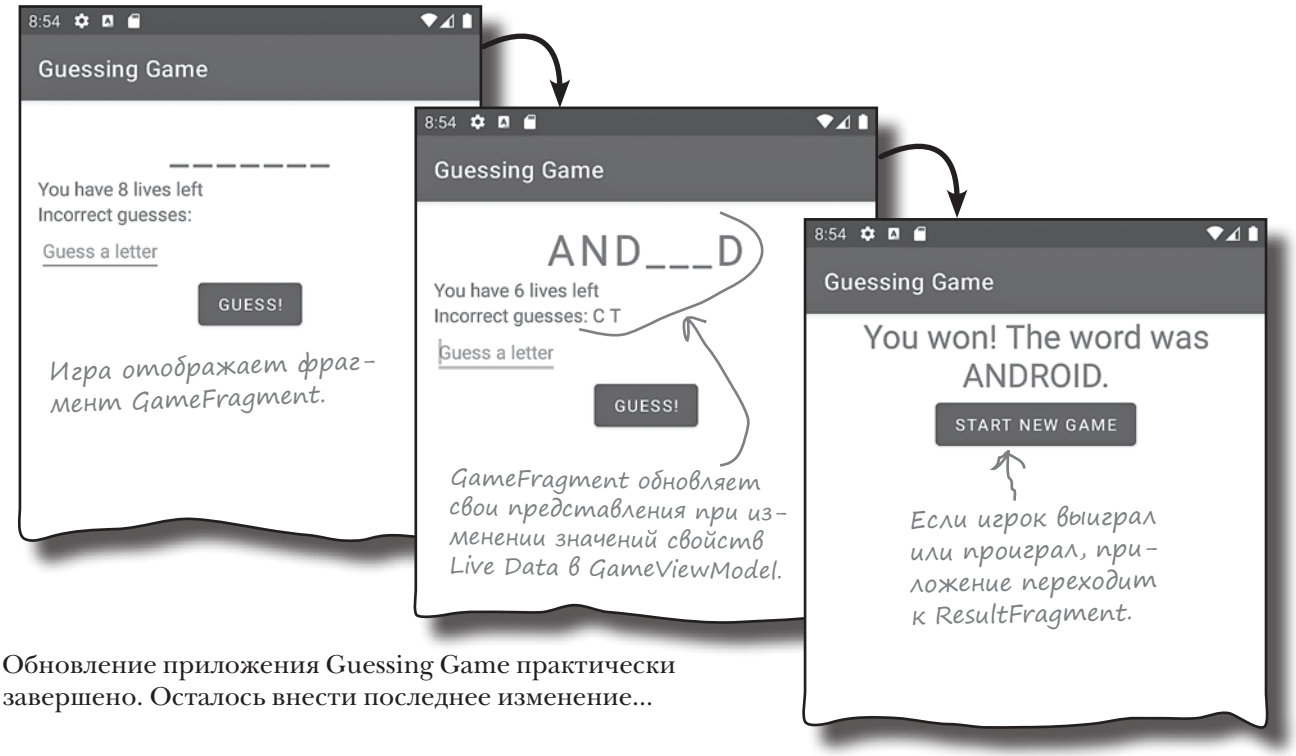


Проведем тест-драйв приложения.

Тест-драйв

Когда вы запускаете приложение, оно работает так же, как прежде. Но в новой версии свойства `MutableLiveData` класса `GameViewModel` защищены — доступ к ним со стороны `GameFragment` ограничен.

- [x] Использование Live Data
- [x] Защита свойств
- [ ] Добавление gameOver



Обновление приложения Guessing Game практически завершено. Осталось внести последнее изменение...

GameFragment содержит игровую логику

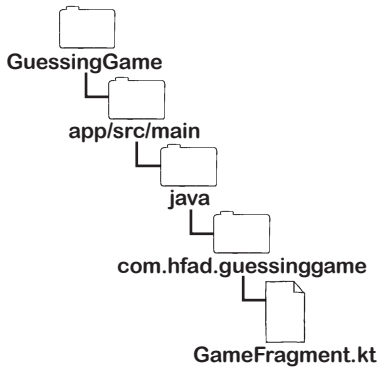
В текущей версии приложения `GameFragment` решает, завершена ли игра, вызывая методы `isWon()` и `isLost()` класса `GameViewModel` после каждого предположения пользователя. Если один из этих методов возвращает *true*, `GameFragment` переходит к `ResultFragment` и передает результат.

Текущая версия кода выглядит так:

```
...
    binding.guessButton.setOnClickListener() {
        viewModel.makeGuess(binding.guess.text.toString().uppercase())
        binding.guess.text = null
        if (viewModel.isWon() || viewModel.isLost()) {
            val action = GameFragmentDirections
                .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
            view.findNavController().navigate(action)
        }
    }
...
```

Каждый раз, когда пользователь вводит предположение, GameFragment проверяет возможный выигрыш или проигрыш.

Если игра выиграна или проиграна, GameFragment переходит к ResultFragment.

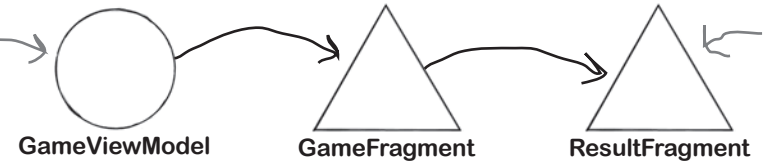


Недостаток такого подхода заключается в том, что решение о завершении игры принимает `GameFragment` вместо `GameViewModel`. Так как определение завершения игры является игровым решением, за него должен отвечать класс `GameViewModel`, а не `GameFragment`.

Принятие решения о завершении игры в GameViewModel

Для устранения этого недостатка мы добавим в `GameViewModel` свойство `MutableLiveData<Boolean>` с именем `_gameOver`, для обращения к значению которого будет использоваться свойство `LiveData` с именем `gameOver`. Этому свойству присваивается значение *true*, когда пользователь выигрывает или проигрывает игру. Когда такое происходит, `GameFragment` реагирует на это переходом к `ResultFragment`.

GameViewModel использует свойство gameOver для оповещения GameFragment о завершении игры.



Если игра завершена, GameFragment переходит к ResultFragment.

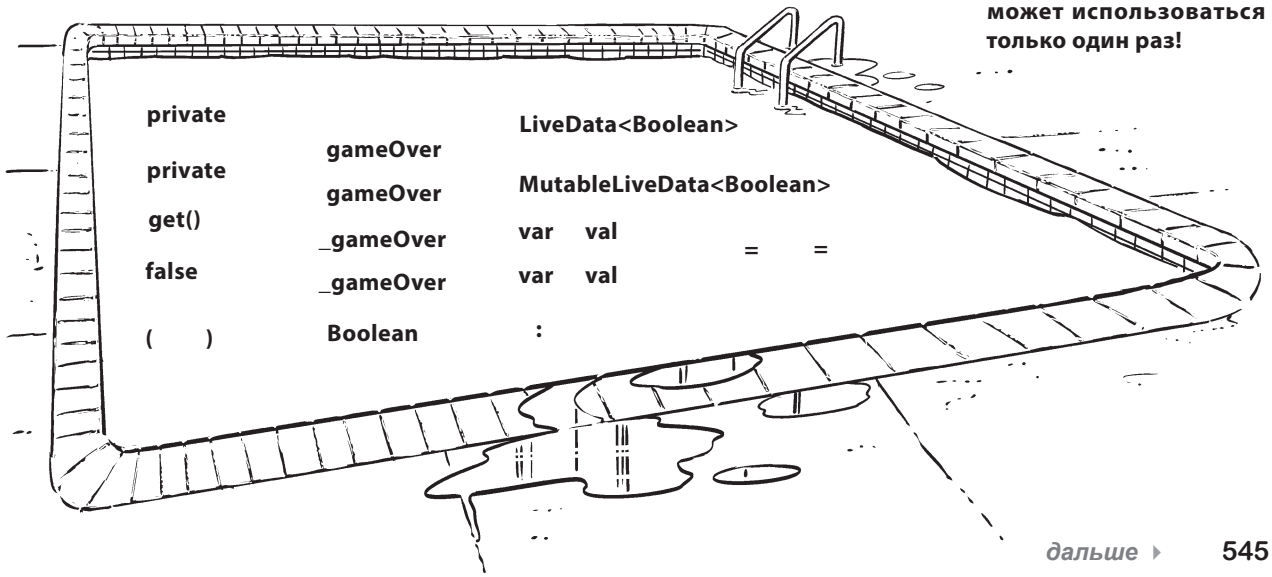
Необходимые изменения вам уже знакомы. Но прежде чем переходить к рассмотрению кода, проверьте свои силы на следующем упражнении.

У бассейна

Добавьте свойство `gameOver` (с резервным свойством `_gameOver`) в `GameViewModel`, чтобы класс `GameFragment` мог реагировать на обновление его значения (тип `Boolean`). Свойство должно инициализироваться значением *false*. Выловите из бассейна фрагменты кода и расставьте их в пустых строках. Каждый фрагмент кода может использоваться **только один раз**, при этом все фрагменты использовать не обязательно.

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    ...
    .....
    .....
    .....
    ...
}
```

Каждый фрагмент кода может использоваться только один раз!



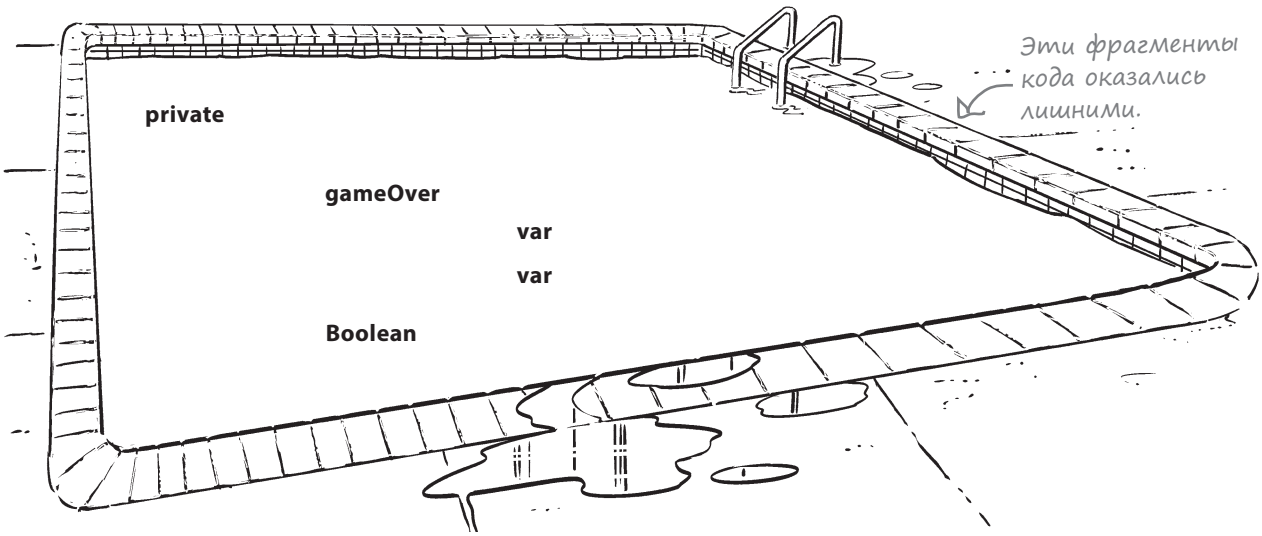
У бассейна. Решение

Добавьте свойство `gameOver` (с резервным свойством `_gameOver`) в `GameViewModel`, чтобы класс `GameFragment` мог реагировать на обновление его значения (тип `Boolean`). Свойство должно инициализироваться значением *false*. Выловите из бассейна фрагменты кода и расставьте их в пустых строках. Каждый фрагмент кода может использоваться **только один раз**, при этом все фрагменты использовать не обязательно.

```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    ...
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver: LiveData<Boolean>
        get() = _gameOver
    ...
}
```

_gameOver — приватное свойство MutableLiveData<Boolean>, которому присваивается false.

gameOver предоставляет GameFragment доступ только для чтения к значению _gameOver.



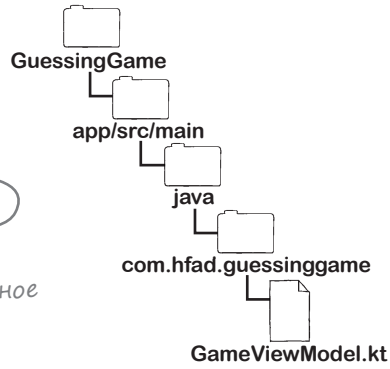
Полный код GameViewModel.kt

Необходимо добавить в `GameViewModel` свойство `gameOver`, а также резервное свойство `_gameOver`. Метод `makeGuess()` присваивает ему *true*, если пользователь угадал все буквы загаданного слова либо у него кончились жизни.

Ниже приведен полный код `GameViewModel`; обновите файл *GameViewModel.kt* (изменения выделены жирным шрифтом):

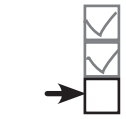
```
package com.hfad.guessinggame

import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData

class GameViewModel : ViewModel() {
    private val words = listOf("Android", "Activity", "Fragment")
    private val secretWord = words.random().uppercase()
    private val _secretWordDisplay = MutableLiveData<String>()
    val secretWordDisplay: LiveData<String>
        get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
        get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
        get() = _livesLeft
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver: LiveData<Boolean>
        get() = _gameOver

    init {
        _secretWordDisplay.value = deriveSecretWordDisplay()
    }
```

Добавляет свойство gameOver и его резервное свойство _gameOver.



Продолжение на следующей странице.

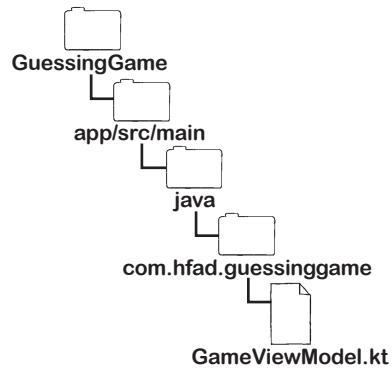
Полный код GameViewModel.kt (продолжение)




```
    private fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }

    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                _secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                _incorrectGuesses.value += "$guess "
                _livesLeft.value = _livesLeft.value?.minus(1)
            }
            if (isWon() || isLost()) _gameOver.value = true
        }
    }

    private fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    private fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }
}
```

Задает свойству _gameOver значение true в случае выигрыша или проигрыша.

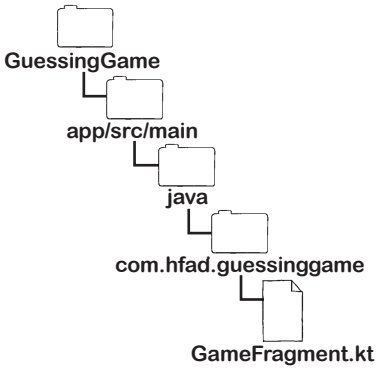
GameFragment уже не нужно вызывать эти методы, поэтому их можно объявить с модификатором private.



Отслеживание нового свойства в GameFragment

После того как свойство `gameOver` будет добавлено в `GameViewModel`, необходимо заставить `GameFragment` реагировать на его обновления. Для этого фрагмент будет наблюдать за свойством, чтобы когда оно примет значение *true*, фрагмент переходил к `ResultFragment`.

Ниже приведен код `GameFragment`; обновите файл *GameFragment.kt* (изменения выделены жирным шрифтом):

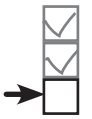



```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
            binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
        })

        viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
            binding.lives.text = "You have $newValue lives left"
        })
```

Код на этой странице изменять не придется.

Продолжение на следующей странице.

GameFragment.kt (продолжение)

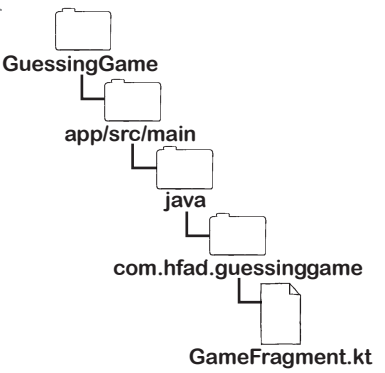



```
        viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
            binding.word.text = newValue
        })

        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
            if (viewModel.isWon() || viewModel.isLost()) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Заставляет фрагмент наблюдать за свойством gameOver модели представления.

Если gameOver присвоено значение true, происходит переход к ResultFragment.

Удалите эти строки.

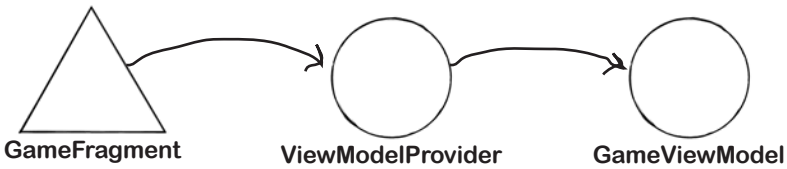


Вот и все! Давайте разберемся, что происходит во время выполнения приложения.

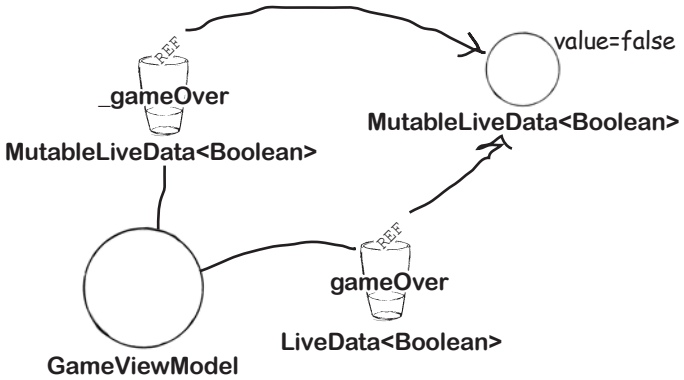
Что происходит при выполнении приложения

Во время выполнения приложения происходят следующие события:

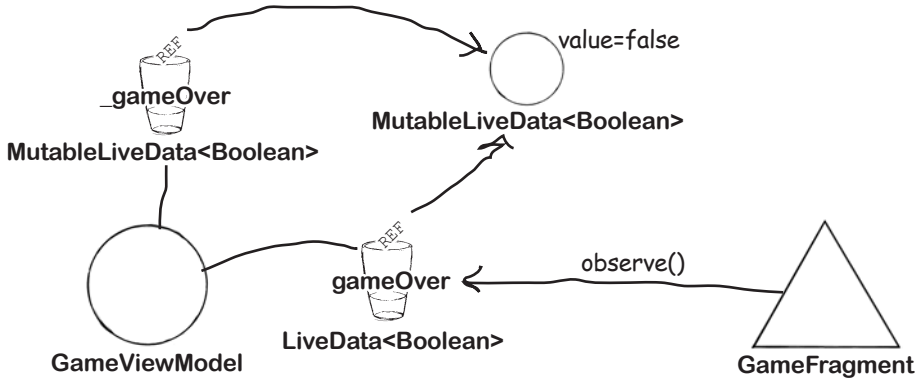
1. **GameFragment запрашивает у класса ViewModelProvider экземпляр GameViewModel.**



2. **Инициализируются свойства GameViewModel.**
Свойства `_gameOver` и `gameOver` ссылаются на объект `MutableLiveData<Boolean>`, у которого свойству `value` присваивается значение *false*.

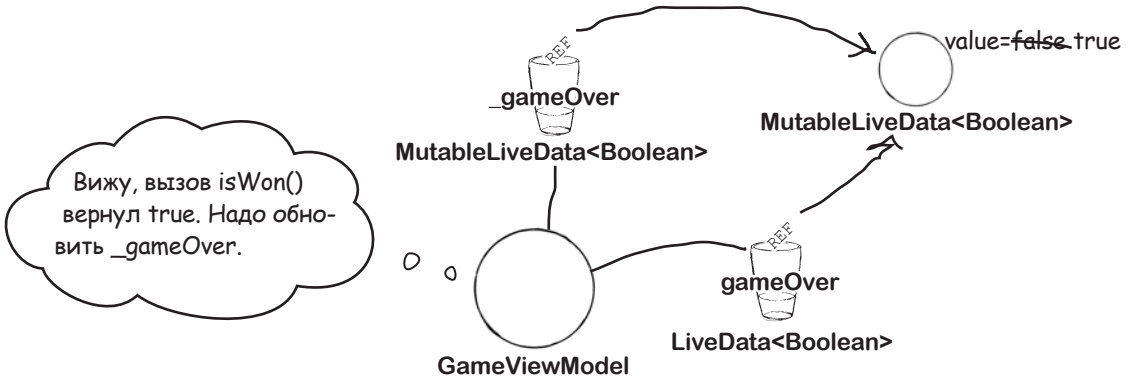


3. **GameFragment наблюдает за свойством gameOver класса GameViewModel.**
`GameFragment` не может обновить объект `MutableLiveData`, ссылка на который хранится в свойстве `gameOver`, но может отреагировать на изменение его значения.

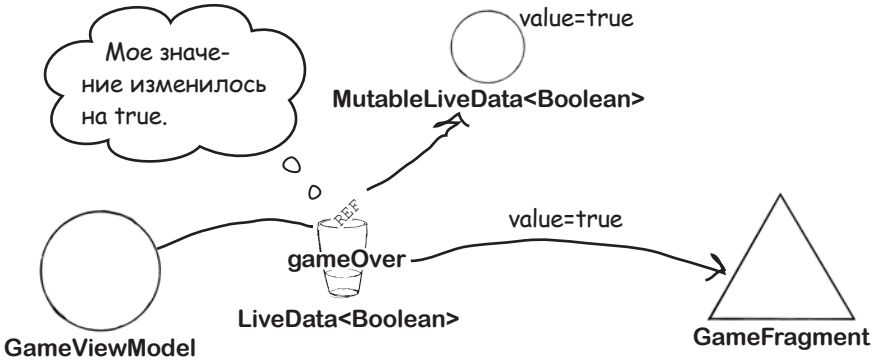


История продолжается

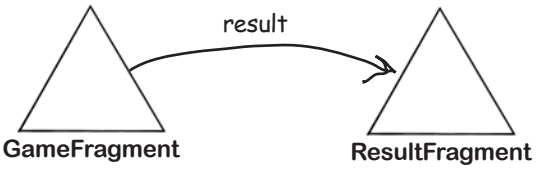
4 **Каждый раз, когда вызывается метод makeGuess() класса GameViewModel, он проверяет, был ли получен результат true при вызове isWon() или isLost().**
Если один из двух результатов равен *true*, то свойству `_gameOver` присваивается значение *true*.



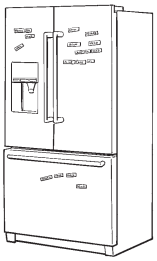
5 **GameFragment замечает, что значение стало равно true, через свойство gameOver класса GameViewModel.**
Новое значение передается `GameFragment`.



6 **GameFragment реагирует переходом к ResultFragment с передачей result.**



Вскоре мы проведем тест-драйв приложения, но сначала небольшое упражнение.



Развлечения с магнитами

Кто-то выложил код фрагмента с именем `LotteryFragment` на холодильнике, но от порыва ветра часть магнитов упала на пол. Удастся ли вам восстановить код фрагмента?

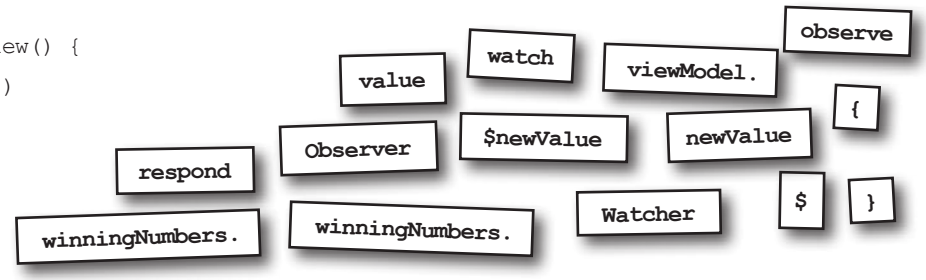
Фрагмент должен наблюдать за свойством `winningNumbers` класса `LotteryViewModel`, которое определяется так:

```
private val _winningNumbers = MutableLiveData<String>()
val winningNumbers: LiveData<String>
    get() = _winningNumbers
```

Когда `winningNumbers` изменяется, фрагмент `LotteryFragment` должен обновить представление `numbers` новым значением.

```
class LotteryFragment : Fragment() {
    private var _binding: FragmentLotteryBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: LotteryViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentLotteryBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(LotteryViewModel::class.java)


        viewModel.                              (viewLifecycleOwner,          { newValue ->

            binding.numbers.text = "Winning numbers:                                          "
        })
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Использовать все магниты не обязательно.



Развлечения с магнитами. Решение

Кто-то выложил код фрагмента с именем `LotteryFragment` на холодильнике, но от порыва ветра часть магнитов упала на пол. Удастся ли вам восстановить код фрагмента?

Фрагмент должен наблюдать за свойством `winningNumbers` класса `LotteryViewModel`, которое определяется так:

```
private val _winningNumbers = MutableLiveData<String>()
val winningNumbers: LiveData<String>
    get() = _winningNumbers
```

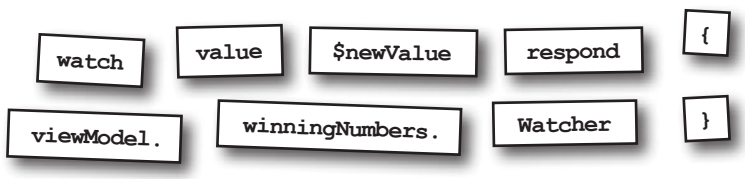
Когда `winningNumbers` изменяется, фрагмент `LotteryFragment` должен обновить представление `numbers` новым значением.

```
class LotteryFragment : Fragment() {
    private var _binding: FragmentLotteryBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: LotteryViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentLotteryBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(LotteryViewModel::class.java)

        viewModel.winningNumbers.observe(viewLifecycleOwner, Observer { newValue ->

            binding.numbers.text = "Winning numbers: $newValue"
        })
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Наблюдать за свойством winningNumbers.

Новое значение свойства выводится в представлении numbers.

Эти магниты не понадобились.



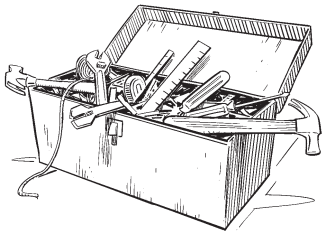
Тест-драйв

Приложение работает так же, как прежде. Однако в новой версии решение о завершении игры принимает `GameViewModel`, а не `GameFragment`. Фрагмент просто наблюдает за свойством `gameOver` модели представления и переходит к `ResultFragment`, когда оно принимает значение *true*.

Поздравляем! Вы построили приложение, которое использует Live Data для реагирования на изменения в момент их возникновения. В следующей главе мы расширим эту идею и воспользуемся новым механизмом, который называется связыванием данных.

ГЛАВА 12

Ваш инструментарий Android



Глава 12 осталась позади, а ваш инструментарий пополнился данными Live Data.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Механизм Live Data позволяет свойствам модели представления сообщать заинтересованным сторонам (например, фрагментам или активностям) об изменении их значений.
- Чтобы свойство использовало Live Data, определите его с ключевым словом `val` и измените его тип на `MutableLiveData<Тип>`, где `Тип` — тип данных, которые должны храниться в свойстве.
- Свойство `value` объекта `MutableLiveData` используется для хранения его значения. Это свойство допускает null.
- Чтобы фрагмент или активность реагировали на изменения модели представления, вызовите метод `observe` свойства и укажите, как должно использоваться новое значение.
- Прямой доступ к свойствам модели представления можно ограничить при помощи резервных свойств.
- Тип `Live Data` похож на `MutableLiveData`, но он является неизменяемым. Он используется для предоставления доступа к значению объекта `MutableLiveData` с запретом на изменение его значения.

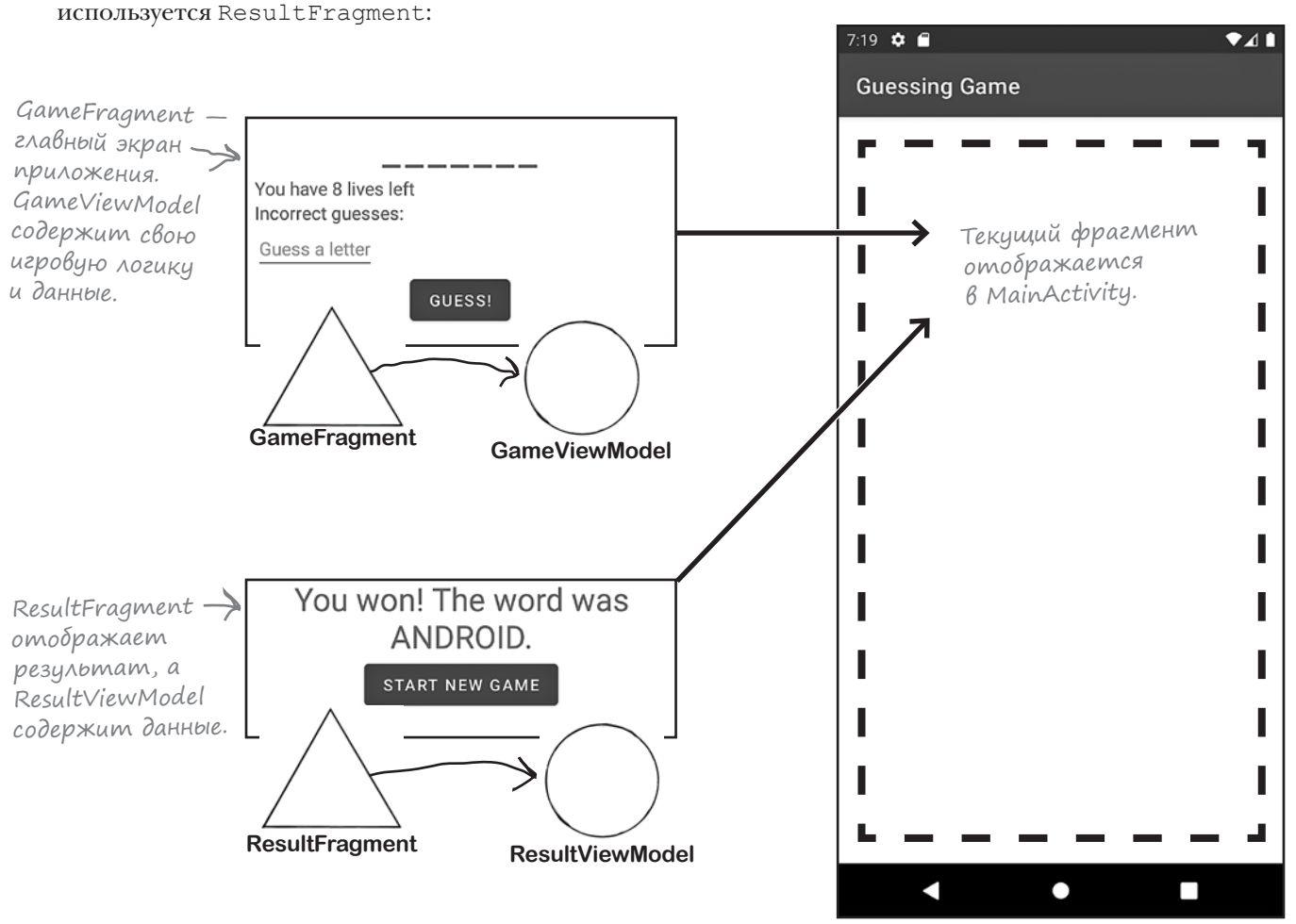
Построение умных макетов

Возможности макетов не ограничиваются управлением внешним видом вашего приложения. У всех макетов, написанных нами ранее, поведение должно было определяться кодом активности или фрагмента. Но только представьте, что макеты могли бы думать самостоятельно и принимать собственные решения. В этой главе представлен механизм **связывания данных**: способ наращивания интеллекта ваших макетов. Вы узнаете, **как заставить представления получать данные** непосредственно от модели представления. Мы воспользуемся **связыванием слушателей**, чтобы кнопки вызывали свои методы. Вы даже увидите, как **одна простая строка кода позволяет реагировать на обновления Live Data**. Скоро ваши макеты станут более мощными, чем когда-либо.

Снова к приложению Guessing Game

В двух предыдущих главах мы построили приложение Guessing Game, в котором пользователь отгадывает буквы, входящие в секретное слово. Когда пользователь успешно отгадывает все буквы или у него кончаются жизни, игра завершается.

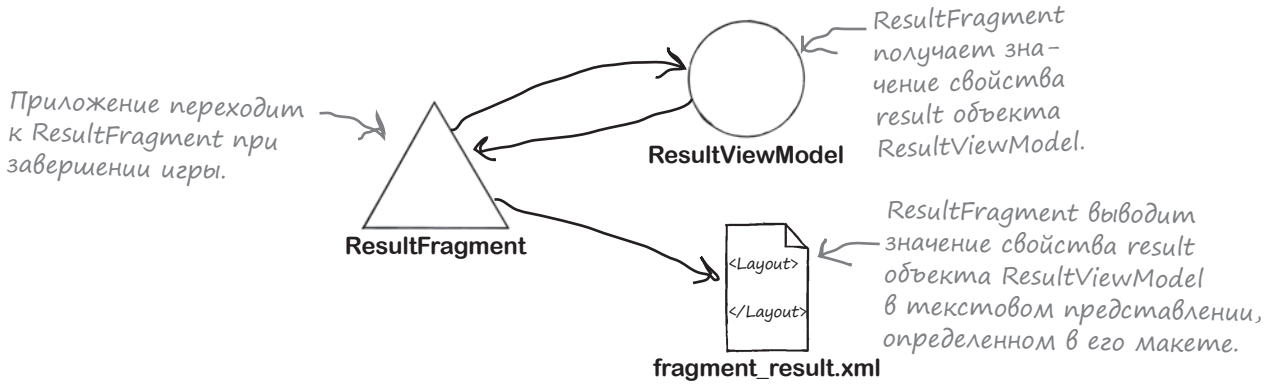
В приложении используются две модели представления (`GameViewModel` и `ResultViewModel`) для хранения игровой логики и данных приложения; они сохраняют свое состояние при повороте экрана устройства. `GameViewModel` используется `GameFragment`, а `ResultViewModel` используется `ResultFragment`:



Первый экран, который видит пользователь, — `GameFragment`. Его взаимодействия с этим экраном составляют суть игры. На этом экране отображается информация (например, количество оставшихся жизней и все ошибочные предположения пользователя), а пользователь получает возможность вводить предположения. После завершения игры приложение переходит к `ResultFragment`. На этом экране выводится информация о том, выиграл или проиграл пользователь и какое слово было загадано.

Фрагменты обновляют представления в своих макетах

Каждый фрагмент приложения отвечает за обновление представлений в своих макетах. Например, `ResultFragment` заполняет свое текстовое представление `won_lost` значением свойства `result` объекта `ResultViewModel`:



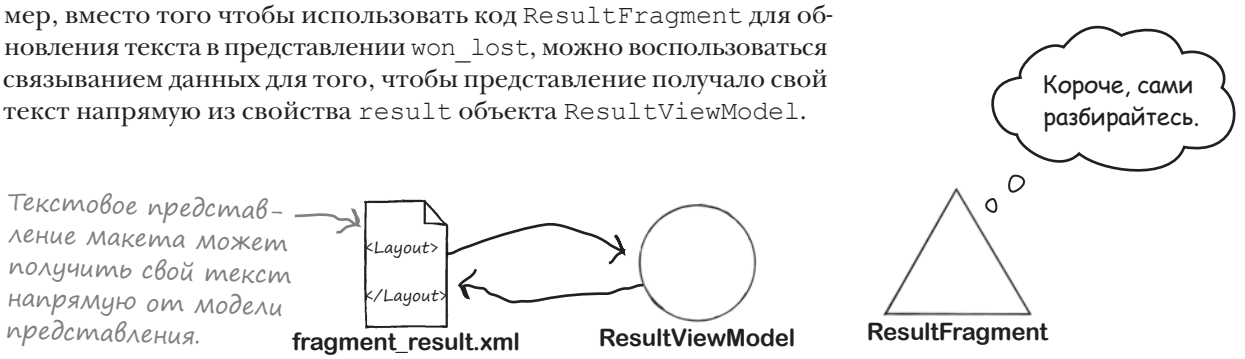
Такой подход работает, но у него есть недостаток. Когда фрагменты отвечают за поддержание актуальности представлений, код Kotlin становится сложнее и создает больше проблем с чтением и сопровождением.

Существует ли другой способ?

Представления могут обновлять себя сами

Связывание данных также является частью Android Jetpack.

В альтернативном решении используется прием, называемый **связыванием данных**. При связывании данных **представления получают свои значения напрямую от модели представления**, так что коду фрагмента уже не приходится заниматься их обновлением. Например, вместо того чтобы использовать код `ResultFragment` для обновления текста в представлении `won_lost`, можно воспользоваться связыванием данных для того, чтобы представление получало свой текст напрямую из свойства `result` объекта `ResultViewModel`.

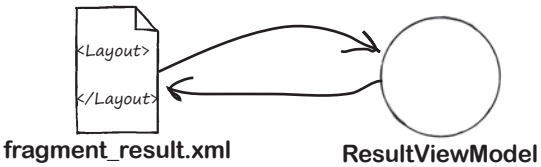


В этой главе вы научитесь применять связывание данных на примере приложения Guessing Game. Рассмотрим последовательность действий для его реализации.

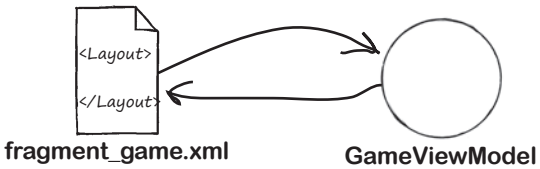
Что мы собираемся сделать

Ниже описаны основные действия, которые необходимо выполнить для того, чтобы в приложении Guessing Game использовалось связывание данных:

1. **Реализация связывания данных для ResultFragment.** Мы обновим фрагмент `ResultFragment`, чтобы в его макете использовалось связывание данных для отображения свойства `result` непосредственно из `ResultViewModel`.



2. **Реализация связывания данных для GameFragment.** Затем мы изменим фрагмент `GameFragment` так, чтобы его макет использовал связывание данных для отображения значений свойств из `GameViewModel`. Заодно мы сделаем так, чтобы макет динамически реагировал на любые обновления данных Live Data.



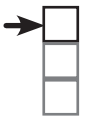
3. **Добавление кнопки Finish Game в GameFragment.** Наконец, мы добавим в макет `GameFragment` новую кнопку, щелчок на которой немедленно завершает игру.

Когда пользователь щелкает на кнопке Finish Game, приложение немедленно переходит к ResultFragment и выводит загаданное слово.

Итак, за дело!

Не забудьте!

В этой главе мы снова изменим приложение Guessing Game. Откройте проект этого приложения.

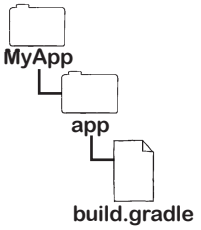


Включение связывания данных в файле build.gradle приложения

Как и в случае со связыванием представлений, чтобы вы могли использовать связывание данных в своем представлении, его необходимо явно включить в разделе `android` файла *build.gradle* приложения. Код включения связывания данных выглядит примерно так:

Эти строки должны быть включены в раздел android файла build.gradle приложения.

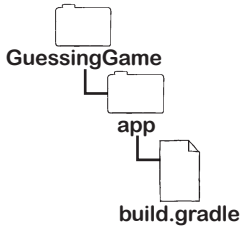
```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```



Мы уже включили связывание представлений в приложении Guessing Game, давайте добавим к нему связывание данных. Откройте файл *GuessingGame/app/build.gradle*, обновите раздел `buildFeatures` и приведите его к следующему виду:

Добавьте эту строку.

```
android {
    ...
    buildFeatures {
        viewBinding true
        dataBinding true
    }
}
```



Затем выберите вариант Sync Now, чтобы синхронизировать это изменение с остальными частями проекта.

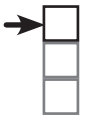
После того как связывание данных будет включено, реализуем его в `ResultFragment`.

Часто Задаваемые Вопросы

В: Я вижу, что в приложении Guessing Game включено как связывание представлений, *так и* связывание данных. Обязательно ли включать связывание представлений, чтобы использовать связывание данных?

О: Нет, не обязательно. Связывание данных очень часто включается без связывания представлений; вы больше узнаете об этом позднее.

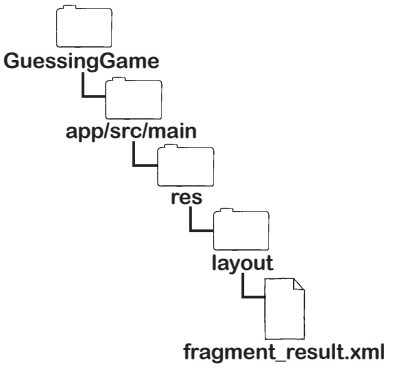
А пока все, что действительно необходимо знать: мы оставляем включенным связывание представлений, чтобы весь существующий код приложения продолжал работать.



ResultFragment обновляет текст в своем макете

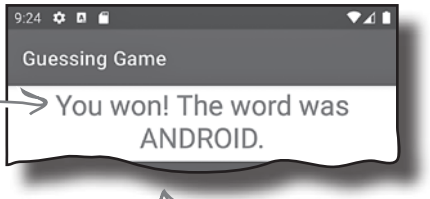
Как вы, возможно, помните, макет `ResultFragment` содержит текстовое представление с идентификатором `won_lost`, которое определяется следующим образом:

```
<LinearLayout ...>
    <TextView
        android:id="@+id/won_lost"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="28sp" />
    ...
</LinearLayout>
```



Когда приложение переходит к `ResultFragment`, фрагмент получает значение свойства `result` от своей модели представления и отображает его в текстовом представлении, для чего используется следующий код:

```
binding.wonLost.text = viewModel.result
```



Мы изменим фрагмент и его макет, чтобы в нем использовалось связывание данных. Вместо того чтобы обновлять текстовое представление из кода `ResultFragment`, мы сделаем так, чтобы текстовое представление получало свое значение прямо от модели представления.

В настоящее время текст won_lost text задается кодом в ResultFragment. Мы изменим фрагмент для использования связывания данных, чтобы представление получало свое текстовое значение прямо из свойства result модели представления.

Как реализуется связывание данных

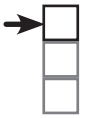
Чтобы текстовое представление получало свой текст от `ResultViewModel`, необходимо обновить код фрагмента `ResultFragment` и его макет. Для этого необходимо выполнить следующие действия:

1. **Добавление элементов <layout> и <data> в макет.**
   Элемент `<layout>` необходим для связывания данных, а элемент `<data>` используется для определения переменной связывания данных, которая соединяет макет с моделью представления.

2. **Переменной связывания данных в макете присваивается экземпляр модели представления.**
   Это будет сделано в коде фрагмента.

3. **Переменная связывания данных используется для обращения к свойствам модели представления.**
   Мы обновим текстовое представление `won_lost`, чтобы оно получало свой текст прямо из модели представления.

Начнем с обновления кода макета.



ResultFragment
GameFragment
Добавление кнопки

1. Добавление элементов <layout> и <data>

Каждый макет, в котором используется связывание данных, записывается по следующей схеме:

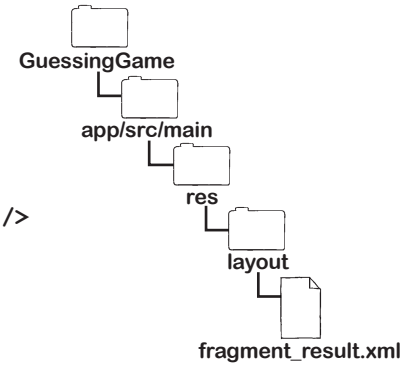
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".ResultFragment">

    <data>
        <variable
            name="resultViewModel"
            type="com.hfad.guessinggame.ResultViewModel" />
    </data>

    ...

</layout>
```

Элемент <layout> находится в корне макета.

Обновлять код пока не нужно; полный код будет приведен через несколько страниц.

В разделе <data> определяются все переменные, используемые для связывания данных.

Здесь размещается элемент представления или группы представлений. Разметка очень похожа на разметку «обычного» макета, не использующего связывание данных.

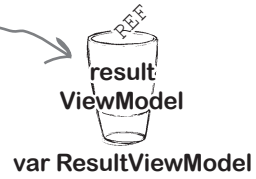


В приведенном макете корневым элементом является элемент **<layout>**. Он включает связывание данных в макете и поэтому должен присутствовать во *всех* макетах, которые должны использовать связывание данных.

В элементе <layout> находится элемент **<data>**. В нем задаются любые переменные (определяемые элементом **<variable>**), необходимые для связывания данных, например модель представления, от которой представления макета должны получать свои данные. Так, в приведенном выше примере используется код:

```
<data>
    <variable
        name="resultViewModel"
        type="com.hfad.guessinggame.ResultViewModel" />
</data>
```

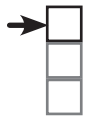
Определяет переменную связывания данных ResultViewModel с именем resultViewModel.



для определения переменной связывания данных с именем resultViewModel, относящейся к типу ResultViewModel.

Элемент <layout> также включает представление или группу представлений для иерархии представлений макета. Например, если вы хотите отобразить представления в линейном макете, в элемент <layout> следует поместить элемент <LinearLayout> вместе со всеми представлениями. Через несколько страниц вы увидите, как это делается.

После того как вы определите переменную связывания данных, необходимо присвоить ей значение.



ResultFragment
GameFragment
Добавление кнопки

2. Присваивание значения переменной связывания данных в макете

Значение переменной связывания данных макета задается в коде Kotlin.

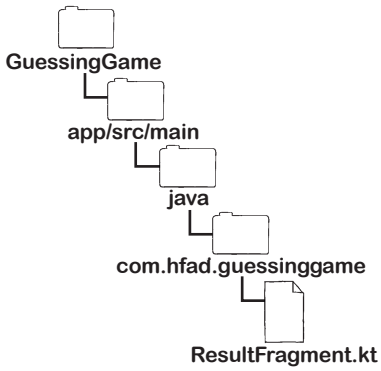
На предыдущей странице был приведен код добавления переменной связывания данных (с именем `resultViewModel`) в файл макета `ResultFragment` *fragment_result.xml*:

```
<data>
    <variable
        name="resultViewModel"
        type="com.hfad.guessinggame.ResultViewModel" />
</data>
```

Переменная имеет тип `ResultViewModel`, и ей должен быть присвоен объект соответствующего типа.

Напомним, что метод `onCreateView()` класса `ResultFragment` включает код получения объекта `ResultViewModel`:

```
...
viewModelFactory = ResultViewModelFactory(result)
viewModel = ViewModelProvider(this, viewModelFactory)
    .get(ResultViewModel::class.java)
...
```

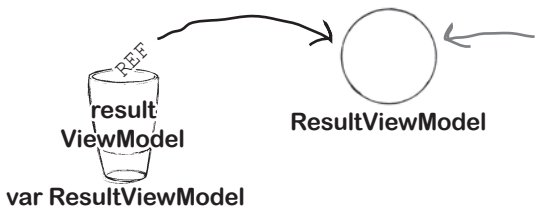


Чтобы присвоить эту модель представления переменной связывания данных макета `resultViewModel`, добавим следующую строку в код `onCreateView()`:

```
binding.resultViewModel = viewModel
```

← *Задает значение переменной связывания данных макета.*

Эта команда использует свойство `binding` фрагмента для того, чтобы присвоить свойству `resultViewModel` макета объект модели представления фрагмента:



Свойству resultViewModel присвоен объект ResultViewModel фрагмента, поэтому для макета доступны свойства и методы модели представления.

После того как вы зададите переменную связывания данных, макет сможет использовать ее для обращения к свойствам и методам модели представления. Давайте посмотрим, как это делается.

3. Использование переменной связывания данных макета для обращения к модели представления

Макет `ResultFragment` определяет текстовое представление `won_lost`, в котором должно выводиться свойство `result` объекта `ResultViewModel`. Его текст задается в коде фрагмента следующим образом:

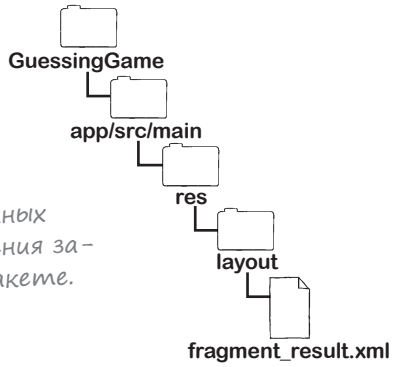
```
binding.wonLost.text = viewModel.result
```

Фрагмент заполняет представление won_lost текстом, который является значением свойства result модели представления.

После определения переменной связывания данных для модели представления в коде макета мы сможем использовать связывание данных для получения текста. Код решения этой задачи выглядит так:

```
<TextView
    android:id="@+id/won_lost"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:textSize="28sp"
    android:text="@{resultViewModel.result}" />
```

При связывании данных текст представления задается в самом макете.



В этом коде переменная `resultViewModel` используется для получения значения свойства `result` модели представления. Затем содержимому текстового представления присваивается значение этого свойства.

И это все, что необходимо знать для реализации связывания данных в `ResultFragment`. Полный код *result_fragment.xml* и *ResultFragment.kt* будет приведен через пару страниц.

Часто Задаваемые Вопросы

В: У всех остальных элементов, которые мы добавляли в макеты, — таких как `<LinearLayout>` и `<Button>`, имена начинались с буквы верхнего регистра. Почему элементы `<layout>`, `<data>` и `<variable>` записаны в нижнем регистре?

О: У других элементов, о которых вы узнали ранее, имена начинались с прописных букв, потому что они являются именами классов типов `View`. Например, элемент `<LinearLayout>` относится к классу `LinearLayout` из пакета `android.view`.

Элементы `<layout>`, `<data>` и `<variable>` записываются в нижнем регистре, потому что они представляют инструкции, а не разновидности `View`. Например, когда Android встречает элемент `<variable>`, он рассматривается как инструкция создания переменной.

Полный код fragment_result.xml

Ниже приведен полный код макета `ResultFragment`; обновите файл *fragment_result.xml* (изменения выделены жирным шрифтом):

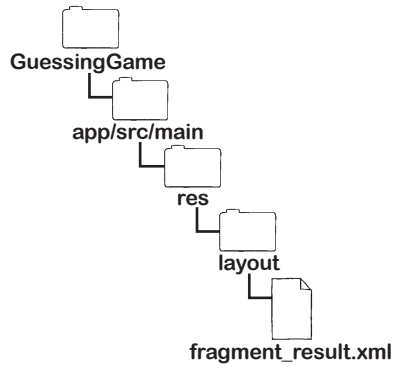
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".ResultFragment">

    <data>
        <variable
            name="resultViewModel"
            type="com.hfad.guessinggame.ResultViewModel" />
    </data>

    <LinearLayout
        ~~xmlns:android="http://schemas.android.com/apk/res/android"~~
        ~~xmlns:tools="http://schemas.android.com/tools"~~
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        ~~tools:context=".ResultFragment">~~

        <TextView
            android:id="@+id/won_lost"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:textSize="28sp"
            android:text="@{resultViewModel.result}" />

        <Button
            android:id="@+id/new_game_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Start new game"/>
    </LinearLayout>
</layout>
```

Добавляем элемент <layout>.

Переместите эти строки из <LinearLayout> в элемент <layout>, чтобы они оставались в корневом элементе...

Определяем переменную связывания данных.

Удалите эти строки и добавьте их в <layout>.

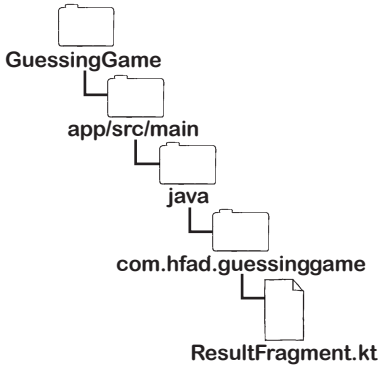


Добавьте эту строку, в которой связывание данных используется для задания текста представления.

Закрывающий тег элемента <layout>

Полный код ResultFragment.kt

Код фрагмента тоже необходимо обновить. Измените содержимое файла *ResultFragment.kt* так, как показано ниже (изменения выделены жирным шрифтом):



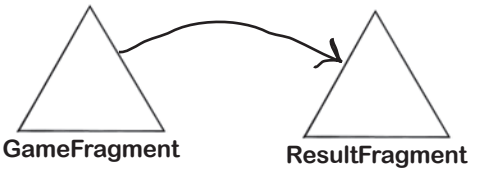

```
package com.hfad.guessinggame

...   Изменять команды импортирования не нужно, поэтому они опущены.

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        val view = binding.root

        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
            .get(ResultViewModel::class.java)
        binding.resultViewModel = viewModel   Создает переменную связывания данных макета.
        binding.wonLost.text = viewModel.result   Эта строка более не нужна, так как текст представления задается через механизм связывания данных.

        binding.newGameButton.setOnClickListener {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Посмотрим, что происходит во время выполнения приложения.

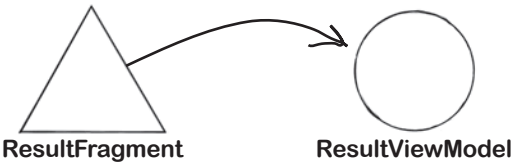
Что происходит во время выполнения приложения

При выполнении приложения происходят следующие события:

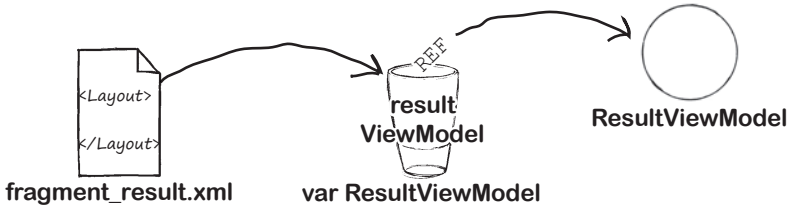
1. **Когда пользователь завершает игру, приложение переходит к ResultFragment.**



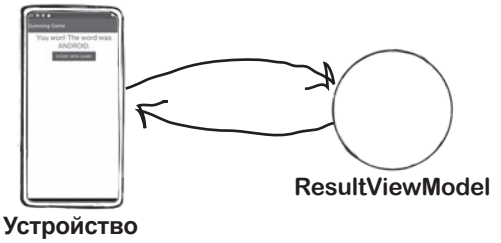
2. **ResultFragment получает ссылку на свой объект ResultViewModel.**



3. **ResultFragment присваивает объект ResultViewModel переменной связывания данных resultViewModel макета.**
Представления макета теперь могут использовать переменную для обращения к свойствам и методам модели представления.



4. **Текстом представления won_lost становится значение свойства result объекта ResultViewModel.**
Текст отображается на экране устройства.

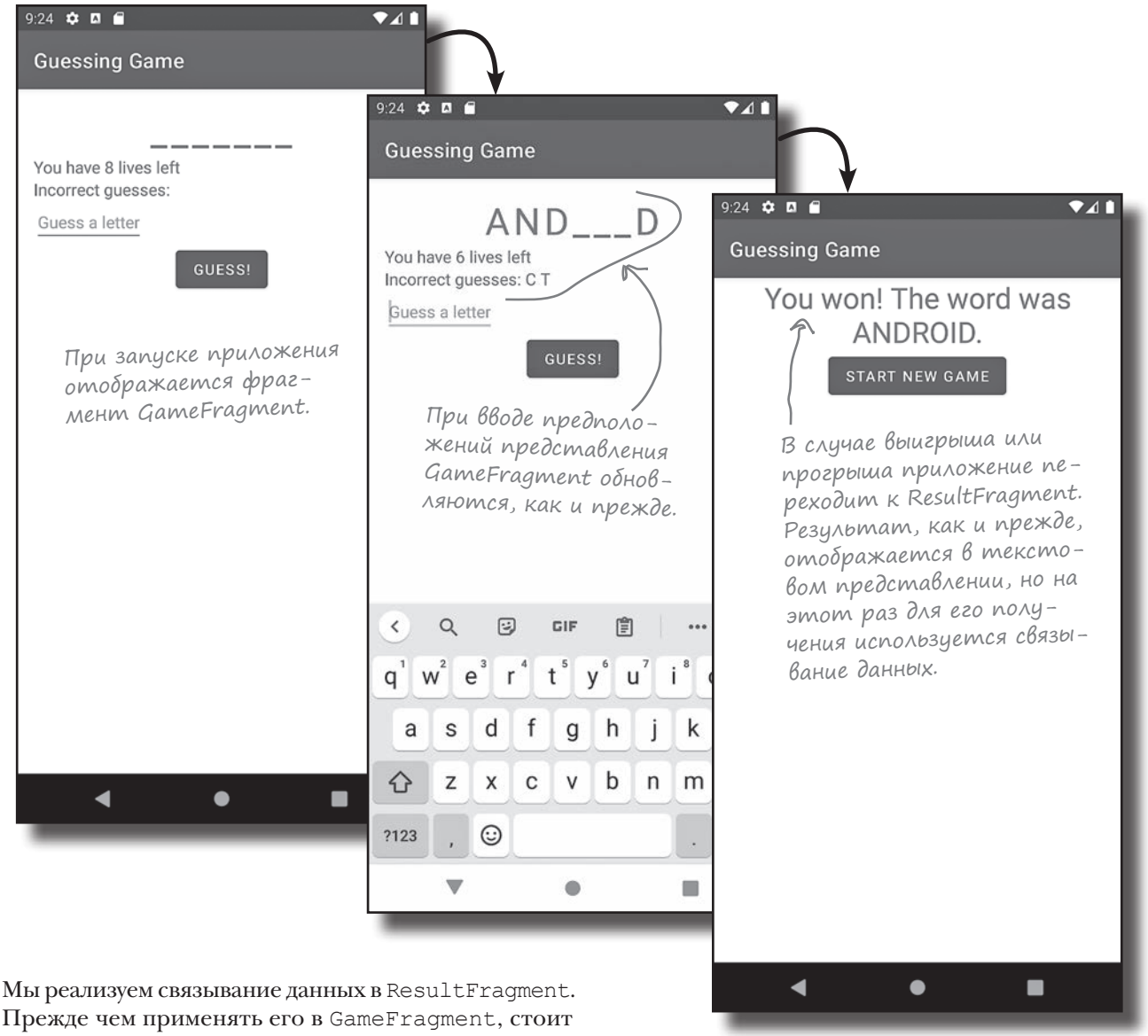


Давайте посмотрим, как работает приложение.

Тест-драйв

ResultFragment
GameFragment
Добавление кнопки

Приложение работает точно так же, как прежде. Однако в новой версии макет `ResultFragment` получает результат игры непосредственно от `ResultViewModel`.



Мы реализуем связывание данных в `ResultFragment`. Прежде чем применять его в `GameFragment`, стоит поближе присмотреться к элементу `<layout>` и понять, почему он необходим для связывания данных.

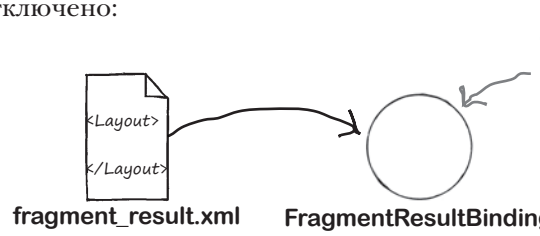
<layout> под увеличительным стеклом

Как говорилось ранее, каждый макет, использующий связывание данных, *должен* содержать корневой элемент `<layout>`:

```
<layout ... >
    ...
</layout>
```

Когда вы включаете связывание данных в файле *build.gradle* приложения, для каждого макета с корневым элементом `<layout>` генерируется класс связывания. **Это тот же класс связывания, который генерируется при включении связывания представлений.**

В приложении Guessing Game файл макета *fragment_result.xml* содержит корневой элемент `<layout>`. Это означает, что его класс связывания `FragmentResultBinding` все равно будет генерироваться связыванием данных, даже если связывание представлений отключено:



Для включения в макет одной или нескольких переменных связывания данных используются элементы `<data>` и `<variable>`:

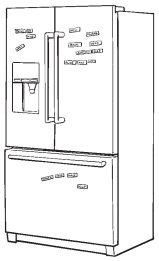
```
<layout ... >
    <data>
        <variable
            name="variableName"
            type="com.hfad.myapp.ClassName" />
    </data>

    ...

</layout>
```

Каждая переменная связывания данных размещается в отдельном элементе <variable>. Здесь используется только одна переменная, но ваш макет вполне может содержать несколько таких переменных.

Каждой переменной в коде Kotlin присваивается объект соответствующего типа. После присваивания представления макета могут использовать переменную связывания данных для обращения к свойствам и методам своего объекта.

Связывание представлений и связывание данных генерируют одинаковые классы связывания, но в разных ситуациях.

Связывание представлений генерирует класс связывания для каждого макета, тогда как связывание данных генерирует его для каждого макета с элементом <layout>.



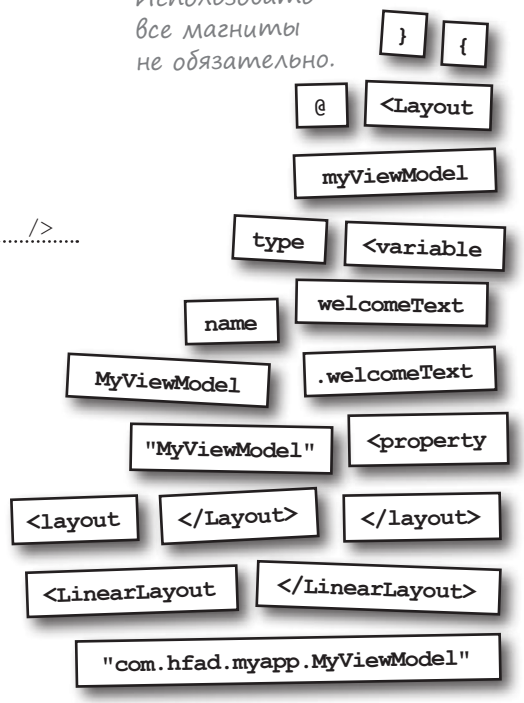
Развлечения с магнитами

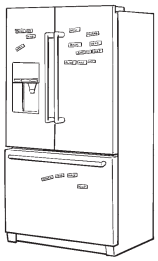
Кто-то выложил код макета на холодильнике, но от сильно хлопнувшей двери часть магнитов упала на пол. Удастся ли вам восстановить код?

Фрагмент должен использовать связывание данных для отображения свойства `welcomeText` из объекта `MyViewModel` в своем текстовом представлении `welcome`. Модель представления находится в пакете с именем `com.hfad.myapp.MyViewModel`.

```
.....

    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MyFragment">

    <data>

        .....

            .....="myViewModel"

            .....=  ..... />
    </data>


      .....
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/welcome"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"

            android:text="  ....." />

      .....

.....
```

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Кто-то выложил код макета на холодильнике, но от сильно хлопнувшей двери часть магнитов упала на пол. Удастся ли вам восстановить код?

Фрагмент должен использовать связывание данных для отображения свойства `welcomeText` из объекта `MyViewModel` в своем текстовом представлении `welcome`. Модель представления находится в пакете с именем `com.hfad.myapp.MyViewModel`.

```
<layout                                   ← Элемент <layout> становится корневым.
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MyFragment">

    <data>
        <variable                         ← Переменная связывания данных определяется в элементе <data>.
            name="myViewModel"            ← Переменная должна обладать именем и типом.
            type="com.hfad.myapp.MyViewModel" />
    </data>

    <LinearLayout                         ← Все представления выводятся в <LinearLayout>.
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/welcome"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{myViewModel.welcomeText}" />   ← Текст представления присваивается свойству welcomeText модели представления.

    </LinearLayout>
</layout>                                 ← Закрывающие теги для <LinearLayout> и <layout>.
```

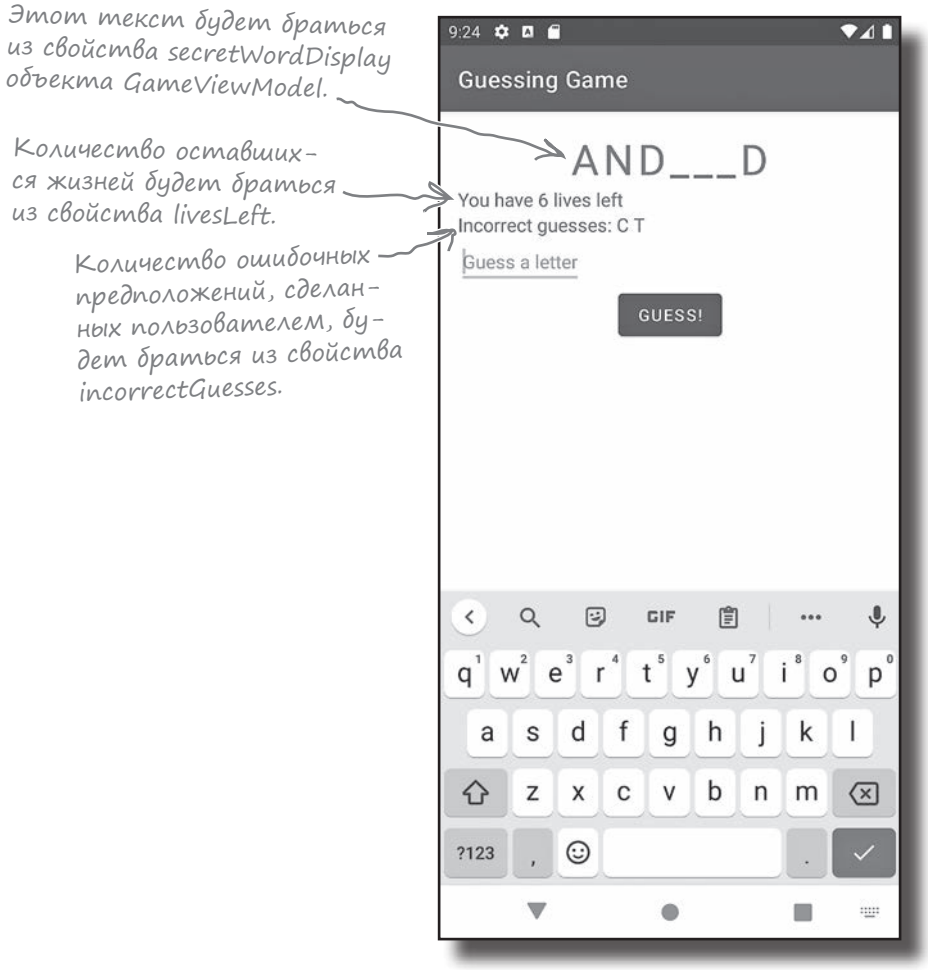
Эти магниты не понадобились.

<property> <Layout "MyViewModel"> welcomeText MyViewModel </Layout>

GameFragment тоже может использовать связывание данных

На данный момент мы реализовали связывание данных для фрагмента `ResultFragment`, чтобы текстовое представление из этого макета получало свой текст прямо из свойства result объекта `ResultViewModel`.

На следующем шаге то же будет сделано с `GameFragment`: мы реализуем связывание данных, чтобы текстовое представление в макете получало свои значения непосредственно из `GameViewModel`:



Для этого необходимо обновить код `GameFragment` и его макет. Начнем с кода макета.

Добавление элементов <layout> и <data> в fragment_game.xml

Как и в случае с `ResultFragment`, начнем с добавления элементов `<layout>` и `<data>` в макет `GameFragment`. Эти элементы указывают, что макет использует связывание данных, и определяют переменную связывания данных. Код выглядит так, как показано ниже; обновите файл *fragment_game.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".GameFragment">

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <LinearLayout
        ~~xmlns:android="http://schemas.android.com/apk/res/android"~~
        ~~xmlns:tools="http://schemas.android.com/tools"~~
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp"
        ~~tools:context=".GameFragment">~~

        ...

    </LinearLayout>
</layout>
```

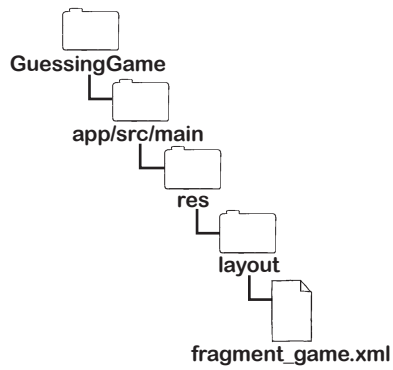
Элемент <layout> становится корневым элементом макета.

Эти строки перемещаются из элемента <LinearLayout>.

Добавляем переменную связывания данных.

Удалите эти строки и переместите их в элемент <layout>.

Закрывающий тег элемента <layout>.



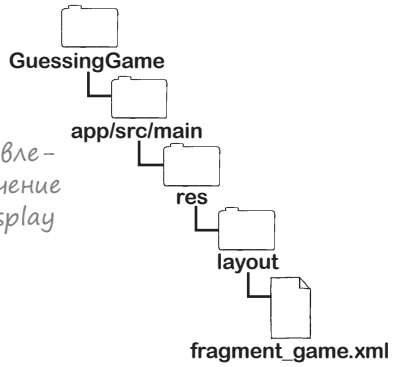
Как видите, в коде определяется переменная связывания данных с именем `gameViewModel` и типом `GameViewModel`. Прежде чем задавать значение этой переменной в коде Kotlin фрагмента, обновим представления макета, чтобы они получали свои значения из модели представления.

Использование переменной связывания данных для задания текста в макете

Как вы узнали ранее, представления могут получать данные напрямую от модели представления с использованием переменной связывания данных макета. Например, текстовое представление `word` из *fragment_game.xml* может отображать значение свойства `secretWordDisplay` объекта `GameViewModel` следующим кодом:

```
<TextView
    android:id="@+id/word"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:textSize="36sp"
    android:letterSpacing="0.1"
    android:text="@{gameViewModel.secretWordDisplay}" />
```

Свойству text представления присваивается значение свойства secretWordDisplay модели представления.



В двух представлениях должен выводиться дополнительный текст

Однако текстовые представления `lives` и `incorrect_guesses` не ограничиваются выводом значений свойств: **они включают дополнительный текст**. Например, в коде `GameFragment` в настоящее время текст представления `incorrectGuesses` задается следующим кодом:

Когда фрагмент GameFragment задает текст incorrect_guesses, в него включается дополнительный текст.

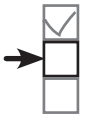
```
binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
```

Таким образом, мы не можем просто использовать свойство `incorrectGuesses` модели представления с использованием следующего кода, так как дополнительный текст в этом случае отображаться не будет:

```
<TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{gameViewModel.incorrectGuesses}" />
```

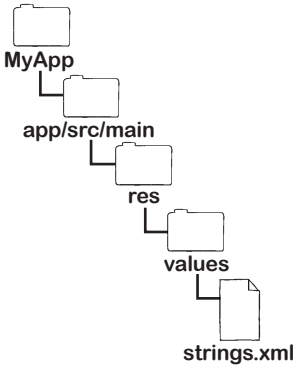
Этот код присваивает свойству text представления значение свойства incorrectGuesses и не включает дополнительный текст.

Чтобы решить эту проблему, мы воспользуемся строковым форматированием для передачи значения свойства строковому ресурсу. Давайте посмотрим, как это делается.



Снова о строковых ресурсах

Как вы узнали ранее в этой книге, включение строковых ресурсов в файл строковых ресурсов приложения позволяет избежать жестко фиксированного текста. Например, следующий код определяет строковый ресурс с именем `my_string`, который выводит текст «This is a String resource»:




```
<resources>
    ...
    <string name="my_string">This is a String resource</string>
</resources>
```

Строковые ресурсы могут получать аргументы

Также можно определить строковые ресурсы, получающие один или несколько аргументов. Это может быть удобно для выбора более сложного текста. Например, следующий ресурс использует обозначение **`%s`** для определения позиции, в которой должен быть вставлен переданный строковый аргумент:

```
<string name="hello">Hello, %s.</string>
```

Используйте для вывода персонализированного сообщения пользователю: например, «Hello, Ingo».

Используйте %s для строковых аргументов и %d для чисел.

А в этом примере **`%d`** сообщает, где должен располагаться строковый аргумент:

```
<string name="messages">You have %d messages.</string>
```

Например: «You have 10 messages».

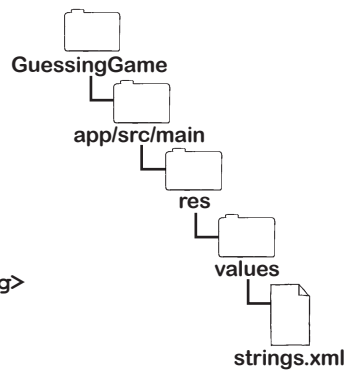
Строковому ресурсу можно передать несколько аргументов, для чего их следует пронумеровать. Например, следующий ресурс получает строку в первом аргументе и число во втором:

```
<string name="welcome">Hello, %1$s. You have %2$d messages.</string>
```

«Hello, Ingo. You have 10 messages».

Добавление двух новых строковых ресурсов в файл strings.xml

В приложении Guessing Game будут определены два новых строковых ресурса: для количества оставшихся жизней и для количества ошибочных предположений, сделанных пользователем. Откройте файл *strings.xml* в папке *app/src/main/res/values* и добавьте в него следующие два ресурса:




```
<resources>
    ...
    <string name="lives_left">You have %d lives left</string>
    <string name="incorrect_guesses">Incorrect guesses: %s</string>
</resources>
```

Обязательно добавьте оба строковых ресурса.

Итак, мы определили два строковых ресурса; используем их в макете `GameFragment`.

Макет может передавать параметры строковым ресурсам

Чтобы использовать два строковых ресурса, которые мы только что создали, необходимо передать каждому ресурсу параметр.

Передача параметров строковым ресурсам из кода макета происходит так:

```
android:text="@{@string/string_name(arg1, arg2)}"
```

Передает два параметра — arg1 и arg2 — строковому ресурсу с именем string_name.

где `string_name` — имя строкового ресурса, а `arg1` и `arg2` — два параметра. `arg1` определяет значение первого аргумента строки, а `arg2` — второго.

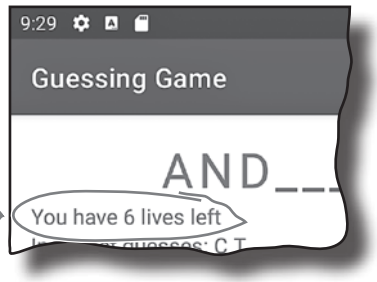
В приложении Guessing Game строковый ресурс `lives_left` определяется следующим образом:

```
<string name="lives_left">You have %d lives left</string>
```

Чтобы использовать его для отображения свойства `livesLeft` объекта `GameViewModel`, можно написать следующий код:

Количество оставшихся жизней выводится в отформатированной строке.

```
<TextView
    android:id="@+id/lives"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
```



Аналогичным образом определение строкового ресурса `incorrect_guesses` выглядит так:

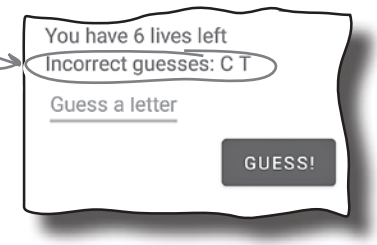
```
<string name="incorrect_guesses">Incorrect guesses: %s</string>
```

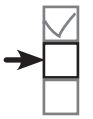
Следовательно, значение свойства `incorrectGuesses` может отображаться следующим кодом:

```
<TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
```

И это все, что необходимо знать для изменения макета `GameFragment`, чтобы в нем использовалось связывание данных. Давайте посмотрим, как выглядит полный код.

Отображает отформатированную строку с количеством ошибочных предположений пользователя.





Полный код fragment_game.xml

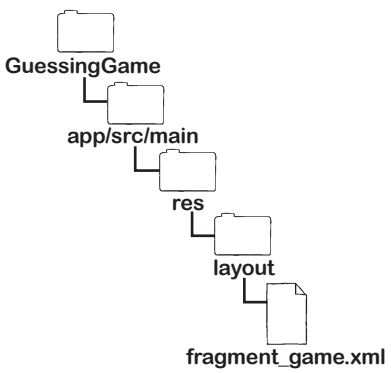
Ниже приведен полный код макета `GameFragment`; обновите файл *fragment_game.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".GameFragment">

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp"
        tools:context=".GameFragment">

        <TextView
            android:id="@+id/word"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:textSize="36sp"
            android:letterSpacing="0.1"
            android:text="@{gameViewModel.secretWordDisplay}" />
```

Мы собираемся использовать связывание данных, поэтому корневым элементом макета должен быть элемент <layout>.

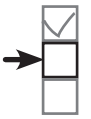
Определяет переменную связывания данных.

Удалите эти строки и переместите их в элемент <layout>.



Выводит значение свойства secretWordDisplay модели представления.

Продолжение на следующей странице.

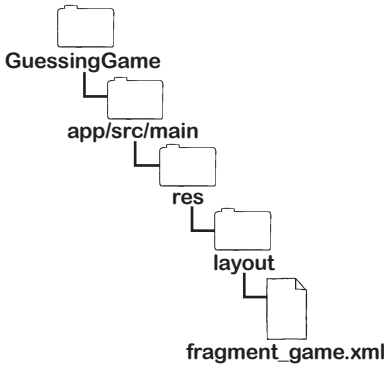


Полный код fragment_game.xml (продолжение)

```
    <TextView
        android:id="@+id/lives"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />

    <TextView
        android:id="@+id/incorrect_guesses"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />

    <EditText
        android:id="@+id/guess"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:hint="Guess a letter"
        android:inputType="text"
        android:maxLength="1" />

    <Button
        android:id="@+id/guess_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Guess!"/>
</LinearLayout>
</layout>
```

Для вывода оставшихся жизней и ошибочных предположений.

Закрывающий тег элемента <layout>



И это весь код, который необходимо изменить в файле макета. Но прежде чем проводить тест-драйв приложения, сначала необходимо обновить соответствующий фрагмент кода в *GameFragment.kt*.

Необходимо задать значение переменной gameViewModel

Первое, что необходимо сделать с кодом `GameFragment`, присвоить переменной связывания данных `gameViewModel` макета экземпляр модели представления фрагмента.

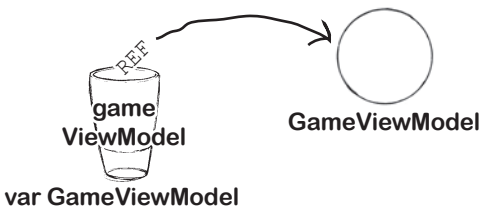
В настоящее время метод `onCreateView()` фрагмента `GameFragment` включает следующую строку, которая использует провайдер модели представления для получения объекта `GameViewModel`:

```
viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

Эту модель представления можно присвоить переменной связывания данных, для чего следует добавить следующую строку:

```
binding.gameViewModel = viewModel
```

После того как переменная `gameViewModel` будет связана с моделью представления подобным образом, представления макета смогут использовать его для получения своих данных.



Также необходимо включить использование Live Data механизмом связывания данных

Помимо присваивания переменной `gameViewModel`, необходимо включить использование Live Data в связывании данных макета.

Но в отличие от `ResultViewModel`, код `GameViewModel` использует Live Data, чтобы при обновлении значений свойств код `GameFragment` реагировал соответствующим образом. Например, следующий код из файла *GameFragment.kt* включает наблюдение за свойством `incorrectGuesses` и обновляет представление `incorrect_guesses` при обновлении свойства:

```
viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
    binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
})
```

Так как теперь в коде используется связывание данных, можно заставить каждое представление реагировать на изменения данных, включив следующую строку в код фрагмента:

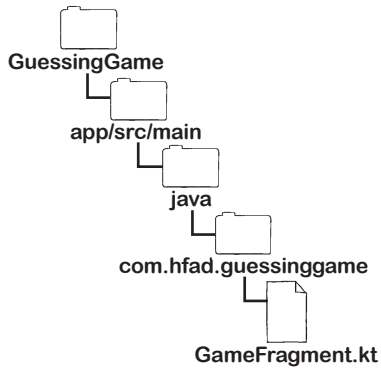
```
binding.lifecycleOwner = viewLifecycleOwner
```

Каждый раз, когда в модели представления обновляется значение свойства Live Data, макету уже не нужно полагаться на то, что код фрагмента обновит его представления. Макет может отреагировать на любые изменения без каких-либо вмешательств со стороны фрагмента.

И это все, что необходимо знать для обновления кода `GameFragment`. Давайте посмотрим, как он выглядит.

Полный код GameFragment.kt

Ниже приведен полный код `GameFragment`; обновите файл *GameFragment.kt* (изменения выделены жирным шрифтом):



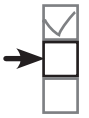

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer

class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        binding.gameViewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
            binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
        })
```

Присваивает значение переменной связывания данных.

Позволяет макету реагировать на обновления данных Live Data.

Удалите эти строки.

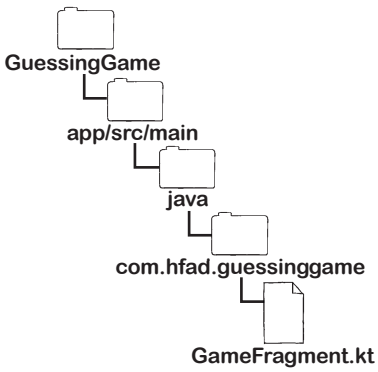
Продолжение на следующей странице.



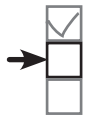
Полный код GameFragment.kt (продолжение)

Удалите эти строки.

```
        viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
            binding.lives.text = "You have $newValue lives left"
        })

        viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
            binding.word.text = newValue
        })

        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        })

        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
        }
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```



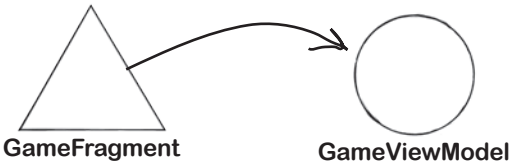
И это все изменения, которые необходимо внести в `GameFragment`. Давайте разберемся, что происходит при выполнении этого кода, и проведем тест-драйв приложения.



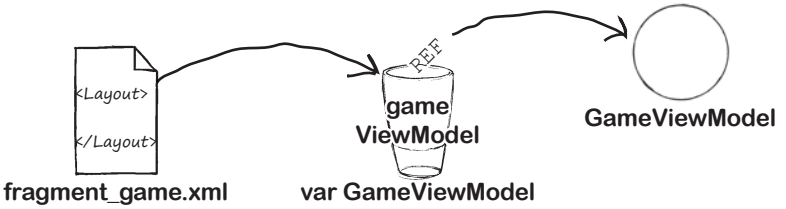
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

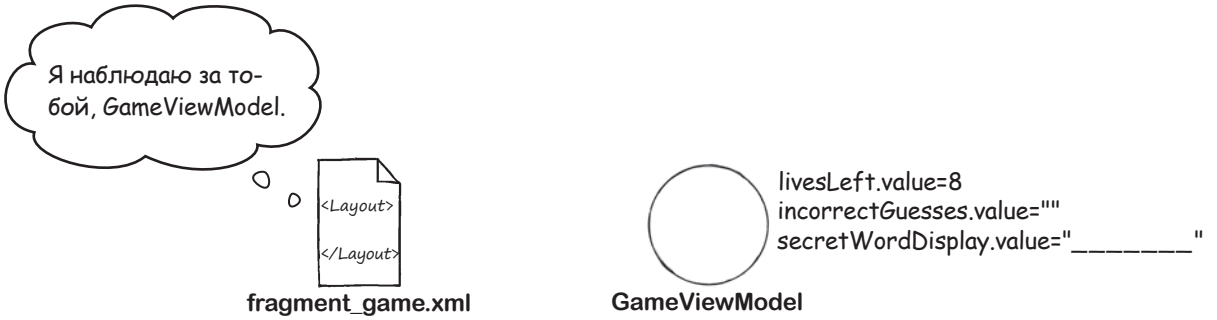
1. **GameFragment получает ссылку на свой объект GameViewModel.**



2. **GameFragment присваивает объект GameViewModel переменной связывания данных gameViewModel макета.**
Представления макета теперь могут использовать эту переменную для обращения к свойствам и методам модели представления.

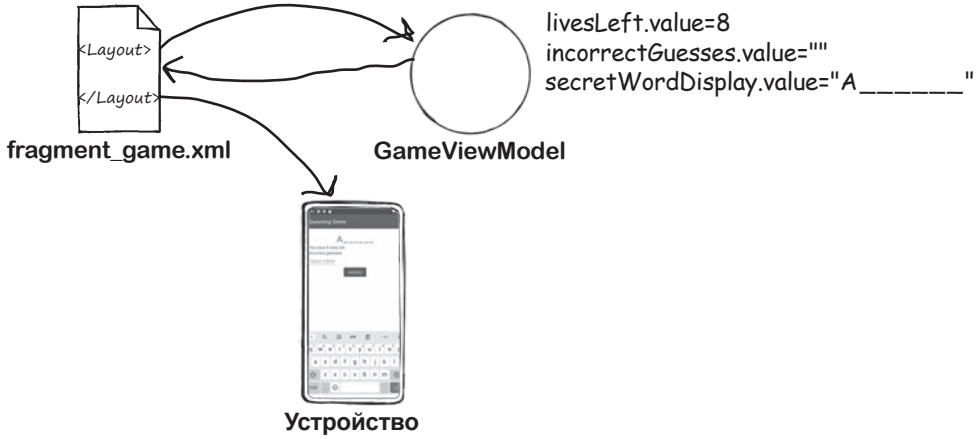


3. **GameFragment назначает владельца жизненного цикла макета.**
Макет теперь может наблюдать за свойствами Live Data модели представления и реагировать на любые изменения.

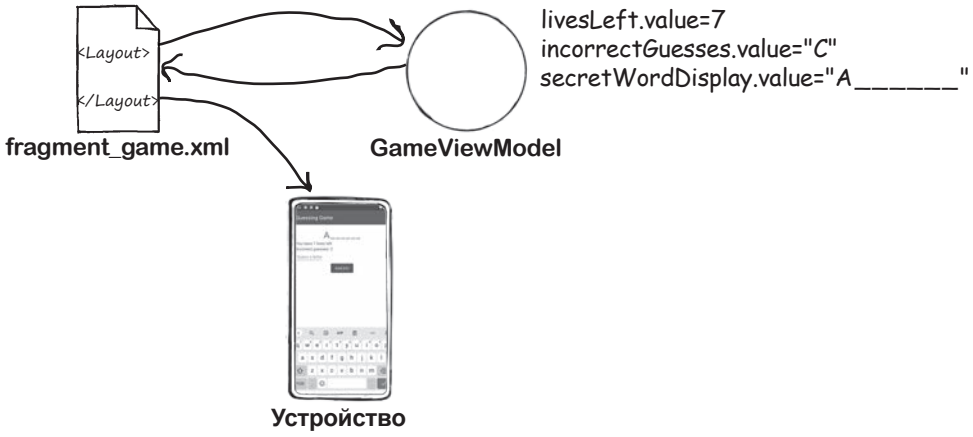


История продолжается

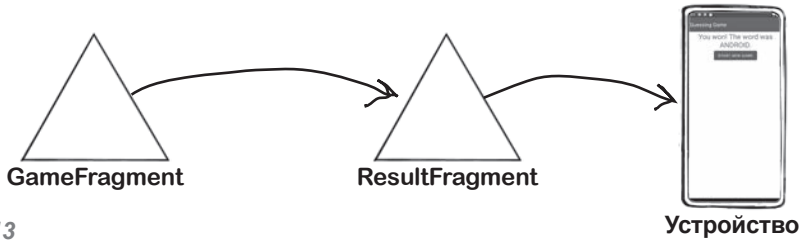
4 **Когда пользователь вводит правильное предположение, обновляется свойство secretWordDisplay объекта GameViewModel.**
Макет наблюдает за этим изменением, а в его представлении `word` отображается новый текст.



5 **Если пользователь вводит ошибочное предположение, обновляются свойства livesLeft и incorrectGuesses объекта GameViewModel.**
Макет наблюдает за изменениями, а в соответствующих представлениях выводится новый текст.

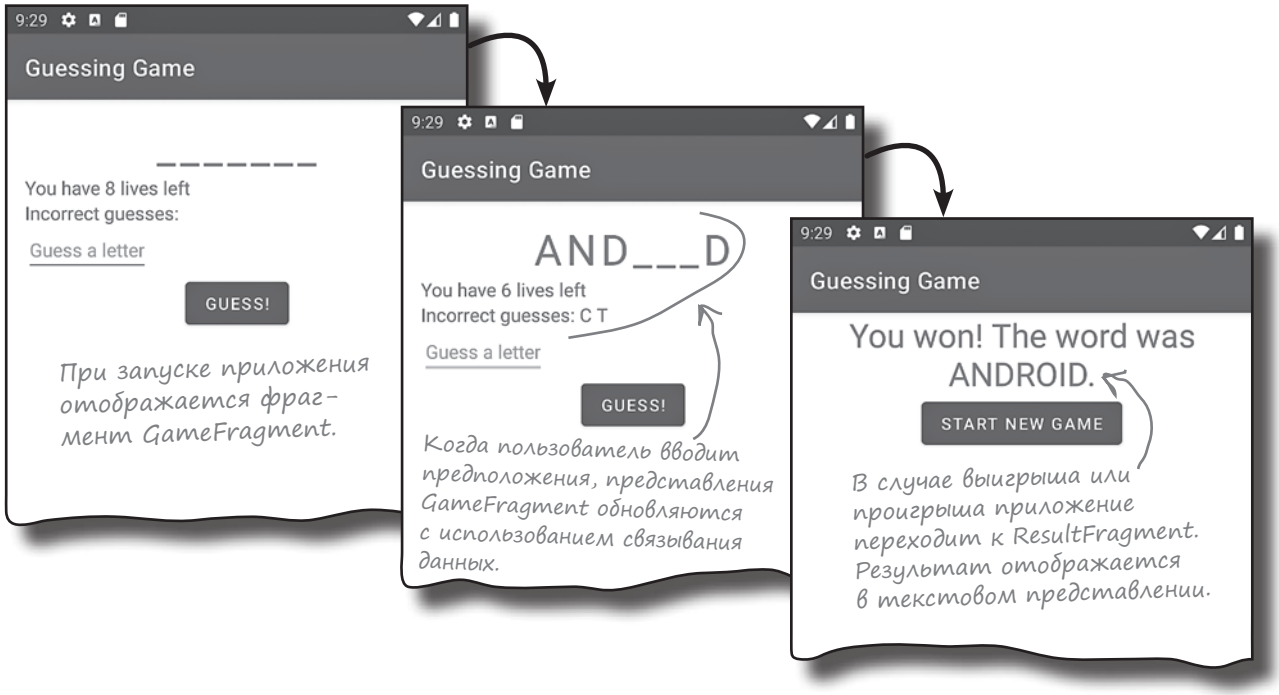


6 **Когда игра завершится, GameFragment переходит к ResultFragment.**



Тест-драйв

Приложение работает так же, как прежде. Однако в этой версии `GameFragment` использует связывание данных для получения значений свойств напрямую из `GameViewModel`.



Часто Задаваемые Вопросы

В: Можно ли использовать связывание данных для того, чтобы представления обновляли свойства в модели представления?

О: Да! Представления могут обновлять изменяемые свойства, для чего вместо `@` используется запись `@=`. Пример такого рода будет приведен в следующей главе.

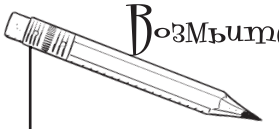
В: Свойству text приложения `word` присваивается значение `"@{gameViewModel.secretWordDisplay}"`. Но `secretWordDisplay` имеет тип `LiveData`, разве не следует вместо этого использовать `secretWordDisplay.value`?

О: Нет. Макет автоматически использует свойство `value`, давать ему команду это делать необязательно.

В: Я попытался присвоить свойству text представления `lives` значение `"@{gameViewModel.livesLeft}"`, но в приложении произошла ошибка. Вы не знаете почему?

О: Свойство `livesLeft.value` относится к типу `Int?`, а свойство `text` представления должно иметь тип `String`. Попробуйте преобразовать свойство в `String` следующей конструкцией: `"@{gameViewModel.livesLeft.toString()}"` Ошибки должны прекратиться.

Возьмите в руку карандаш



Приведенный ниже макет должен использовать строковый ресурс (с именем `msg`) для вывода текста в текстовом представлении: «Hello, `user`. The numbers are `numbersText`. You guessed `numberGuessed` right».

`user`, `numbersText` и `numberGuessed` — свойства `LotteryViewModel`; `user` имеет тип `String`, `numbersText` — тип `String`, а `numberGuessed` — тип `Int`.

Сможете ли вы написать код строкового ресурса?

Макет:

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".LotteryFragment">

    <data>
        <variable
            name="vm"
            type="com.hfad.lottery.LotteryViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/message"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/msg(vm.user, vm.numbersText, vm.numberGuessed)}" />
    </LinearLayout>
</layout>
```

Строковый ресурс:

```
<string name="msg">                                                    </string>
```

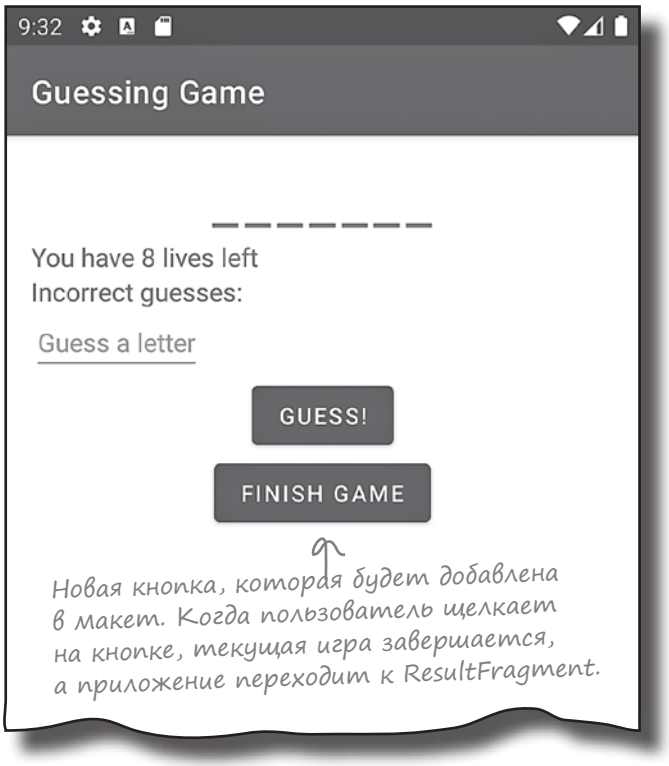
Ответ на с. 603.

Связывание данных может использоваться для вызова методов

К настоящему моменту мы реализовали связывание данных, чтобы представления получали свои значения прямо из модели представления, вместо того чтобы зависеть от кода фрагмента.

Связывание данных можно использовать и другим способом: кнопка может вызвать метод из модели представления *без написания дополнительного кода фрагмента*. Чтобы показать, как это делается, мы добавим в макет `GameFragment` новую кнопку Finish Game; щелчок на этой кнопке будет завершать игру.

Новая версия экрана будет выглядеть так:



Чтобы новая кнопка завершала игру, необходимо сделать следующее:

1. **Добавление нового метода finishGame() в GameViewModel.**
   Этот метод присваивает свойству `_gameOver` модели представления значение *true*. Когда это происходит, существующий код `GameFragment` реагирует на изменение переходом к `ResultFragment`.

2. **Добавление новой кнопки в макет, по щелчку на которой будет вызываться метод finishGame().**
   Кнопка будет вызывать метод с помощью связывания данных.

Начнем с обновления кода `GameViewModel`.

Добавление метода finishGame() в GameViewModel.kt

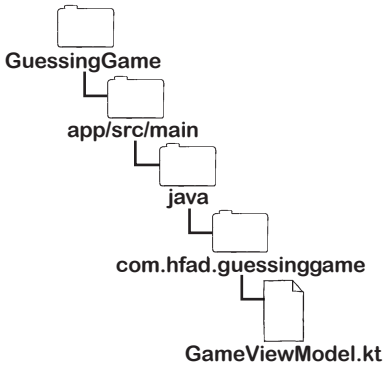
В код `GameViewModel` необходимо добавить новый метод `finishGame()`, который присваивает `_gameOver` значение *true*.

Вы уже знаете, как это делается; ниже приведен новый код *GameViewModel.kt*. Обновите код и включите новый метод, выделенный жирным шрифтом:

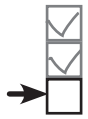
```
package com.hfad.guessinggame

...
```

Изменять код на этой странице не нужно.

```
class GameViewModel : ViewModel() {
    private val words = listOf("Android", "Activity", "Fragment")
    private val secretWord = words.random().uppercase()
    private val _secretWordDisplay = MutableLiveData<String>()
    val secretWordDisplay: LiveData<String>
        get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
        get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
        get() = _livesLeft
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver: LiveData<Boolean>
        get() = _gameOver

    init {
        _secretWordDisplay.value = deriveSecretWordDisplay()
    }

    private fun deriveSecretWordDisplay() : String {
        var display = ""
        secretWord.forEach {
            display += checkLetter(it.toString())
        }
        return display
    }
```



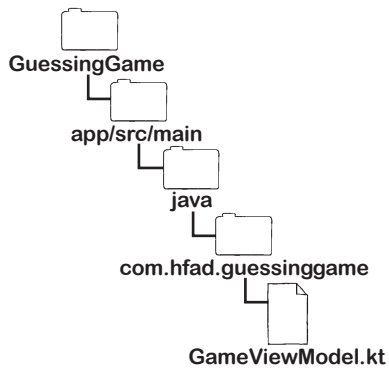
Продолжение на следующей странице.



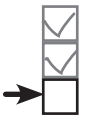
GameViewModel.kt (продолжение)

```
    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
        true -> str
        false -> "_"
    }

    fun makeGuess(guess: String) {
        if (guess.length == 1) {
            if (secretWord.contains(guess)) {
                correctGuesses += guess
                _secretWordDisplay.value = deriveSecretWordDisplay()
            } else {
                _incorrectGuesses.value += "$guess "
                _livesLeft.value = _livesLeft.value?.minus(1)
            }
            if (isWon() || isLost()) _gameOver.value = true
        }
    }

    private fun isWon() = secretWord.equals(secretWordDisplay.value, true)

    private fun isLost() = livesLeft.value ?: 0 <= 0

    fun wonLostMessage() : String {
        var message = ""
        if (isWon()) message = "You won!"
        else if (isLost()) message = "You lost!"
        message += " The word was $secretWord."
        return message
    }

    fun finishGame() {
        _gameOver.value = true
    }
}
```

Добавьте этот метод.



И это весь код, который необходимо добавить. На следующем шаге в макет `GameFragment` будет добавлена новая кнопка Finish Game, по щелчку на которой будет вызываться метод `finishGame()`.



ResultFragment
GameFragment
Добавление кнопки

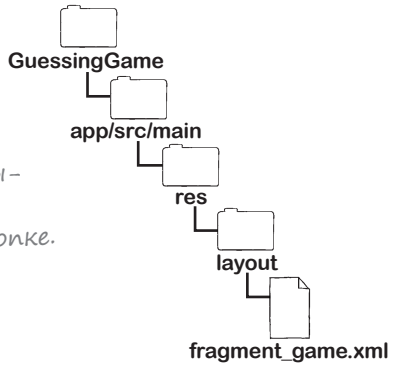
Использование связывания данных для вызова метода по щелчку на кнопке

Как было сказано выше, вы можете воспользоваться связыванием данных для того, чтобы по щелчку на кнопке вызывался метод его модели представления без написания дополнительного кода фрагмента. Для этого следует добавить к кнопке атрибут **`android:onClick`** и присвоить ему выражение для вызова метода.

В приложении Guessing Game в макет `GameFragment` нужно добавить новую кнопку, по щелчку на которой будет вызываться метод `finishGame()` объекта `GameViewModel`. Код кнопки выглядит так:

```
<Button
    android:id="@+id/finish_game_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Finish Game"
    android:onClick="@{() -> gameViewModel.finishGame()}" />
```

Обеспечивает вызов finishGame() по щелчку на кнопке.



Строка:

```
android:onClick="@{() -> gameViewModel.finishGame()}"
```

эквивалентна включению следующего кода во фрагмент:

```
binding.finishGameButton.setOnClickListener() {
    viewModel.finishGame()
}
```

кроме того, что на этот раз код фрагмента не потребуется. Во внутренней реализации связывание данных использует следующую строку:

```
android:onClick="@{() -> gameViewModel.finishGame()}"
```

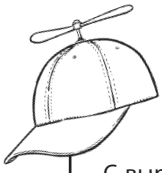
для создания слушателя `onClickListener` для кнопки. Когда пользователь щелкает на кнопке, выполняется код, который вызывает метод `finishGame()` модели представления.

Когда вы используете выражение связывания для вызова метода:

```
"@{() -> gameViewModel.finishGame()}"
```

выражение связывания иногда называют **связыванием слушателя**.

И это все, что необходимо знать для включения кнопки в макет `GameFragment` и ее реагирования на щелчки. Посмотрим, как выглядит полный код.



Серьезное программирование

С выражениями связывания можно сделать еще много интересного. Дополнительную информацию можно найти по адресу

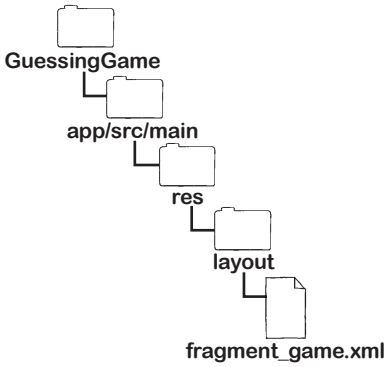
https://developer.android.com/topic/libraries/data-binding/expressions

Полный код fragment_game.xml

Ниже приведен полный код макета `GameFragment`; обновите код *fragment_game.xml* и включите в него новую кнопку (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".GameFragment">

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp">

        <TextView
            android:id="@+id/word"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:textSize="36sp"
            android:letterSpacing="0.1"
            android:text="@{gameViewModel.secretWordDisplay}" />

        <TextView
            android:id="@+id/lives"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
```

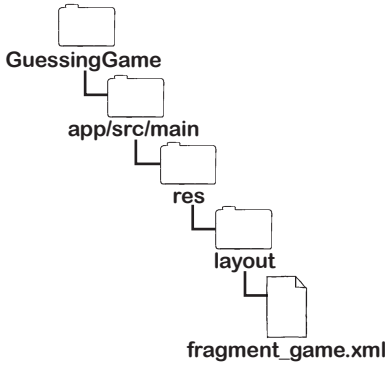
Код на этой странице изменять не нужно.



Продолжение на следующей странице.

Полный код fragment_game.xml (продолжение)

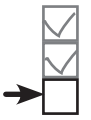
```
        <TextView
            android:id="@+id/incorrect_guesses"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />

        <EditText
            android:id="@+id/guess"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:hint="Guess a letter"
            android:inputType="text"
            android:maxLength="1" />

        <Button
            android:id="@+id/guess_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Guess!"/>
```



Добавляет новую кнопку.

```
        <Button
            android:id="@+id/finish_game_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Finish Game"
            android:onClick="@{() -> gameViewModel.finishGame()}" />
    </LinearLayout>
</layout>
```

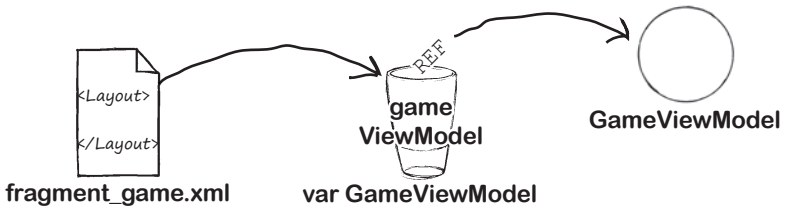
Посмотрим, что происходит во время выполнения кода.



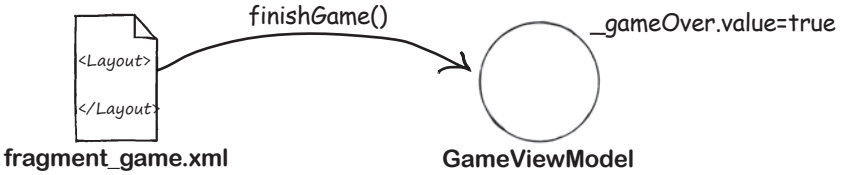
Что происходит при выполнении приложения

При выполнении приложения происходит следующее:

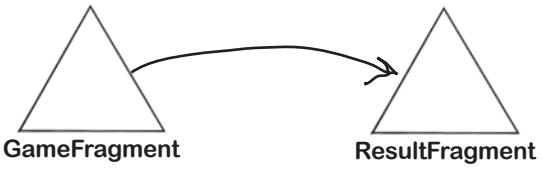
1. **GameFragment присваивает объект GameViewModel переменной связывания данных gameViewModel макета.**
Представления макета теперь могут использовать переменную для обращения к свойствам и методам модели представления.



2. **Пользователь щелкает на кнопке Finish Game.**
Кнопка вызывает метод `finishGame()` объекта `GameViewModel`, который присваивает свойству `_gameOver` значение *true*.



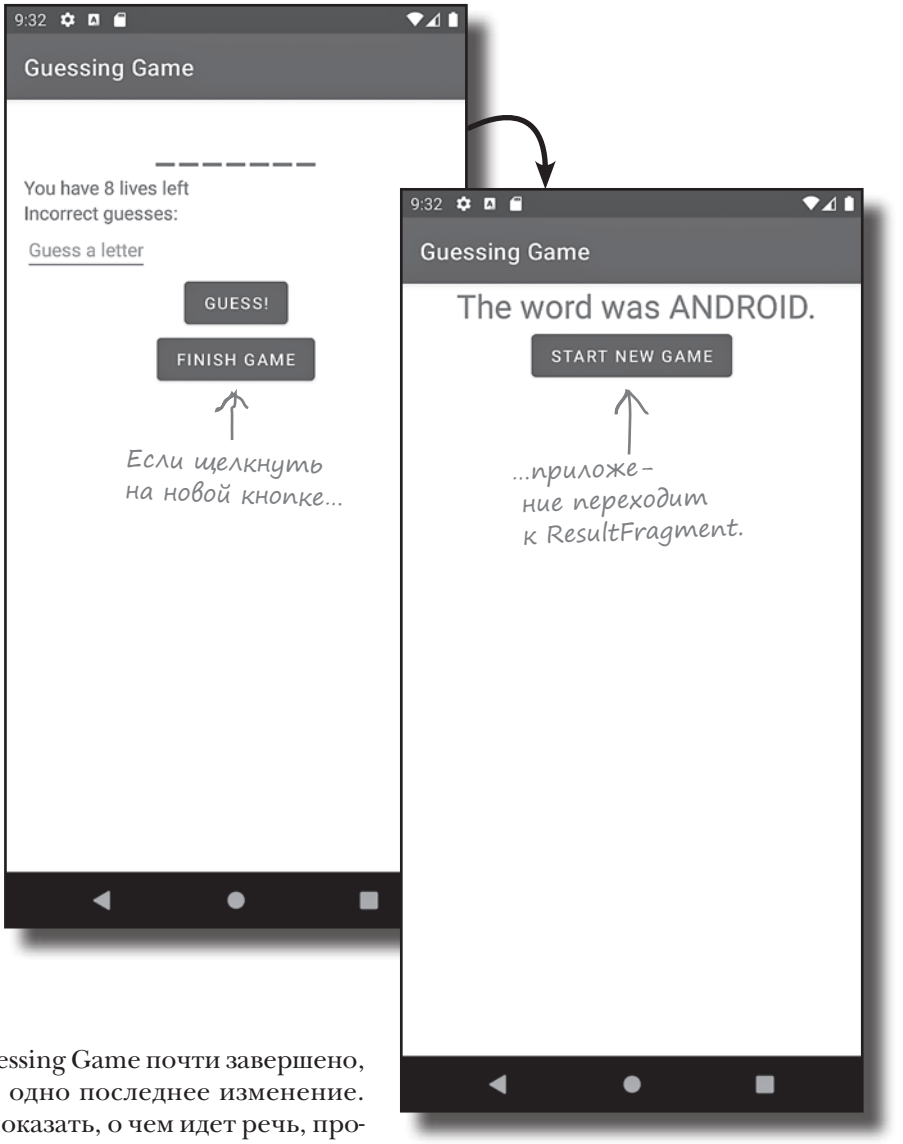
3. **GameFragment обнаруживает, что значение _gameOver изменилось.**
Фрагмент реагирует переходом к `ResultFragment`.



Проведем тест-драйв приложения.

Тест-драйв

При запуске приложения в макете `GameFragment` появляется новая кнопка Finish Game. По щелчку на кнопке приложение переходит к фрагменту `ResultFragment`, в котором выводится загаданное слово.



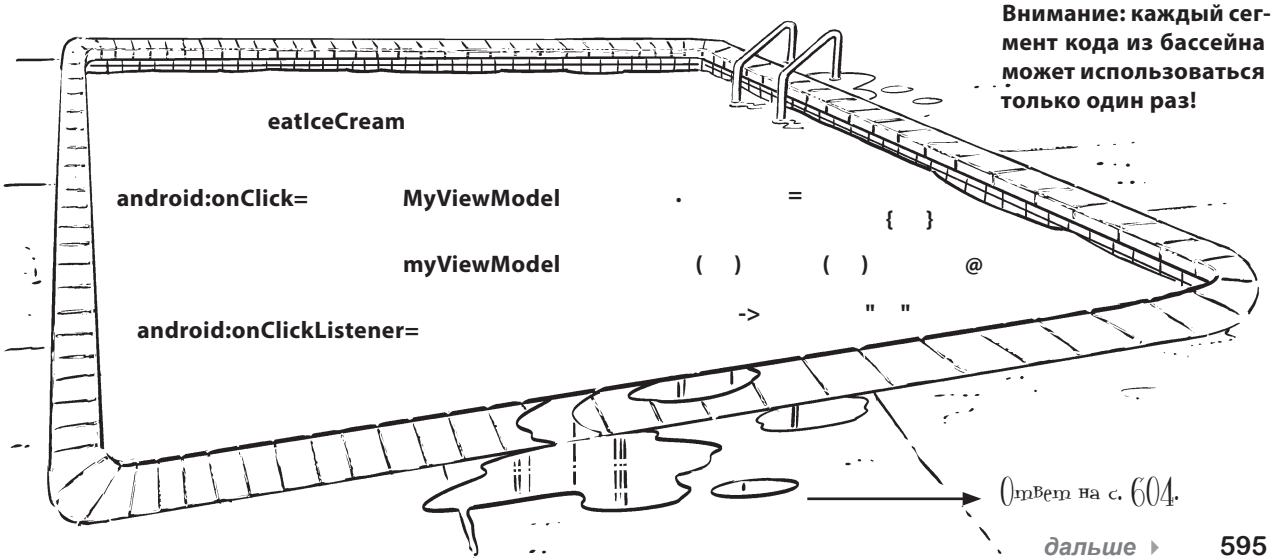
Приложение Guessing Game почти завершено, осталось внести одно последнее изменение. Но прежде чем показать, о чем идет речь, проверьте свои силы на следующем упражнении.

У бассейна

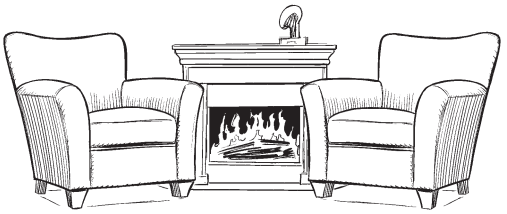
Обновите приведенный ниже код макета, чтобы по щелчку на кнопке вызывался метод `eatIceCream()` класса `MyViewModel`. Выловите сегменты кода из бассейна и расставьте их в пропусках. Каждый сегмент может использоваться только один раз; использовать все сегменты не обязательно.

```
<layout ...>
    <data>
        <variable
            name="myViewModel"
            type="com.hfad.myapp.MyViewModel" />
    </data>

    <LinearLayout ...>

       <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Eat Ice Cream"
             ............................................................................................/>.....

    </LinearLayout>
</layout>
```

Внимание: каждый сегмент кода из бассейна может использоваться только один раз!



Ответ на с. 604.



Беседа у камина

Тема: связывание представлений и связывание данных обсуждают свои различия

Связывание данных:

Связывание представлений, надо поговорить.

Наверняка ты замечаешь, как часто нас путают. Но хотя в наших именах присутствует «связывание», это еще не означает, что у нас много общего.

Ты так думаешь?

Возможно, но мы делаем это в совершенно разных обстоятельствах.

Можно подумать, ты спас всех от `findViewById`. Это заслуга класса связывания, а не твоя. Кроме того, *я тоже* генерирую такой класс.

Кроме того, генерировать класс связывания для каждого макета крайне неэффективно.

Связывание представлений:

Правда? О чем?

Признай, что имена действительно похожи. Кроме того, у нас много общего.

Да! Мы оба генерируем классы связывания. *Одинаковые* классы связывания. И не притворяйся, что это не так.

По правде говоря, я намного удобнее и полезнее. Я генерирую класс связывания для каждого макета, и в большинстве случаев это именно то, что вам нужно. Разработчики мне благодарны. Я даже получаю письма от фанатов. «Я твой самый большой поклонник! Спасибо, что избавил меня от `findViewById`. Я думал, эта боль никогда не завершится!»

Какая разница...

Строго говоря, я игнорирую все макеты, у которых атрибут `viewBindingIgnore` содержит *true*. Такой подход намного проще, чем у тебя.

Связывание данных:

У меня сложно? Просто я не такой неразборчивый, как ты. Я генерирую классы связывания только для макетов с элементом `<layout>`. При таком подходе разработчик может решить, когда именно использовать меня.

Немножко дополнительного кода, но я могу делать то, о чем ты можешь только мечтать. Например, я позволяю представлениям получать значения прямо из модели представления.

Так в этом и суть. Ты зависишь от кода фрагмента, который сообщает представлениям, что они должны делать, — а со мной это не обязательно. Я позволяю значениям получать данные напрямую, что означает упрощение кода фрагмента. Я даже позволяю им использовать свойства Live Data, чтобы они могли автоматически обновляться.

Нет, не нужен. Я использую двустороннее связывание, в котором представления могут обновлять значения *и* реагировать на изменения. Присмотрись повнимательнее, это изменит твою жизнь.

Ну конечно. А почему нет?

Я бы не назвал простые вызовы методов «безответственными», и это избавляет разработчика от написания лишнего кода фрагмента. Но я соглашусь с тем, что в сложных ситуациях код должен находиться во фрагменте, а не в макете.

Ты просто завидуешь.

Связывание представлений:

Да, сложно! Кому захочется включать еще больше разметки XML в код макета? Столько лишней разметки!

Я это тоже могу. Через код фрагмента.

А как насчет обновления значений свойств? Уверен, что для этого тебе нужен код фрагмента.

Ха. Ты еще скажи, что ты позволяешь представлениям вызывать методы.

Это приводит к чрезмерному усложнению кода макета. Не думал, что ты будешь настолько безответственным.

Выходит, даже ты признаешь, что твои возможности небезграничны.

ResultFragment
GameFragment
Добавление кнопки

Связывание представлений можно отключить

Обновление приложения Guessing Game почти завершено, но осталось внести еще одно изменение, чтобы в нем генерировались только те классы связывания, которые используются для связывания данных.

Как вы знаете, связывание данных и связывание представлений генерируют одни и те же классы связывания, но в разных обстоятельствах: связывание представлений генерирует класс связывания для *каждого* макета, тогда как связывание данных генерирует класс для каждого макета с корневым элементом `<layout>`.

В приложении Guessing Game классы связывания используются только для файлов *fragment_game.xml* и *fragment_result.xml*. Так как оба файла включают элемент `<layout>`, связывание представлений можно спокойно отключить, а его классы связывания по-прежнему будут генерироваться механизмом связывания данных:

Связывание данных не генерирует класс связывания только для файла activity_main.xml, потому что он не включает элемент <layout>. Этот класс связывания использоваться не будет, так что связывание представлений можно безопасно отключить.

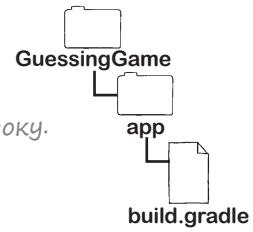
Связывание данных генерирует эти два файла за нас, так как файлы макетов содержат корневой элемент <layout>.

Отключение связывания представлений в файле build.gradle приложения

Чтобы отключить связывание представлений, откройте файл *GuessingGame/app/build.gradle* и удалите соответствующую строку из раздела `buildFeatures`:

```
android {
    ...
    buildFeatures {
        viewBinding true
        dataBinding true
    }
}
```

Удалите эту строку.

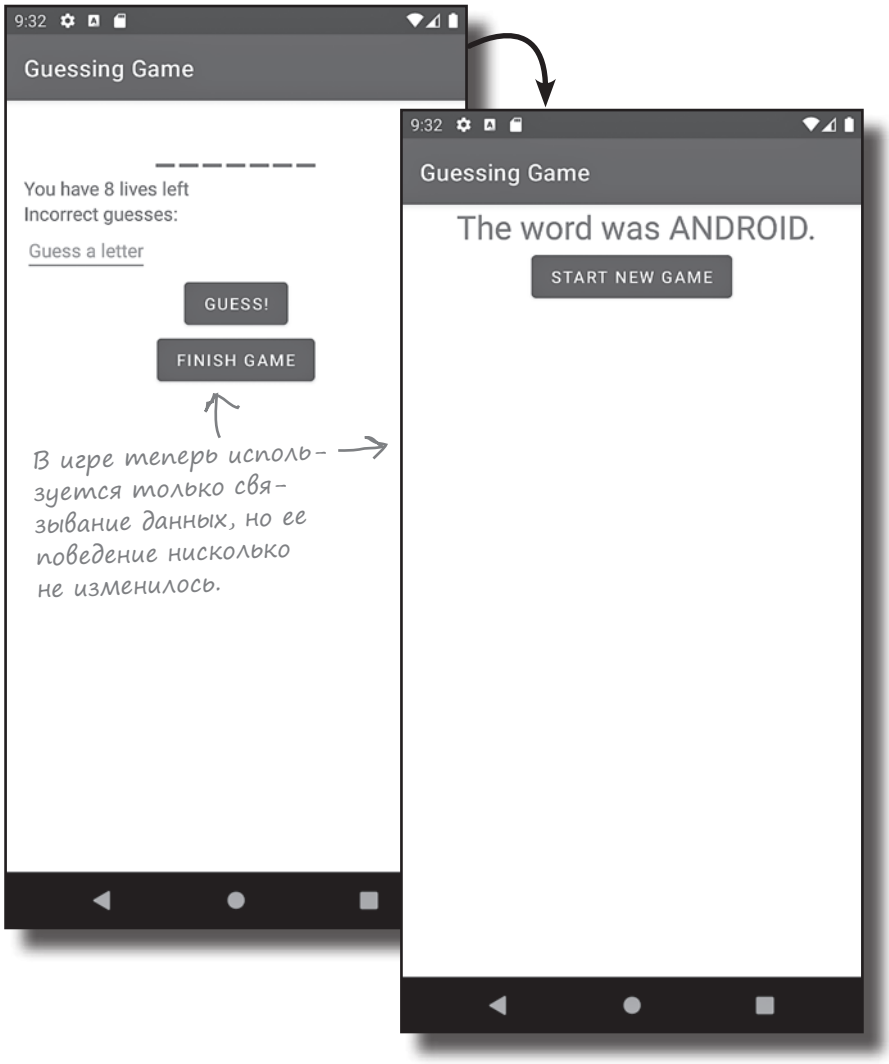


Затем щелкните на ссылке Sync Now, чтобы синхронизировать изменения с остальными частями проекта.

Давайте проведем быстрый тест-драйв приложения и убедимся в том, что оно продолжает успешно работать.

Тест-драйв

Игра работает точно так же, как и прежде. Отключение связывания представлений ни на что не повлияло, потому что связывание данных сгенерировало все классы связывания, необходимые для работы приложения.

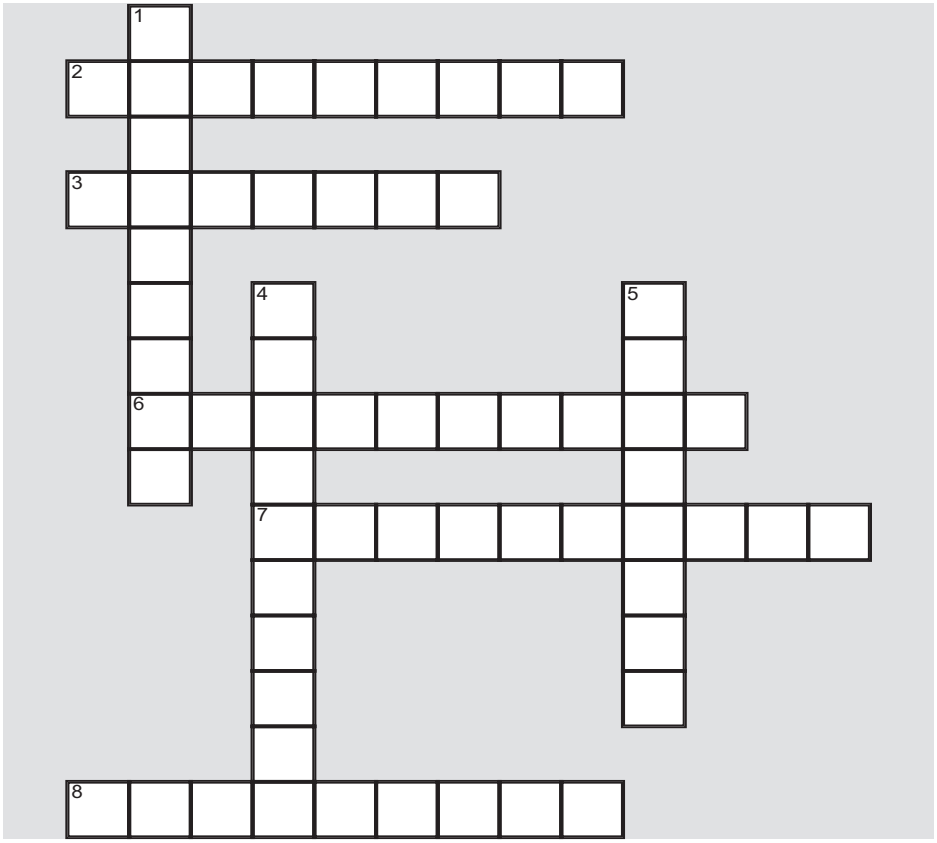


Поздравляем! Вы освоили связывание данных и научились использовать его в сочетании с моделями представлений и Live Data. Этот подход способствует упрощению кода фрагментов и помогает строить динамичные приложения, быстро реагирующие на действия пользователя.

Кроссворд

Каждое из определений следующего кроссворда представляет собой выражение связывания, используемое в свойстве android:text представления TextView. Сможете ли вы решить кроссворд по результату каждого определения?

Подсказка: в каждом выражении связывания используется строковый ресурс и модель представления. Их код приведен на следующей странице — вместе с кодом макета, включающим TextView.



По горизонтали

2 "@{@string/string2(vm.d[l])}"

3 "@{@string/string3(vm.d[2], vm.g[0])}"

6 "@{@string/stringl(vm.a)}"

7 "@{@string/string3(vm.f[l], vm.h[0])}"

8 "@{@string/stringl(vm.d[vm.i[l]])}"

По вертикали

1 "@{@string/string3(vm.e[2], vm.g[0])}"

4 "@{@string/string4(vm.h[2], vm.a)}"

5 "@{@string/string4{vm.h[l], vm.g[2])}"

Макет:

```
<?xml version="1.0" encoding="utf-8"?>
<layout...>

    <data>
        <variable
            name="vm"
            type="com.hfad.crossword.CrosswordViewModel" />
    </data>

    <LinearLayout...>
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text= ..../>....
    </LinearLayout>
</layout>
```

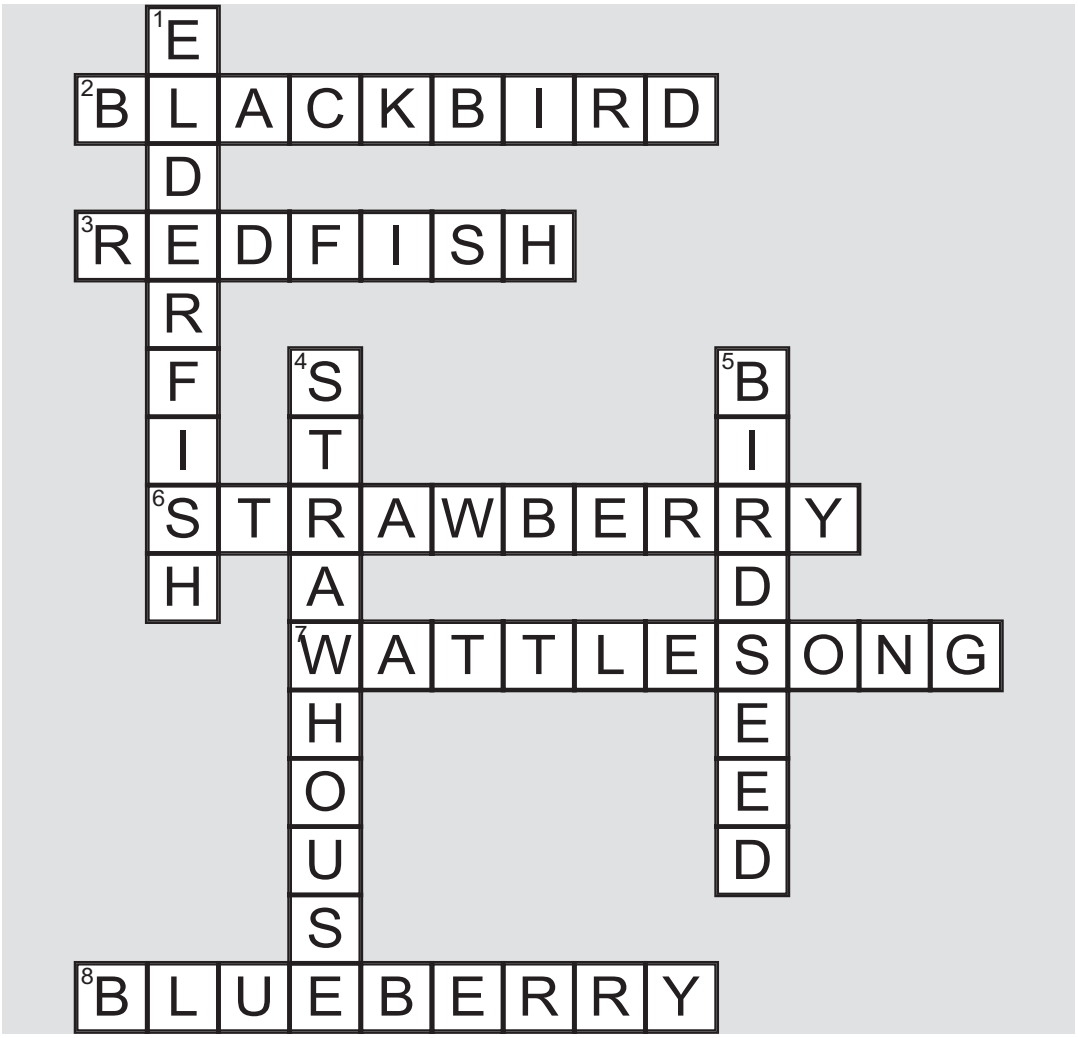
Модель представления:

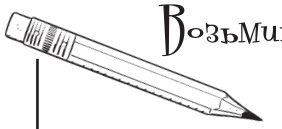
```
...
class CrosswordViewModel : ViewModel() {
    val a = "STRAW"
    val b = "RASP"
    val c = "CRAN"
    val d = listOf("BLUE", "BLACK", "RED")
    val e = listOf("GOOSE", "MOOSE", "ELDER")
    val f = listOf("LINGON", "WATTLE", "WEAVER")
    val g = listOf("FISH", "FLY", "BIRD", "HORSE")
    val h = listOf("SONG", "SEED", "HOUSE")
    val i = listOf(2,0,1,2,0)
}
```

Файл строкового ресурса:

```
<resources>
    <string name="app_name">Crossword</string>
    <string name="string1">%sBERRY</string>
    <string name="string2">%sBIRD</string>
    <string name="string3">%1$s%2$s</string>
    <string name="string4">%2$s%1$s</string>
</resources>
```

Кроссворд. Решение





Возьмите в руку карандаш
Решение

Приведенный ниже макет должен использовать строковый ресурс (с именем `msg`) для вывода текста в текстовом представлении: «Hello, `user`. The numbers are `numbersText`. You guessed `numberGuessed` right».

`user`, `numbersText` и `numberGuessed` — свойства `LotteryViewModel`; `user` имеет тип `String`, `numbersText` — тип `String`, а `numberGuessed` — тип `Int`.

Сможете ли вы написать код строкового ресурса?

Макет:

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".LotteryFragment">

    <data>
        <variable
            name="vm"
            type="com.hfad.lottery.LotteryViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/message"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:text="@{@string/msg(vm.user, vm.numbersText, vm.numberGuessed)}" />
    </LinearLayout>
</layout>
```

Строковый ресурс должен получать следующие три аргумента.

Строковый ресурс:

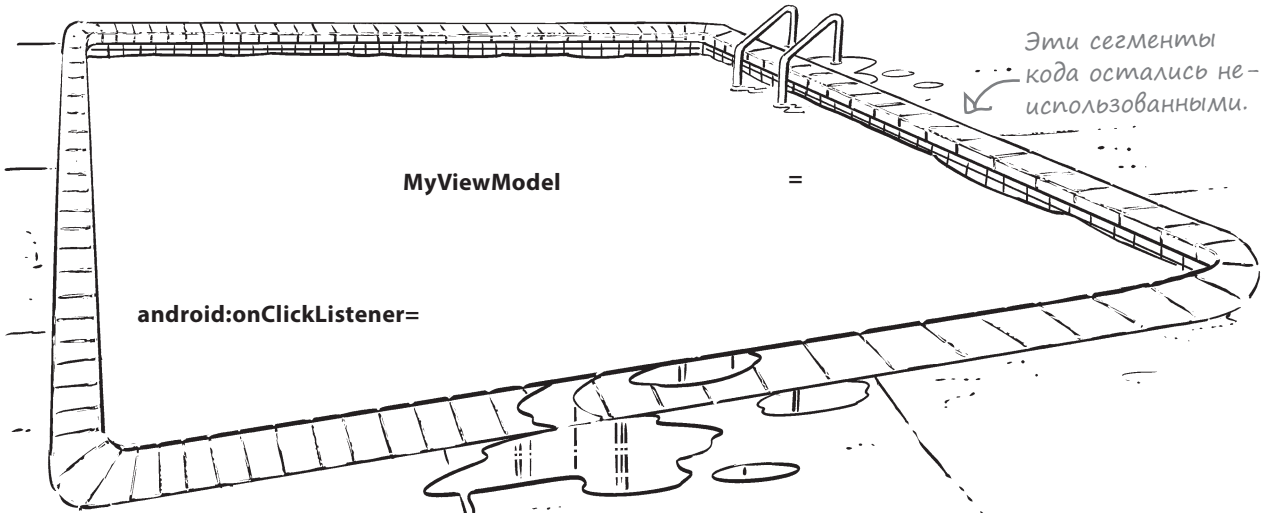
```
<string name="msg">Hello, %1$s. The numbers are %2$s. You guessed %3$d right.</string>
```

У бассейна. Решение

Обновите приведенный ниже код макета, чтобы по щелчку на кнопке вызывался метод `eatIceCream()` класса `MyViewModel`. Выловите сегменты кода из бассейна и расставьте их в пропусках. Каждый сегмент может использоваться только один раз; использовать все сегменты не обязательно.

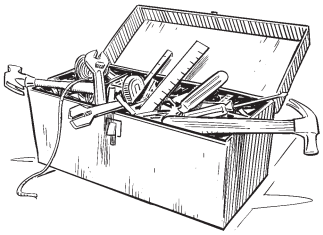
```
<layout ...>
    <data>
        <variable
            name="myViewModel"
            type="com.hfad.myapp.MyViewModel" />
    </data>

    <LinearLayout ...>

       <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Eat Ice Cream"
            android:onClick= " @ { ( ) -> myViewModel . eatIceCream ( ) } "/>

    </LinearLayout>
</layout>
```

Это выражение использует связывание данных для вызова метода eatIceCream() модели представления по щелчку на кнопке.



Эти сегменты кода остались неиспользованными.

Ваш инструментарий Android



Глава 13 осталась позади, а ваш инструментарий пополнился связыванием данных.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Связывание данных предоставляет прямой доступ к свойствам и методам.
- Связывание данных включается в файле *build.gradle* приложения.
- Чтобы использовать связывание данных, включите корневой элемент `<layout>` в каждый макет, который должен использовать связывание данных.
- Связывание данных генерирует класс связывания для каждого макета с корневым элементом `<layout>`. Этот класс связывания не отличается от того, который генерируется для связывания представлений.
- Используйте элементы `<data>` и `<variable>` для определения переменных.
- Строковые ресурсы могут получать аргументы.
- Связывание данных позволяет реагировать на обновление значений свойств Live Data.

Номер с видом

В большинстве приложений используются данные. Но если вы не позаботитесь о том, чтобы эти данные были где-то сохранены, **данные будут навсегда потеряны** при закрытии приложения. В мире приложений Android информация обычно ***хранится в базах данных***, поэтому в этой главе мы представим **библиотеку хранения данных Room**. Вы научитесь **строить базы данных**, **создавать таблицы** и **определять методы доступа к данным** с использованием аннотаций классов и интерфейсов. Вы узнаете, как **использовать сопрограммы** для выполнения кода баз данных в фоновом режиме. Заодно вы научитесь **преобразовывать данные Live Data при их изменении с помощью *Transformations.map()***.

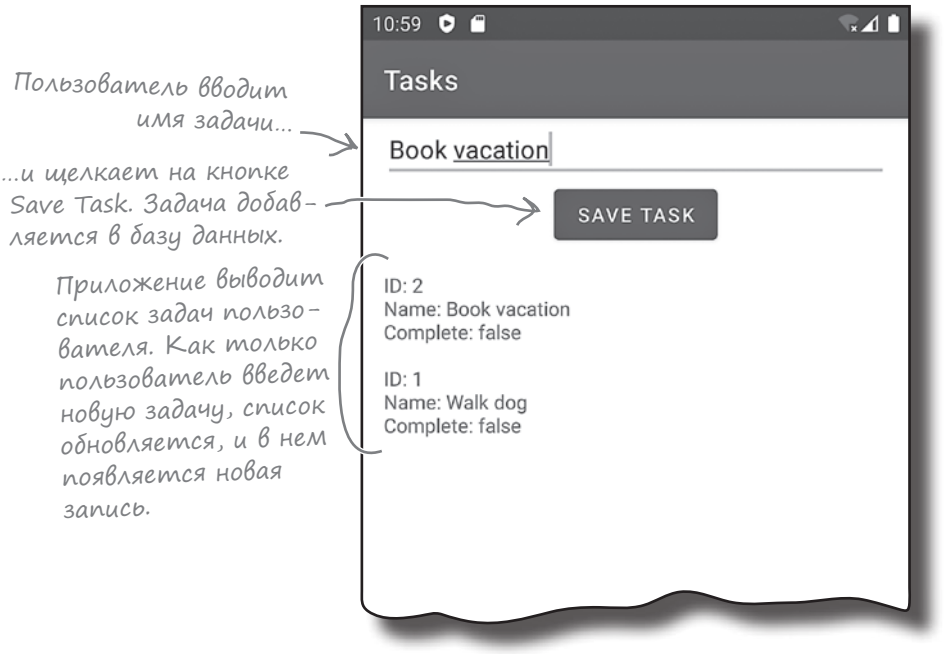
Многим приложениям требуется хранить данные

Почти во всех приложениях, написанных нами ранее, использовались небольшие объемы статических данных. Например, в приложении Guessing Game, построенном в главах с 11-й по 13-ю, в модели представления хранился статический массив строк, из которого игра случайным образом выбирала загаданное слово.

Однако в реальном мире приложения обычно не ограничиваются статическими данными; они должны иметь возможность сохранять данные, которые могут изменяться, чтобы изменения не терялись при закрытии приложения. Например, приложение-проигрыватель может хранить списки воспроизведения, а игра может хранить информацию о текущем состоянии игрока, чтобы при очередном запуске он мог продолжить игру с того места, где она была остановлена.

Приложения могут хранить информацию в базе данных

Как правило, данные пользователя удобнее всего хранить в базе данных, поэтому в этой главе вы научитесь работать с базой данных на примере приложения Tasks. В этом приложении пользователь может добавлять задачи в базу данных и выводить список всех уже введенных задач. Приложение будет выглядеть так:

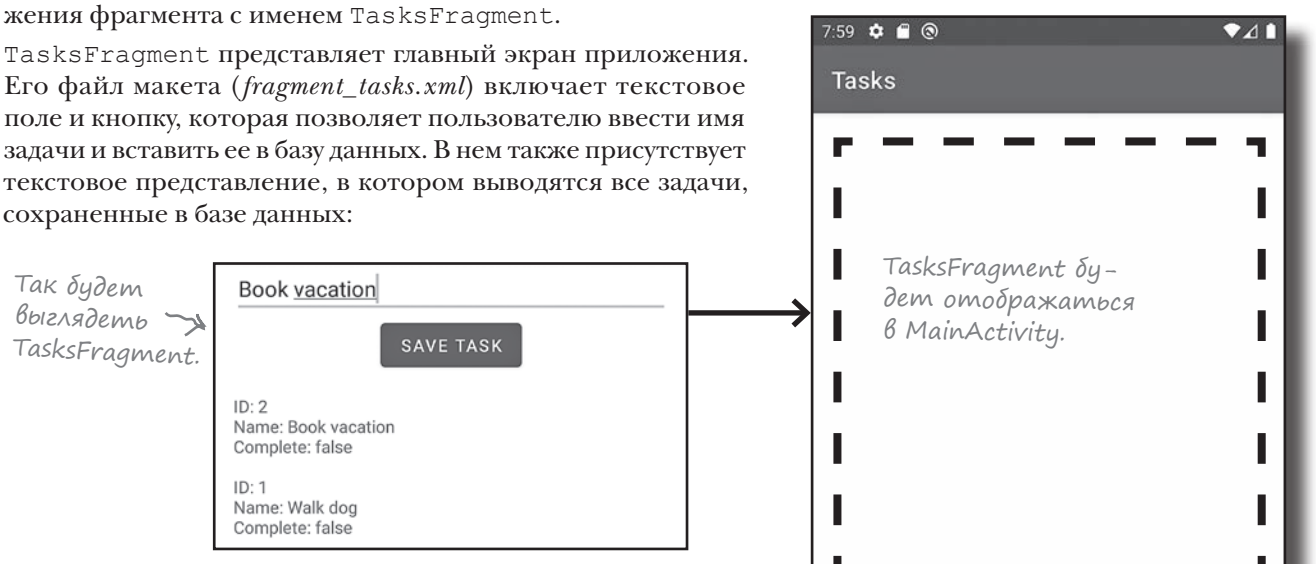


Прежде чем начать строить приложения, стоит разобраться в его структуре.

Структура приложения

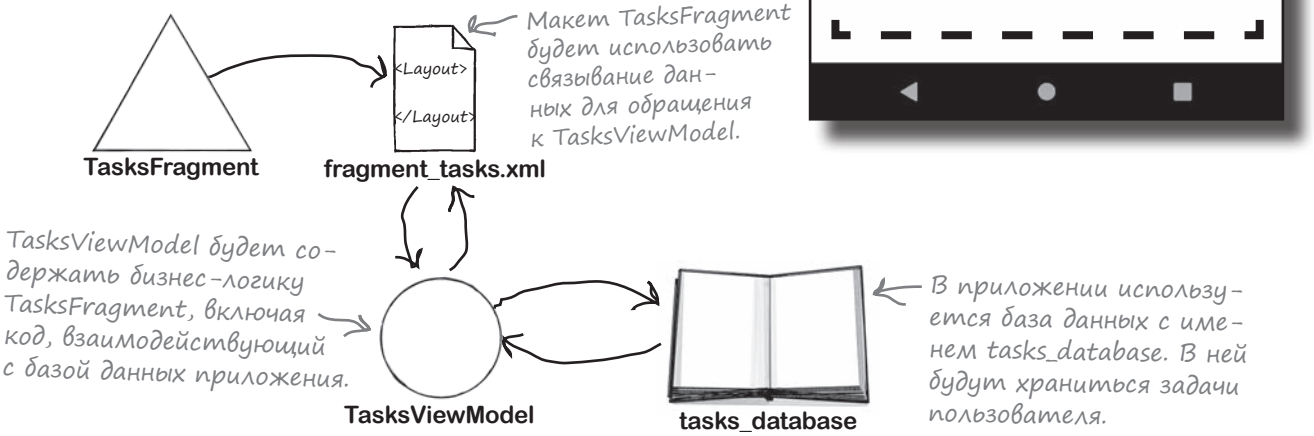
Приложение содержит одну активность (с именем `MainActivity`), которая будет использоваться для отображения фрагмента с именем `TasksFragment`.

`TasksFragment` представляет главный экран приложения. Его файл макета (*fragment_tasks.xml*) включает текстовое поле и кнопку, которая позволяет пользователю ввести имя задачи и вставить ее в базу данных. В нем также присутствует текстовое представление, в котором выводятся все задачи, сохраненные в базе данных:



Мы также добавим в приложение модель представления (с именем `TasksViewModel`), которая будет использоваться фрагментом `TasksFragment` для бизнес-логики. Она включает свойства и методы, которые используются фрагментом для взаимодействия с базой данных приложения. Мы также включим связывание данных, чтобы макет `TasksFragment` мог напрямую обращаться к модели представления.

Эти компоненты будут взаимодействовать по следующей схеме:



Для создания базы данных будет использоваться библиотека Android ***Room***. Что же собой представляет Room?

Room — библиотека баз данных, работающая на базе SQLite

Room также является частью Android Jetpack.

Во внутренней реализации многие базы данных Android используют SQLite. SQLite — упрощенная, стабильная, быстрая и оптимизированная для однопользовательского доступа система, и все эти особенности делают ее хорошим кандидатом для приложений Android. Тем не менее написание кода для создания, управления и взаимодействия с базами данных SQLite может быть непростым делом.

Для упрощения задачи в Android Jetpack включена библиотека хранения данных ***Room***, которая работает на базе SQLite. С Room вы пользуетесь всеми преимуществами SQLite, но с более простым кодом. Например, библиотека поддерживает удобные аннотации, которые позволяют быстро писать код баз данных, менее однообразный и более надежный.

MVVM — архитектурный паттерн проектирования, определяющий структуру приложения. Название происходит от термина «модель-представление-модель представления» (Model-View-ViewModel).

Приложения Room обычно используют структуру MVVM

Приложения, использующие Room, включая приложение Tasks, обычно используют структуру архитектурного паттерна проектирования, который называется ***MVVM*** (сокращение от Model-View-ViewModel). Эта структура выглядит так:

Представление

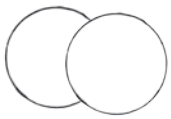
Включает весь код, отвечающий за пользовательский интерфейс: код активностей, фрагментов и макетов.

В приложении Tasks на уровне представления (View) находятся MainActivity, TasksFragment и их макеты.

Активности и фрагменты

Модель представления

Объекты моделей представлений отвечают за бизнес-логику и данные каждого представления.



Например, TasksViewModel.

Модели представлений

Модель

Данные, лежащие в основе приложения. В данном случае это база данных, но в других приложениях это может быть удаленный источник данных.

Вы больше узнаете об уровне базы данных в этой главе.

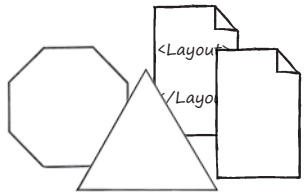
База данных

Эта структура похожа на ту, которая используется в построенном нами приложении Guessing Game, не считая того, что в структуре присутствует дополнительный уровень модели для базы данных. Это означает, что UI-код активностей и фрагментов четко отделяется от бизнес-логики, содержащейся в модели представления, а модель представления отделяется от остального кода, обеспечивающего работу базы данных.

О том, как использовать структуру MVVM, вы узнаете в процессе построения приложения Tasks.

Что мы собираемся сделать

Основные этапы построения приложения Tasks:



1. **Подготовка базового приложения.**
Мы создадим файлы приложения, обновим его файлы *build.gradle*, чтобы они использовали необходимые библиотеки, а также напишем базовый код активности, фрагмента и макета.

2. **Написание кода базы данных.**
На этом этапе мы добавим код создания базы данных с таблицей, а также методы доступа к данным, необходимые для взаимодействия с данными таблицы.

Мы определим базу данных с именем tasks_database.

tasks_database

taskId	task_name	task_done

task_table

База данных включает таблицу с именем task_table.

3. **Вставка записей задач.**
Мы создадим модель представления и обновим фрагмент приложения, чтобы приложение могло использоваться для вставки записей.

В макет будут добавлены представления, которые будут использоваться для вставки записей задач.

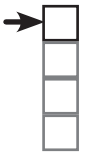
На последнем этапе мы выведем список задач, чтобы пользователь мог просмотреть введенные записи.

4. **Вывод списка записей с описаниями задач.**
Наконец, мы обновим код модели представления и фрагмента, чтобы приложение выводило список всех записей с описаниями задач, хранящимися в базе данных.

Создание проекта Tasks

Для приложения Tasks будет создан новый проект по схеме, уже знакомой вам по предыдущим главам. Выберите вариант Empty Activity, введите имя «Tasks» и имя пакета «com.hfad.tasks», подтвердите папку сохранения по умолчанию. Убедитесь в том, что выбран язык Kotlin и минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android.

Затем мы обновим файлы *build.gradle* проекта, чтобы они включали все возможности и зависимости, необходимые для приложения.



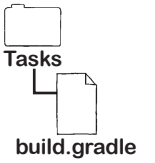
Подготовка
База данных
Вставка
Чтение записей

Добавление переменной в файл build.gradle проекта...

В этой главе будут использоваться две библиотеки Room, поэтому в файл *build.gradle* проекта будут добавлены новые переменные для используемой версии. Откройте файл *Tasks/build.gradle* и добавьте следующие строки (выделенные жирным шрифтом) в раздел `buildscript`:

```
buildscript {
    ext.lifecycle_version = "2.3.1"
    ext.room_version = "2.3.0"
    ...
}
```

Переменные определяют, какие версии библиотек жизненного цикла и Room будут использоваться. Они помогают поддерживать порядок и целостность структуры.



...и обновление файла build.gradle приложения

В файле *build.gradle* приложения необходимо включить связывание данных и добавить зависимости для модели представления, Live Data и библиотек Room.

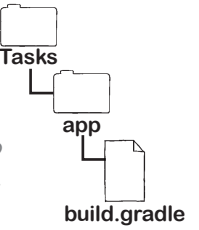
Откройте файл *Tasks/app/build.gradle* и включите следующие строки (выделенные жирным шрифтом) в соответствующие разделы:

```
plugins {
    ...
    id 'kotlin-kapt'
}
```

Средство обработки аннотаций Kotlin, необходимое для Room.

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
...
```

В приложении будет использоваться связывание данных, его необходимо включить.



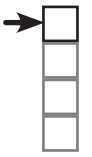

```
dependencies {
    ...
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
    ...
}
```

Модель представления и библиотеки Live Data.

Будут использоваться эти библиотеки Room.

Также необходимо использовать компилятор Room.

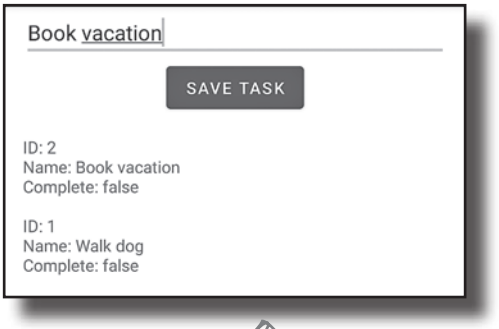
Затем щелкните на ссылке Sync Now, чтобы синхронизировать внесенные изменения с остальными частями проекта.



Создание TasksFragment

Приложение включает один фрагмент с именем `TasksFragment`, который будет использоваться для вывода списка всех задач из базы данных и вставки новых задач.

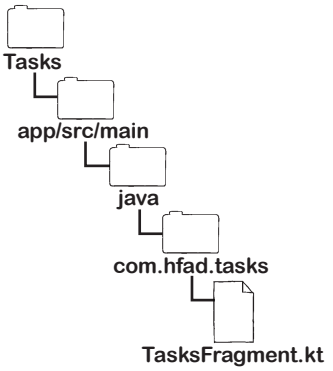
Чтобы создать `TasksFragment`, выделите файл *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «TasksFragment» и имя макета «fragment_tasks» и убедитесь в том, что выбран язык Kotlin.



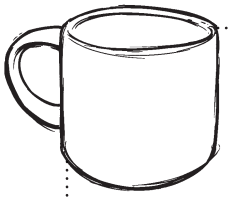
Фрагмент TasksFragment.

Обновление TasksFragment.kt

После того как фрагмент `TasksFragment` будет добавлен в проект, убедитесь в том, что содержимое *TasksFragment.kt* совпадает с приведенным ниже:



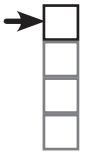

```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```



Расслабьтесь

Если компилятор не распознает класс FragmentTasksBinding, не беспокойтесь.

Дело в том, что в макет фрагмента еще не включен элемент `<layout>`*, необходимый для генерирования класса связывания.*



Подготовка
База данных
Вставка
Чтение записей

Обновление fragment_tasks.xml

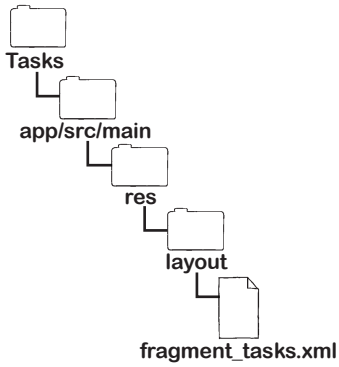
Также необходимо обновить макет `TasksFragment`, чтобы он использовал связывание данных, и включить представления для ввода новых и вывода списка существующих задач.

Откройте файл *fragment_tasks.xml* и включите в него следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task" />

        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</layout>
```

Чтобы макет использовал связывание данных, он должен содержать корневой элемент <layout>.

Позднее мы включим переменную связывания данных.



Текстовое поле будет использоваться для ввода имени задачи.

Кнопка сохраняет задачу.

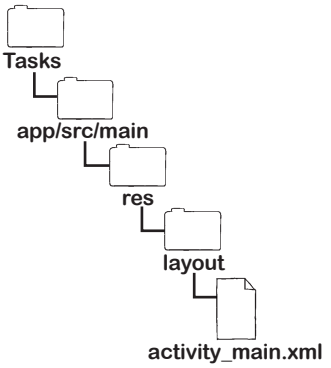
Текстовое представление используется для вывода списка записей с описаниями задач, хранящихся в базе данных.

Подготовка
База данных
Вставка
Чтение записей

Отображение TasksFragment в макете MainActivity's...

Чтобы использовать фрагмент `TasksFragment`, необходимо включить его в макет `MainActivity` в `FragmentContainerView`.

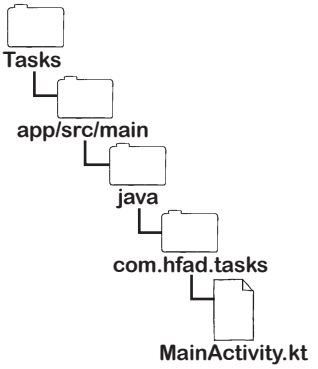
Обновите файл *activity_main.xml*, чтобы он выглядел так:




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:name="com.hfad.tasks.TasksFragment"
    tools:context=".MainActivity" />
```

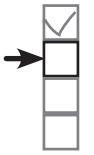
...и проверка кода MainActivity.kt

После обновления макета откройте *MainActivity.kt* и убедитесь в том, что он содержит следующий код:




```
package com.hfad.tasks

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Код фрагмента и активности обновлен, можно переходить к работе с базой данных Room.



Подготовка
База данных
Вставка
Чтение записей

Как создаются базы данных Room

Room использует набор интерфейсов и классов с аннотациями для создания баз данных SQLite для вашего приложения. Для этого необходимы три условия:

1. Класс базы данных.

Класс определяет базу данных, включая ее имя и номер версии. Он используется для получения экземпляра базы данных.

База данных

2. Классы данных таблиц.

Вся информация в базе данных хранится в таблицах. Каждая таблица определяется при помощи класса данных, который включает аннотации для имени таблицы и ее столбцов.

Класс данных Kotlin позволяет создавать объекты, основным предназначением которых является хранение данных.

taskId	task_name	task_done
1	Walk dog	false
2	Book vacation	false
3	Arrange picnic	false

Table

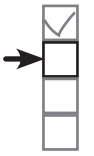
3. Интерфейсы доступа к данным.

Для взаимодействия с таблицей используется интерфейс, который определяет методы доступа к данным, необходимые для вашего приложения. Например, если вам потребуется выполнять вставку записей, в интерфейс включен метод `insert()`, а для получения всех записей добавляется метод `getAll()`.

Интерфейс доступа к данным

Room использует эти три составляющие для генерирования всего кода, необходимого приложению для создания базы данных SQLite, ее таблиц и методов доступа к данным.

На нескольких ближайших страницах мы покажем, как написать код этих трех компонентов, на примере определения базы данных для приложения Tasks. Начнем с определения таблиц.



Данные задач будут храниться в таблице

Как говорилось ранее, вся информация базы данных хранится в одной или нескольких таблицах. Каждая таблица состоит из строк и столбцов; каждая строка соответствует записи, а каждый столбец содержит один блок данных (например, число или текст).

Для каждого типа данных, который должен храниться в базе данных, создается отдельная таблица. Например, в календарном приложении может присутствовать таблица для хранения событий, а в приложении для прогноза погоды — таблица для хранения географических областей.

В приложении Tasks должны храниться записи с описаниями задач; мы создадим таблицу с именем «task_table». Таблица будет выглядеть примерно так:

Записи, которые вводятся пользователем в приложении.

taskId	task_name	task_done
1	Walk dog	false
2	Book vacation	false
3	Arrange picnic	false

Таблица состоит из трех столбцов: taskId, task_name и task_done.

Таблицы определяются классом данных с аннотациями

Для каждой таблицы, которая должна быть включена в базу данных, определяется класс данных. Этот класс данных должен включать свойство для каждого столбца таблицы, а аннотации сообщают Room, как должна быть настроена конфигурация таблицы.

Чтобы вы поняли, как это делается, мы определим класс данных с именем `Task`, который будет использоваться для создания таблицы в базе данных приложения Tasks.

(Data) Task
taskId: Long
taskName: String
taskDone: Boolean

Класс данных Task включает свойство для кажого столбца таблицы.

Создание класса данных Task

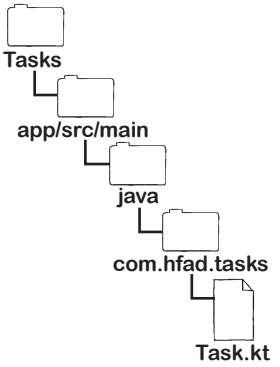
Начнем с создания класса данных. Выделите пакет *com.hfad.tasks* в папке *app/src/main/java*, после чего выберите команду File→New→Kotlin Class/File. Введите имя файла «Task» и выберите вариант Class.

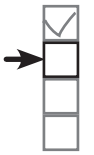
После того как файл будет создан, включите в него следующий код:

```
package com.hfad.tasks

data class Task(
    var taskId: Long = 0L,
    var taskName: String = "",
    var taskDone: Boolean = false
)
```

Класс Task должен быть классом данных.

Добавляем эти три свойства.





Подготовка
База данных
Вставка
Чтение записей

Определение имени таблицы аннотацией @Entity

После создания класса данных `Task` необходимо добавить аннотации, которые сообщают Room, как должна определяться конфигурация таблицы. Начнем с имени таблицы.

Чтобы задать имя таблицы, добавьте в класс данных аннотацию **`@Entity`** с именем таблицы. Таблице из приложения Tasks будет присвоено имя «task_table», а соответствующий код выглядит так:

```
...
@Entity(tableName = "task_table")
data class Task(
...
```

Сообщает Room, что класс описывает сущность task_table.

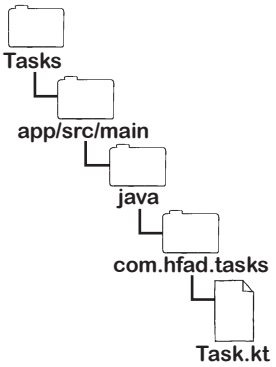
Первичный ключ задается аннотацией @PrimaryKey

Затем необходимо задать первичный ключ таблицы. Он используется для однозначной идентификации отдельной записи и не может принимать повторяющиеся значения.

В приложении Tasks для первичного ключа таблицы `task_table` будет использоваться свойство `taskId`, а таблица будет автоматически генерировать его значения, чтобы они были уникальными. Для этого используется аннотация **`@PrimaryKey`**:

```
...
    @PrimaryKey(autoGenerate = true)
    var taskId: Long = 0L,
...
```

taskId — первичный ключ, значения которого автоматически генерируются Room.



Определение имен столбцов аннотацией @ColumnInfo

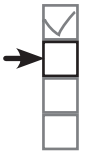
Остается задать имена столбцов для свойств `taskName` и `taskDone`. Для этого используется аннотация **`@ColumnInfo`**:

```
...
    @ColumnInfo(name = "task_name")
    var taskName: String = "",

    @ColumnInfo(name = "task_done")
    var taskDone: Boolean = false
}
```

Эти аннотации переопределяют имена столбцов, которые будут использоваться для этих двух свойств.

Учтите, что аннотация `@ColumnInfo` *необходима только в том случае, если имя столбца должно отличаться от имени свойства.* Если опустить эту аннотацию, Room присвоит столбцу имя, совпадающее с именем столбца.

И это все, что необходимо знать для создания класса данных `Task`. Полный код приводится на следующей странице.



Полный код Task.kt

Ниже приведен полный код класса данных `Task`; обновите файл *Task.kt* (изменения выделены жирным шрифтом):

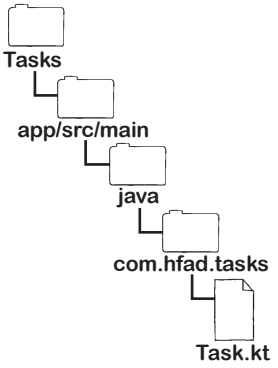
```
package com.hfad.tasks

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "task_table")
data class Task(

    @PrimaryKey(autoGenerate = true)
    var taskId: Long = 0L,

    @ColumnInfo(name = "task_name")
    var taskName: String = "",

    @ColumnInfo(name = "task_done")
    var taskDone: Boolean = false
)
```

Импортируемые классы.

Имя таблицы.

Определяет первичный ключ.

Имена двух столбцов.



Room использует этот файл для создания таблицы с именем `task_table` с автоматически генерируемым первичным ключом `taskId` и двумя дополнительными столбцами с именами `task_name` и `task_done`. Таблица выглядит примерно так:

taskId	task_name	task_done

task_table

Вы увидите, как Room добавляет эту таблицу в базу данных, через несколько страниц, когда мы напишем класс базы данных. Но сначала мы определим методы доступа к базе данных, чтобы приложение могло вставлять, читать, обновлять и удалять данные из таблицы.

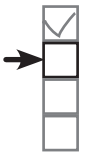
Часто Задаваемые Вопросы

В: Если опустить аннотацию `@ColumnInfo`, создаст ли Room столбец для этого свойства класса данных?

О: Да. Будет создан столбец, имя которого совпадает с именем свойства. Аннотация `@ColumnInfo` просто позволяет переименовать столбец.

В: Можно ли запретить Room создавать столбец для одного из свойств класса данных?

О: Да. Если вы не хотите, чтобы информация одного из свойств класса данных сохранялась в базе данных, его можно пометить аннотацией `@Ignore`. Для такого свойства Room не будет создавать столбец в таблице.

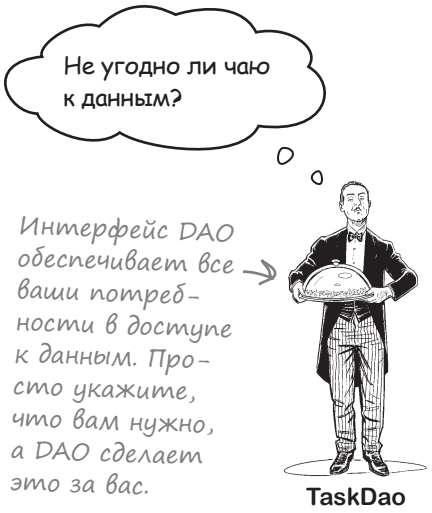


Подготовка
База данных
Вставка
Чтение записей

Использование интерфейса для определения операций с данными

Чтобы определить, как приложение должно обращаться к данным таблицы, следует создать интерфейс с аннотациями. Интерфейс определяет объект DAO (*Data Access Object*, то есть «объект доступа к данным») со всеми методами, необходимыми приложению для вставки, чтения, обновления и удаления данных.

Чтобы вы лучше поняли, как это делается, мы создадим новый интерфейс с именем `TaskDao`, который используется приложением Tasks для взаимодействия с данными `task_table`.



Создание интерфейса TaskDao

Начнем с создания интерфейса. Выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя файла «TaskDao» и выберите вариант Interface.

После того как файл будет создан, убедитесь в том, что код выглядит так:

```
package com.hfad.tasks

interface TaskDao {

}
```

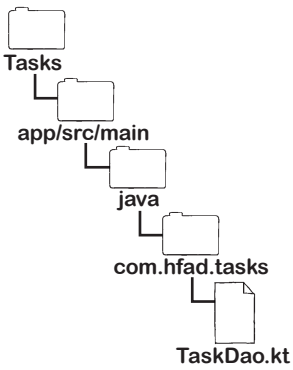
TaskDao определяется как интерфейс.

Использование @Dao для пометки интерфейса для доступа к данным

На следующем шаге необходимо сообщить Room, что интерфейс `TaskDao` определяет методы доступа к данным Для этого интерфейс помечается аннотацией **`@Dao`**:

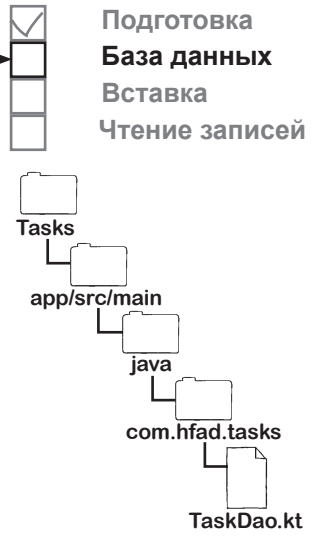
```
...
@Dao
interface TaskDao {
}
```

Сообщает Room, что интерфейс используется для доступа к данным.



После того как интерфейс будет помечен аннотацией `@Dao`, следует добавить методы с аннотациями, которые будут использоваться приложением для взаимодействия с данными. Например, если вы хотите, чтобы в приложении выполнялась вставка записей, необходимо добавить соответствующий метод в интерфейс; чтобы выполнять чтение, вы добавляете другой метод, и т. д.

К счастью, Room предоставляет четыре аннотации `@Insert`, `@Update`, `@Delete` и `@Query`, с которыми эти методы добавляются проще простого. Давайте добавим несколько методов доступа к данным в `TaskDao`.



Использование аннотации @Insert для вставки записи

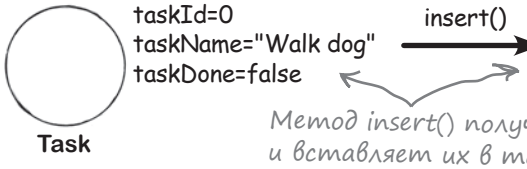
Начнем с определения метода доступа к данным `insert()`, который будет использоваться приложением для вставки задачи в `task_table`. Метод имеет один параметр `Task` с данными вставляемой задачи. Метод помечается аннотацией **`@Insert`**, которая сообщает Room, что метод используется для вставки записей.

Полный код метода `insert()`:

```
@Insert
fun insert(task: Task)
```

Да! И это весь код, который необходимо включить в TaskDao для этого метода. Все остальное Room сделает за вас.

Встречая аннотацию `@Insert`, Room автоматически генерирует весь код, необходимый приложению для вставки записи в таблицу, чтобы вам не пришлось писать его самостоятельно. Например, для приведенного выше метода `insert()` будет сгенерирован весь код, необходимый для вставки данных объекта `Task` в таблицу `task_table`:



taskId	task_name	task_done
1	Walk dog	false

Метод insert() получает значения свойств Task и вставляет их в таблицу. Таблица заменяет taskId автоматически сгенерированным значением.

Любые методы, помеченные аннотацией `@Insert`, могут получать в аргументах один или несколько объектов сущностей, то есть объектов, тип которых помечен аннотацией `@Entity`. Методы `@Insert` также могут получать коллекции объектов сущностей. Например, следующий метод вставляет все записи, включенные в аргумент `List<Task>`:

```
@Insert
fun insertAll(tasks: List<Task>)
```

Используйте этот метод, если потребуется добавить в базу данных несколько записей задач. Он вставляет все объекты Task из списка List<Task>.

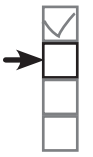
Использование @Update для обновления записей

Room также может генерировать весь код, необходимый для обновления одной или нескольких существующих записей в таблице. Для этого в интерфейс DAO добавляется метод, помеченный аннотацией **`@Update`**. Например, следующий метод `update()` генерирует весь код, необходимый для обновления существующей записи с описанием задачи:

```
@Update
fun update(task: Task)
```

Обновляет запись задачи с соответствующим значением taskId.

При вызове этот метод обновляет запись с соответствующим значением `taskId`, чтобы ее данные совпадали со значениями свойств объекта `Task`.



Подготовка
База данных
Вставка
Чтение записей

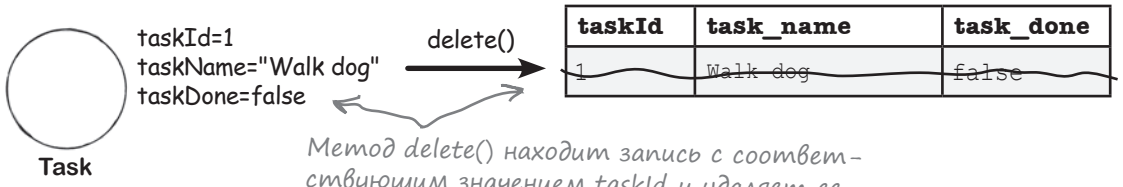
Использование @Delete для удаления записей

Также существует аннотация **@Delete**, предназначенная для пометки любых методов, которые должны удалять из таблицы конкретные записи. Например, для удаления одной записи можно воспользоваться следующим методом:

```
@Delete
fun delete(task: Task)
```

Удаляет запись задачи с соответствующим значением taskId.

Код, генерируемый Room для этого метода, удаляет запись с соответствующим значением `taskId`:



Метод delete() находит запись с соответствующим значением taskId и удаляет ее.

Использование @Query для всех остальных операций

Все остальные методы доступа к данным помечаются аннотацией `@Query`. Эта аннотация позволяет определить команду SQL (с командами `SELECT`, `INSERT`, `UPDATE` или `DELETE`), которая будет выполняться при вызове метода.

В этой книге мы не будем учить вас пользоваться языком SQL, но если вы захотите узнать больше, можем порекомендовать книгу «Head First SQL» Линн Бейли (Lynn Beighley).

Например, в приложении Tasks следующий код может использоваться для определения метода с именем `get()`, возвращающего объект Live Data `Task` для записи с соответствующим значением `taskId`:

```
@Query("SELECT * FROM task_table WHERE taskId = :taskId")
fun get(taskId: Long): LiveData<Task>
```

Возвращает запись задачи с соответствующим значением taskId.

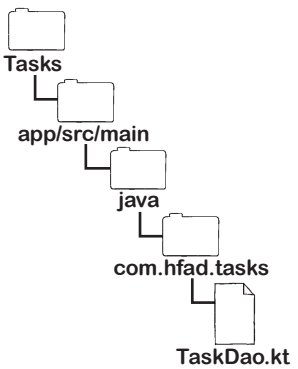
Также определим метод `getAll()`, который возвращает объект Live Data `List` со всеми записями, хранящимися в таблице:

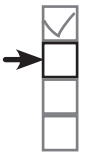
```
@Query("SELECT * FROM task_table ORDER BY taskId DESC")
fun getAll(): LiveData<List<Task>>
```

Возвращает все записи задач, упорядоченные по убыванию значений taskId.

Все эти методы возвращают объекты Live Data, и приложение может использовать их для оповещения об изменениях данных. Мы воспользуемся этой возможностью позднее в этой главе, чтобы поддерживать актуальность списка записей, отображаемого в `TasksFragment`.

Теперь вы знаете все, что необходимо знать для завершения кода `TaskDao`. Полный код приводится на следующей странице.





Полный код TaskDao.kt

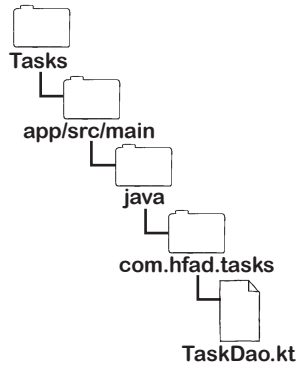
Ниже приведен полный код интерфейса `TaskDao`; обновите код *TaskDao.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update

@Dao
interface TaskDao {
    @Insert
    fun insert(task: Task)

    @Update
    fun update(task: Task)

    @Delete
    fun delete(task: Task)

    @Query("SELECT * FROM task_table WHERE taskId = :taskId")
    fun get(taskId: Long): LiveData<Task>

    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

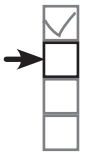
Импортируемые классы.

Интерфейс помечается аннотацией @Dao.

Включаемые методы доступа к данным.



Мы написали полный код класса данных `Task` (который определяет таблицу) и интерфейса `TaskDao` (который определяет методы доступа к данным). А сейчас вы узнаете, как определяются базы данных.



Создание абстрактного класса TaskDatabase

Чтобы определить базу данных приложения, следует создать абстрактный класс. Абстрактный класс определяет имя и номер версии базы данных, а также любые классы или интерфейсы, определяющие таблицы и методы доступа к базе данных.

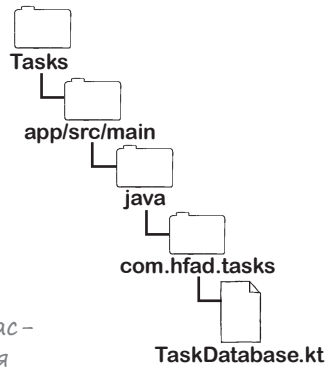
В приложении Tasks для определения базы данных будет использован абстрактный класс с именем `TaskDatabase`. Выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя файла «TaskDatabase» и выберите вариант Class.

Класс `TaskDatabase` должен расширять `RoomDatabase`, поэтому обновите код *TaskDatabase.kt*:

```
package com.hfad.tasks

import androidx.room.RoomDatabase

abstract class TaskDatabase : RoomDatabase() {
}
```

Импортируемый класс.

Класс должен расширять RoomDatabase.



Пометка класса аннотацией @Database

Затем класс необходимо пометить аннотацией **`@Database`**, которая сообщает Room, что он определяет базу данных. Эта задача решается следующим кодом:

```
package com.hfad.tasks

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
}
```

Импортируем класс Database.

Таблица, определенная в классе данных Task, добавляется в базу данных.

Как видно из этого примера, аннотация **`@Database`** имеет три атрибута: **`entities`**, **`version`** и **`exportSchema`**.

`entities` задает любые классы (помеченные атрибутом `@Entity`), определяющие таблицы, которые библиотека Room должна добавить в базу данных. В приложении Tasks это класс данных `Task`.

`version` — целое число, определяющее номер версии базы данных. В данном примере он равен 1, так как создается первая версия базы данных.

Наконец, атрибут `exportSchema` сообщает Room, нужно ли экспортировать схему базы данных в папку, чтобы сохранить историю ее версий. В данном примере этому атрибуту присваивается *false*.

Если вы изменяете файлы сущностей базы данных, необходимо обновить номер версии.

Если изменить схему без обновления номера версии, при попытке запустить его произойдет ошибка.

- Подготовка
- **База данных**
- Вставка
- Чтение записей

Добавление свойств для интерфейсов DAO

Затем необходимо определить интерфейсы (помеченные аннотацией `@Dao`), которые будут использоваться для доступа к данным. Для этого следует добавить свойство для каждого интерфейса.

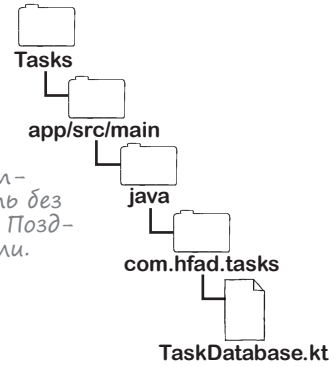
В приложении Tasks определяется один интерфейс DAO с именем `TaskDao`, поэтому в код `TaskDatabase` добавляется новое свойство `taskDao`:

```
...
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
    abstract val taskDao: TaskDao
}
```

Сообщает Room, что вы хотите использовать методы доступа к данным, определенные в TaskDao.

Создание и возвращение экземпляра базы данных

Наконец, нам понадобится метод `getInstance()`, который создает базу данных и возвращает ее экземпляр. Код выглядит так:




```
...
abstract class TaskDatabase : RoomDatabase() {
    ...
    companion object {
        @Volatile
        private var INSTANCE: TaskDatabase? = null

        fun getInstance(context: Context): TaskDatabase {
            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        TaskDatabase::class.java,
                        "tasks_database"
                    ).build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

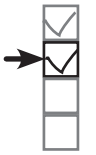
Метод getInstance() помещается в объект-компаньон, чтобы его можно было вызвать без предварительного создания TaskDatabase. Позднее вы увидите, почему мы так поступили.

Этот метод используется для получения экземпляра базы данных. Во всех базах данных, которые вы будете создавать, он будет практически одинаковым.

Если база данных не существует, метод getInstance() создает ее.

Возвращает экземпляр TaskDatabase.

И это все, что нам понадобится для класса `TaskDatabase`.
Полный код будет приведен на следующей странице.



Полный код TaskDatabase.kt

Ниже приведен полный код абстрактного класса `TaskDatabase`; обновите код *TaskDatabase.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.tasks

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {
    abstract val taskDao: TaskDao

    companion object {
        @Volatile
        private var INSTANCE: TaskDatabase? = null

        fun getInstance(context: Context): TaskDatabase {
            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        TaskDatabase::class.java,
                        "tasks_database"
                    ).build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

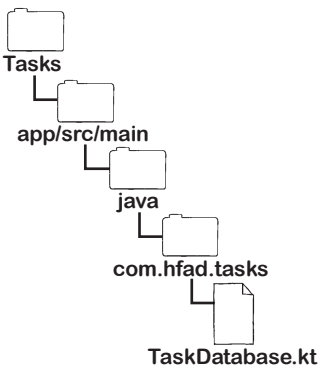
Импортируем эти классы.

Добавляем эту аннотацию.

Определяем свойство для TaskDao.

Определяется объект-компаньон, чтобы вызвать метод getInstance() в виде TaskDatabase.getInstance().

Возвращает экземпляр TaskDatabase. Строит базу данных, если она еще не существует.



Мы написали весь код базы данных, необходимый для приложения Tasks. Но прежде чем создавать оставшийся код приложения, проверьте свои силы на следующем упражнении.

Возьмите в руку карандаш

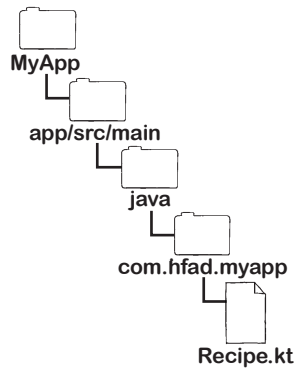
Таблица с именем `recipe_table` должна содержать следующие столбцы:

recipeId	name	ingredients	method

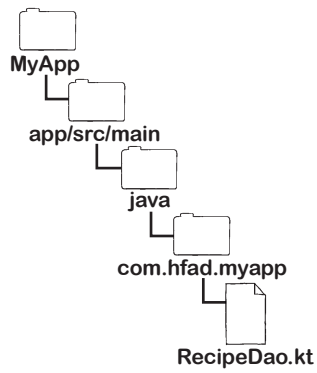
recipeId — автоматически генерируемый первичный ключ.

Попробуйте расставить аннотации в классе данных `Recipe` и интерфейсе `RecipeDao`, чтобы библиотека Room создавала заданную таблицу вместе с методами вставки, обновления и удаления ее данных.

```
data class Recipe(
    var recipeId: Long = 0L,
    var recipeName: String = "",
    var recipeIngredients: String = "",
    var recipeMethod: String = ""
)
```




```
interface RecipeDao {
    fun insert(recipe: Recipe)
    fun update(recipe: Recipe)
    fun delete(recipe: Recipe)
}
```



Возьмите в руку карандаш
Решение

Таблица с именем `recipe_table` должна содержать следующие столбцы:

recipeId	name	ingredients	method

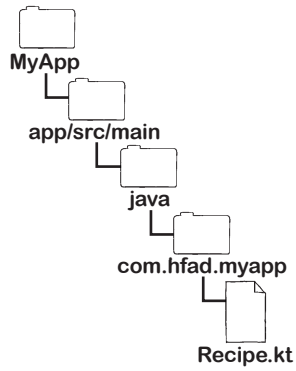
Попробуйте расставить аннотации в классе данных `Recipe` и интерфейсе `RecipeDao`, чтобы библиотека Room создавала заданную таблицу вместе с методами вставки, обновления и удаления ее данных.

Имя таблицы.

```
@Entity(tableName = "recipe_table")
data class Recipe(
    @PrimaryKey(autoGenerate = true)
    var recipeId: Long = 0L,

    @ColumnInfo(name = "name")
    var recipeName: String = "",

    @ColumnInfo(name = "ingredients")
    var recipeIngredients: String = "",

    @ColumnInfo(name = "method")
    var recipeMethod: String = ""
)
```

recipeId назначается первичным ключом, а его значения генерируются автоматически.

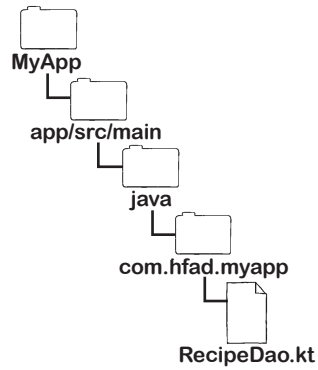
Имена столбцов для этих свойств.

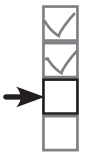



```
@Dao
interface RecipeDao {
    @Insert
    fun insert(recipe: Recipe)

    @Update
    fun update(recipe: Recipe)

    @Delete
    fun delete(recipe: Recipe)
}
```

RecipeDao помечается как DAO.

Методы снабжаются аннотациями для вставки, обновления и удаления записей.





Снова о MVVM

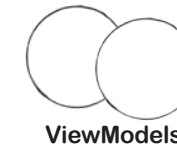
Ранее в этой главе мы упоминали о том, что структура приложения Tasks будет основана на архитектурном паттерне проектирования MVVM («модель–представление–модель представления»). Кратко напомним, как выглядят такие структуры:

Представление

Весь код, отвечающий за пользовательский интерфейс: код активностей, фрагментов и макетов.

Модель представления

Объекты моделей представлений отвечают за бизнес-логику и данные каждого представления.



Модель

Код базы данных приложения, включая ее сущности и объекты DAO.

Весь код модели готов...

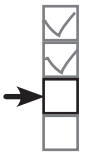
К настоящему моменту мы написали весь код базы данных, необходимый приложению; для этого были созданы файлы сущностей, DAO и файлы определения базы данных (`Task`, `TaskDao` и `TaskDatabase`). Написание всего кода базы данных означает, что мы завершили часть архитектуры приложения, относящуюся к модели.

...и мы переходим к модели представления

На следующем этапе мы будем работать над моделью представления. Для этого будет создана модель представления (`TasksViewModel`), в которой будет храниться бизнес-логика `TasksFragment`. Модель приложения будет включать методы, использующие `TaskDao` для вставки записей в базу данных.

Итак, займемся созданием `TasksViewModel`.

Также в приложении должен выводиться список задач, хранящихся в базе данных, но пока сосредоточимся на вставке записей.



Подготовка
База данных
Вставка
Чтение записей

Создание TasksViewModel

Чтобы создать класс `TasksViewModel`, выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя «TasksViewModel» и выберите вариант Class.

Мы обновим код модели представления, чтобы `TasksFragment` мог использовать его для вставки новых записей с описаниями задач. Для этого коду потребуются три составляющие:

1. **Ссылка на объект TaskDao.**
   `TasksViewModel` будет использовать этот объект для взаимодействия с базой данных, поэтому он будет передаваться модели представления в ее конструкторе.

2. **Строковое свойство для хранения имени новой задачи.**
   Когда пользователь вводит новое имя задачи, `TasksFragment` обновляет это свойство его значением.

3. **Метод addTask(), который будет вызываться из TasksFragment.**
   Этот метод будет создавать новый объект `Task`, присваивать ему имя и вставлять его в базу данных вызовом метода `insert()` объекта `TaskDao`.

Ниже приведен базовый код модели представления для этих трех составляющих; обновите код *TasksViewModel.kt*, чтобы он выглядел так:

```
package com.hfad.tasks

import androidx.lifecycle.ViewModel

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""

    fun addTask() {
        val task = Task()
        task.taskName = newTaskName
        dao.insert(task)
    }
}
```

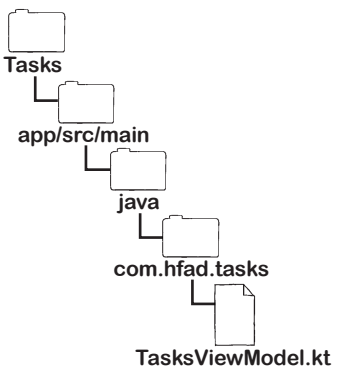
Импортируем этот класс.

TasksViewModel расширяет ViewModel.

Объект TaskDao передается TasksViewModel.

Для имени задачи.

Этот метод создает объект Task и использует метод insert() объекта TaskDao для добавления его в базу данных.



Но прежде чем `TasksFragment` сможет вызывать метод `addTask()`, необходимо внести еще одно изменение.

Операции баз данных могут работать меедлееенноооо...

Некоторые задачи в мире Android, такие как вставка записей в базу данных, потенциально могут занимать много времени. По этой причине библиотека хранения данных Room настаивает на том, что ***все операции доступа к данным должны выполняться в фоновом потоке,*** чтобы они не блокировали главный поток Android и не тормозили пользовательский интерфейс.

Формально существует настройка, которая позволяет переопределить этот режим, но лучше выполнять такие задачи в фоновом потоке, чтобы они не мешали остальным частям приложения.

Так как мы разрабатываем приложения Android на языке Kotlin, для того чтобы все методы доступа к данным выполнялись в фоновом режиме, в нашем приложении будут применяться ***сопрограммы.***

Для фонового выполнения кода доступа к данным будут использоваться сопрограммы

Сопрограмма представляет собой упрощенный программный поток, который позволяет выполнять блоки кода в асинхронном режиме. Использование сопрограмм означает, что фоновое задание (например, вставка записей в базу данных) запускается так, что остальным частям кода не приходится дожидаться его завершения. Взаимодействие пользователя с приложением становится более плавным.

Сопрограмма — блок кода, выполнение которого может приостанавливаться; такой код обычно выполняется в фоновом режиме.

Перевод кода доступа к данным на использование сопрограмм происходит относительно прямолинейно. Вам придется внести всего два изменения:

1. **Все методы доступа к данным в DAO помечаются ключевым словом suspend.** Тем самым каждый метод преобразуется в сопрограму, которая выполняется в фоновом режиме и может приостанавливаться, например:

Преобразует метод insert() в сопрограмму.

```
@Insert
suspend fun insert(task: Task)
```

2. **Сопрограммы DAO запускаются в фоновом режиме.** Например, для вызова метода `insert()` объекта `TaskDao` из `TasksViewModel` используется следующий код:

Сопрограмма запускается в одной области видимости с моделью представления.

```
viewModelScope.launch {
    ...
    dao.insert(task)
}
```

Эти изменения необходимы для всех методов доступа к данным, кроме тех, которые возвращают данные Live Data. Room уже использует фоновый поток для методов, возвращающих объекты Live Data, а это означает, что вам не придется вносить дополнительные изменения в ваш код.

Обновим код `TaskDao` и `TasksViewModel`, чтобы в нем использовались сопрограммы.

Расслабьтесь

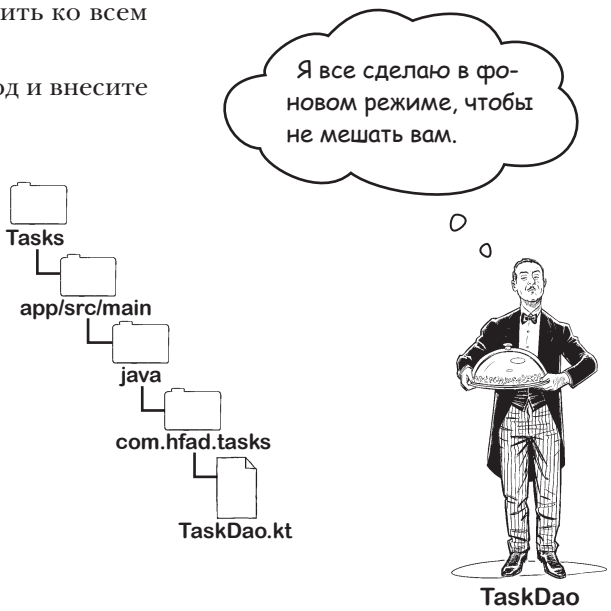
Не беспокойтесь, если вы еще неуверенно чувствуете себя с сопрограммами.

В тексте приведена вся информация, необходимая для выполнения методов доступа к данным в фоновом режиме.

1. Пометка методов TaskDao ключевым словом suspend

Прежде всего необходимо пометить все методы доступа к данным `TaskDao`, не использующие Live Data, ключевым словом `suspend`. Это означает, что изменение необходимо применить ко всем методам, кроме `get()` и `getAll()`.

Ниже приведен полный код *TaskDao.kt*; обновите код и внесите изменения, выделенные жирным шрифтом:



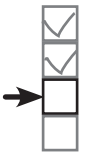

```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update

@Dao
interface TaskDao {
    @Insert
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    @Delete
    suspend fun delete(task: Task)

    @Query("SELECT * FROM task_table WHERE taskId = :key")
    fun get(key: Long): LiveData<Task>

    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

Методы insert(), update() и delete() помечаются ключевым словом suspend.

Методы get() и getAll() используют Live Data, которые изменяются в фоновом режиме. А следовательно, эти методы можно оставить в исходном виде.

Пометка методов ключевым словом suspend превращает их в сопрограммы.

И это весь код, необходимый для преобразования методов `TaskDao` в сопрограммы, которые могут выполняться в фоновом режиме.

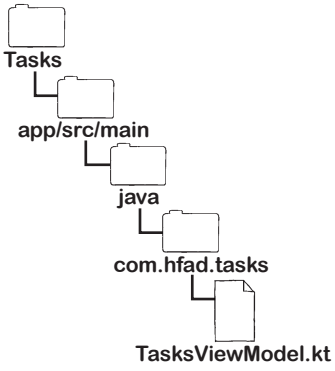


2. Метод insert() запускается в фоновом режиме

Далее необходимо обновить метод `addTask()` объекта `TasksViewModel`, чтобы он запускал метод `insert()` объекта `TaskDao` как сопрограмму. Этот код приведен ниже; обновите файл *TasksViewModel.kt* (изменения выделены жирным шрифтом):

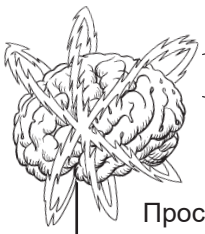
```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }
}
```

Сопрограмма запускается в одной области видимости с моделью представления.



Это изменение означает, что при каждом вызове метода `addTask()` будет вызываться метод `insert()` класса `TaskDao` (сопрограмма) для вставки записей в фоновом режиме.

Мы написали весь код, необходимый для модели представления приложения Task. Затем объект `TasksViewModel` будет добавлен в `TasksFragment`, чтобы он мог обращаться к свойствам и методам модели представления, а пользователь получил возможность вставлять записи с описаниями задач.



Мозговой штурм

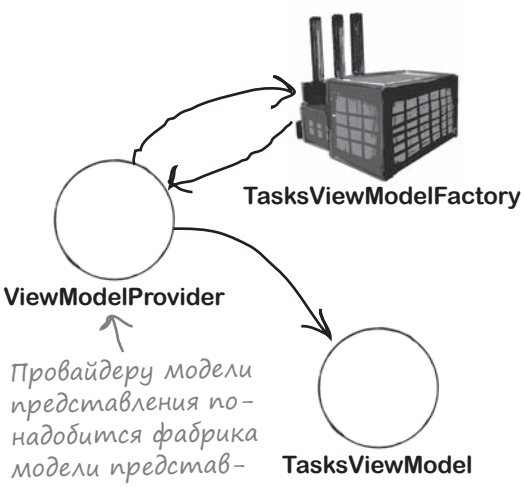
Просмотрите код `TasksViewModel`. Как вы думаете, что нужно сделать далее для включения экземпляра этого класса в `TasksFragment`?

TasksViewModel необходима фабрика модели представления

Как вы узнали в главе 11, чтобы добавить модель представления в код фрагмента, следует запросить ее у провайдера модели представления. Провайдер модели представления возвращает текущий объект модели представления фрагмента, если он существует, или создает его в противном случае.

Если модель представления имеет конструктор без аргументов, провайдер модели представления может создать экземпляр без дополнительной помощи. Но если у конструктора есть аргументы, ему потребуется поддержка фабрики модели представления.

В приложении Tasks для получения объекта `TasksViewModel` понадобится провайдер модели представления. Так как конструктор `TasksViewModel` получает аргумент `TaskDao`, сначала необходимо определить класс `TasksViewModelFactory`.

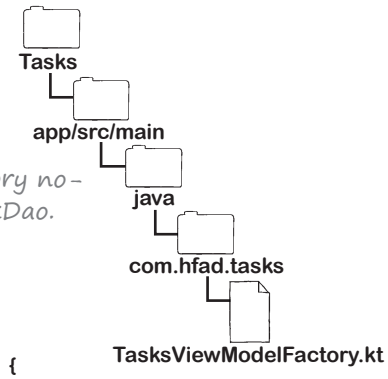


Провайдеру модели представления понадобится фабрика модели представления, которая упрощает создание моделей представлений, не имеющих конструктора без аргументов.

Создание TasksViewModelFactory

Чтобы создать фабрику, выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Kotlin Class/File. Введите имя файла «TasksViewModelFactory» и выберите вариант Class.

Код `TasksViewModelFactory` почти не отличается от кода фабрики модели представления, написанного в главе 11. Обновите файл *TasksViewModelFactory.kt*, чтобы он выглядел так:

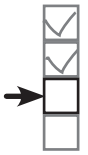



```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider

class TasksViewModelFactory(private val dao: TaskDao)
                            : ViewModelProvider.Factory {

    override fun <T: ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(TasksViewModel::class.java)) {
            return TasksViewModel(dao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

TasksViewModelFactory получает объект TaskDao.

Объект TaskDao используется для создания объекта TasksViewModel.

Фабрика модели представления готова. Давайте воспользуемся ею для включения объекта `TasksViewModel` в `TasksFragment`.



TasksViewModelFactory необходим объект TaskDao

Для включения `TasksViewModel` в `TasksFragment` необходимо создать объект `TasksViewModelFactory` и передать его провайдеру модели представления. Провайдер использует фабрику для создания модели представления.

Но тут возникает проблема: конструктору `TasksViewModelFactory` требуется аргумент `TaskDao`, а значит, такой объект необходимо получить, чтобы иметь возможность создать объект. Однако `TaskDao` — интерфейс, а не конкретный класс. Как же получить объект `TaskDao`?

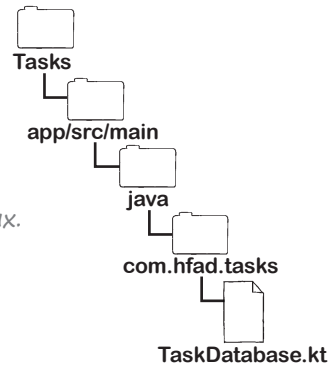
Код TaskDatabase содержит свойство TaskDao

Когда мы писали код `TaskDatabase`, в него были включены две ключевые составляющие: свойство `TaskDao` с именем `taskDao` и метод `getInstance()`, возвращающий экземпляр базы данных. Кратко напомню, как выглядит этот код:

```
abstract class TaskDatabase : RoomDatabase() {
    abstract val taskDao: TaskDao

    companion object {
        ...
        fun getInstance(context: Context): TaskDatabase {
            ...
        }
    }
}
```

Свойство taskDao.

Возвращает экземпляр базы данных.



Чтобы получить ссылку на объект `TaskDao` в коде `TasksFragment`, можно вызвать метод `getInstance()` объекта `TaskDatabase` и обратиться к его свойству `taskDao`. Соответствующий код выглядит так:

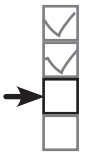
requireNotNull() — функция Kotlin, которая выдает исключение IllegalArgumentException, если ее аргумент равен null. Мы используем ее для полной надежности и абсолютной уверенности в том, что значение application отлично от null.

```
val application = requireNotNull(this.activity).application
val dao = TaskDatabase.getInstance(application).taskDao
```

Этот код будет добавлен в TasksFragment на следующей странице.

Этот код получает ссылку на текущее приложение, строит базу данных, если она еще не существует, и возвращает ее экземпляр. Затем ее объект `TaskDao` присваивается локальной переменной с именем `dao`.

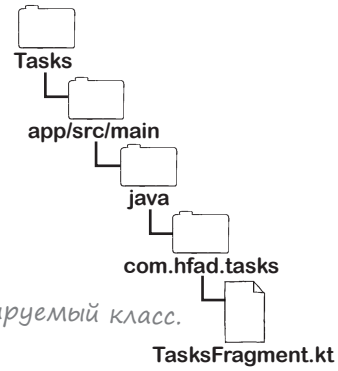
Теперь вы знаете, как получить ссылку на объект `TaskDao`, и мы можем обновить код `TasksFragment` для создания объекта `TasksViewModelFactory`, который будет использоваться для получения `TasksViewModel`. Давайте рассмотрим этот код.



Обновленный код TasksFragment.kt

Ниже приведен обновленный код `TasksViewModel`. Как видите, теперь в нем присутствует код получения объекта `TaskDao` и создания `TasksViewModelFactory`. Фабрика передается провайдеру модели представления, который использует ее для получения экземпляра `TasksViewModel`.

Обновите код *TasksFragment.kt* (изменения выделены жирным шрифтом):

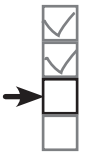



```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.tasks.databinding.FragmentTasksBinding
import androidx.lifecycle.ViewModelProvider

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Импортируемый класс.

Строит базу данных (если она еще не существует) и получает ссылку на свойство taskDao.

Получение модели представления.



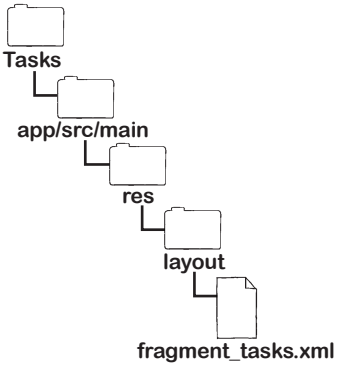
TasksFragment может использовать связывание представлений

Объект `TasksViewModel` был добавлен в `TasksFragment`, и фрагмент может использовать его свойства и методы для вставки записи в базу данных. Для этого будет использоваться связывание данных, которое предоставляет макету прямой доступ к свойствам и методам модели представления.

Чтобы включить связывание данных, необходимо сначала добавить переменную связывания данных в макет фрагмента; включите приведенный ниже код (выделенный жирным шрифтом) в раздел `<data>` файла *fragment_tasks.xml*:

```
<layout
    ...>
    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>
    ...
</layout>
```

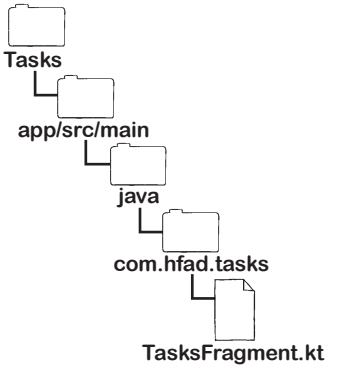
Определяет переменную связывания данных с именем viewModel и типом TasksViewModel.



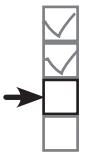
Затем свойство `viewModel` фрагмента присваивается переменной связывания данных, для чего в метод `onCreateView()` объекта `TasksFragment` включается следующая строка (выделенная жирным шрифтом):

```
...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel

        return view
    }
...
```

Включает связывание данных, чтобы макет мог использовать его для обращения к свойствам и методам модели представления.



Полный код обоих файлов будет приведен через несколько страниц. А пока обновим файл *fragment_tasks.xml*, чтобы переменная `viewModel` использовалась в нем для вставки записей в базу данных.



Подготовка
База данных
Вставка
Чтение записей

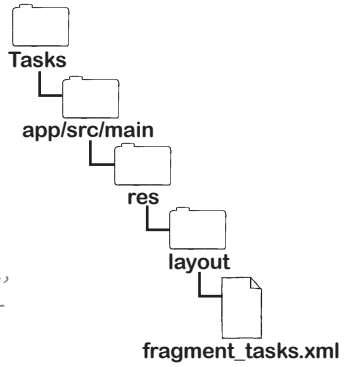
Для вставки данных будет использоваться связывание данных

Чтобы вставить в базу данных новую запись с описанием задачи, необходимо: (1) задать свойству `newTaskName` объекта `TasksViewModel` имя новой задачи и (2) вызвать его метод `addTask()`. И то и другое можно сделать в макете `TasksFragment` с использованием связывания данных.

Присваивание свойству newTaskName объекта TasksViewModel

Чтобы задать значение свойства `newTaskName` модели представления, мы свяжем его с текстовым полем `task_name` в макете фрагмента. Это делается с помощью кода:

```
<EditText
    android:id="@+id/task_name"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text"
    android:hint="Enter a task name"
    android:text="@={viewModel.newTaskName}" />
```



Задает свойству newTaskName модели представления текст, введенный пользователем.

По щелчку на этой кнопке будет вставляться новая запись с описанием задачи.

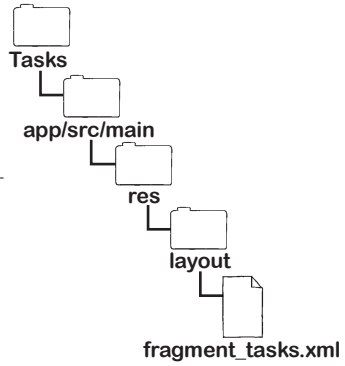
Обратите внимание: для связывания свойства с текстовым полем используется конструкция **`@=`** вместо `@`. `@=` означает, что текстовое поле может обновлять свойство, с которым оно связано, — в данном случае `newTaskName`.

Вызов метода addTask() объекта TasksViewModel

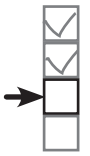
Для добавления записи с описанием задачи, мы воспользуемся связыванием данных, чтобы по щелчку на кнопке Save Task макета вызывался метод `addTask()` модели представления. Вы уже знаете, как это делается, поэтому мы просто приведем код:

```
<Button
    android:id="@+id/save_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Save Task"
    android:onClick="@{() -> viewModel.addTask()}" />
```

По щелчку на кнопке будет вызываться метод addTask() модели представления.



И это все изменения, которые необходимо внести в *fragment_tasks.xml* для вставки записи в базу данных. Давайте посмотрим, как выглядит полный код.



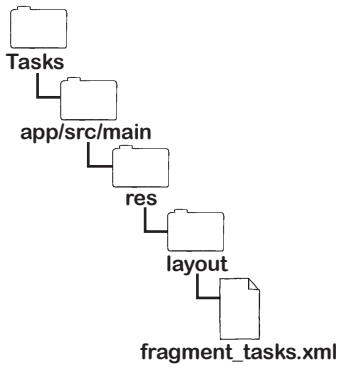
Подготовка
База данных
Вставка
Чтение записей

Полный код fragment_tasks.xml

Ниже приведен полный код макета `TasksFragment`; обновите файл *fragment_tasks.xml* (изменения выделены жирным шрифтом):

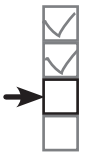
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name"
            android:text="@={viewModel.newTaskName}" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task"
            android:onClick="@{() -> viewModel.addTask()}" />

        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</layout>
```

Добавляет переменную связывания данных.



Задает свойству newTaskName модели представления имя задачи из EditText.

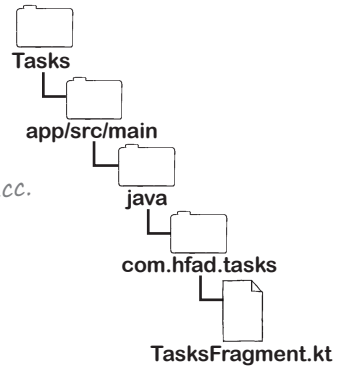
Вызывает метод addTask() модели представления, когда пользователь щелкает на кнопке для сохранения задачи.



Подготовка
База данных
Вставка
Чтение записей

Полный код TasksFragment.kt

Прежде чем запускать приложение, также необходимо убедиться в том, что `TasksFragment` включает весь код, необходимый для присваивания значения переменной связывания данных макета. Полный код приведен ниже; обновите файл *TasksFragment.kt*, если это не было сделано ранее (изменения выделены жирным шрифтом):

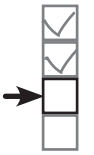



```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Импортируемый класс.

Добавьте эти строки.

Давайте разберемся, что происходит при выполнении кода.

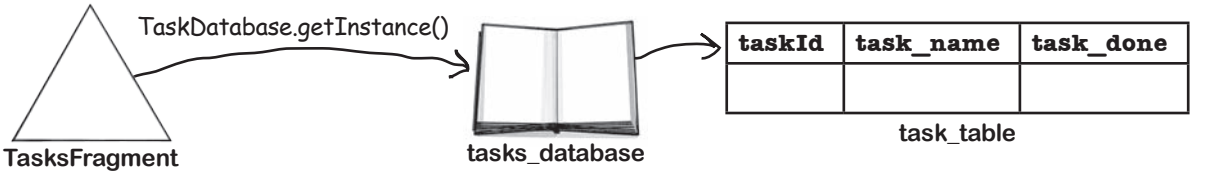


Подготовка
База данных
Вставка
Чтение записей

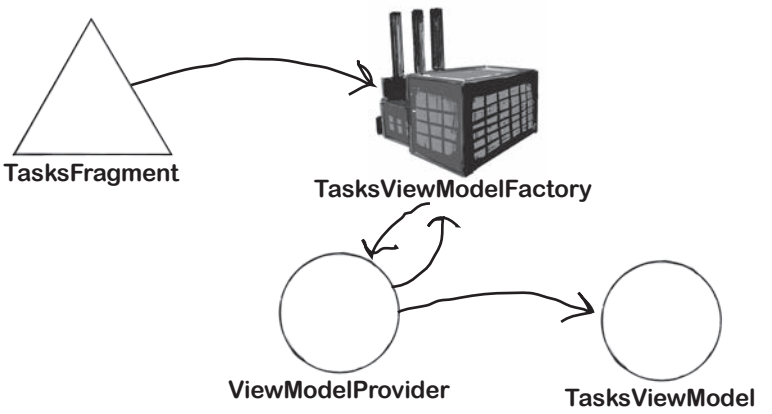
Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

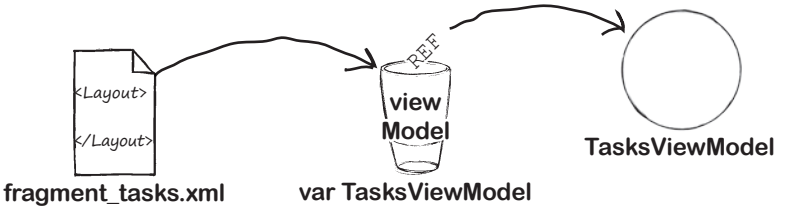
1. **TasksFragment вызывает метод TaskDatabase.getInstance(), который строит базу данных, если она еще не существует.**
При этом базе данных присваивается имя `tasks_database`, а класс данных `Task` используется для создания таблицы с именем `task_table`. После этого метод возвращает экземпляр базы данных.

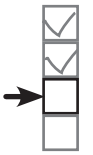


2. **TasksFragment получает объект TaskDao экземпляра базы данных и использует его для создания фабрики TasksViewModelFactory.**
Провайдер модели представления использует только что созданный объект `TasksViewModelFactory` для создания `TasksViewModel`.



3. **TasksFragment присваивает переменной связывания данных viewModel макета объект TasksViewModel.**

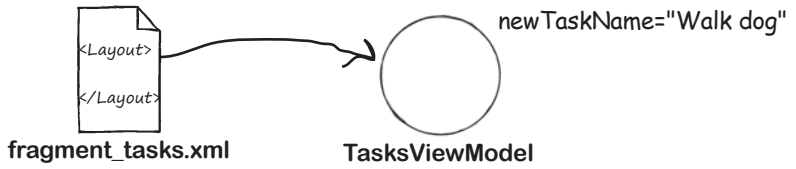




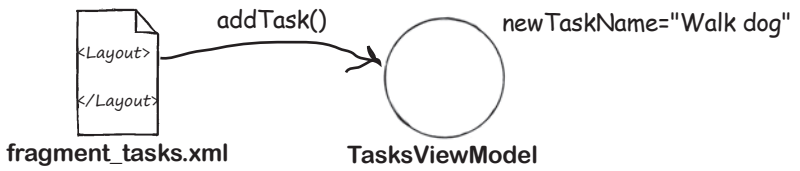
Подготовка
База данных
Вставка
Чтение записей

История продолжается

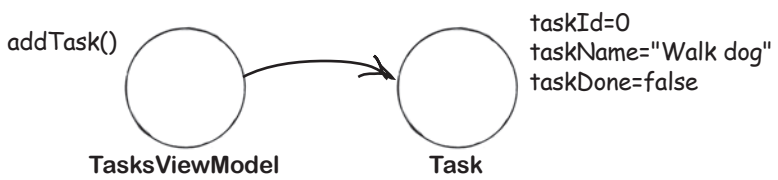
4 **Пользователь вводит имя задачи в текстовом поле.**
Макет использует связывание данных, чтобы задать свойству `newTaskName` модели представления введенное значение.



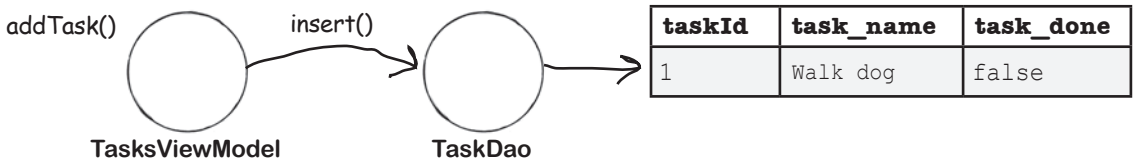
5 **Пользователь щелкает на кнопке Save Task.**
Связывание данных используется для вызова метода `addTask()` объекта `TaskViewModel`.



6 **Метод addTask() создает новый объект Task и задает его свойству taskName значение newTaskName.**



7 **Метод addTask() вызывает метод insert() объекта TaskDao.**
Это приводит к вставке записи данных объекта `Task` в таблицу базы данных. Таблица автоматически генерирует значение первичного ключа `taskId`.



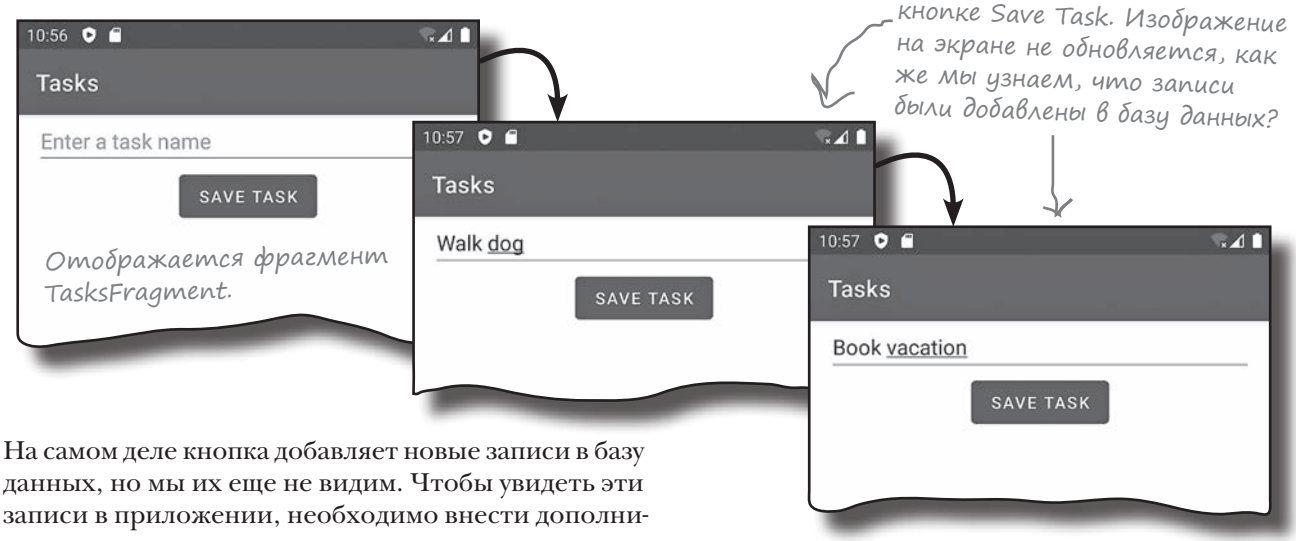
taskId	task_name	task_done
1	Walk dog	false

Проведем тест-драйв приложения.

Тест-драйв

При запуске приложения в `MainActivity` отображается фрагмент `TasksFragment`. Если ввести пару новых задач, по щелчку на кнопке Save Task вроде бы ничего не происходит.

Для каждой из этих задач мы вводим имя и щелкаем на кнопке Save Task. Изображение на экране не обновляется, как же мы узнаем, что записи были добавлены в базу данных?



На самом деле кнопка добавляет новые записи в базу данных, но мы их еще не видим. Чтобы увидеть эти записи в приложении, необходимо внести дополнительные изменения.

Часто Задаваемые Вопросы

В: Когда я попытался ввести задачу, в приложении произошла ошибка. Не знаете почему?

О: Например, это может произойти, если вы обновили схему базы данных (сущности, содержащиеся в базе данных) без обновления номера версии. Room отслеживает схемы для всех версий базы данных. При обнаружении каких-либо несоответствий в журнал приложения выводится сообщение об ошибке, а приложение аварийно завершается.

В: Понятно. И как исправить ошибку?

О: Самое простое решение — обновлять номер версии базы данных в файле, определяющем базу данных (в данном случае *TaskDatabase.kt*).

В: Как просмотреть журнал приложения?

О: Найдите соответствующую команду в Android Studio. В журнал направляются все сообщения об ошибках, поэтому поиск информации при обнаружении ошибок удобно начать с него.

В: Существуют ли какие-либо служебные программы, в которых можно просмотреть базу данных и содержащиеся в ней записи и таблицы?

О: В Android Studio для этой цели включена полезная программа Database Inspector. Дополнительную информацию можно найти по адресу *https://developer.android.com/studio/inspect/database*

В: Какие еще возможности баз данных можно использовать с Room?

О: Можно определять отношения между таблицами, создавать представления, проводить миграцию данных при обновлении базы данных и многое другое. Дополнительную информацию можно найти по адресу *https://developer.android.com/training/data-storage/room*

В: В коде макета для связывания текстового поля со свойством модели представления используется конструкция "@=" вместо "@". Почему?

О: Потому что текстовое поле должно не только читать значение, но и обновлять значение свойства. Конструкция `@=` вместо `@` дает такую возможность.

Кто я?

Компания компонентов, облаченных в маскарадные костюмы, развлекается игрой «Кто я?». Игрок дает подсказку, а остальные на основании сказанного им пытаются угадать, кого он изображает. Будем считать, что игроки всегда говорят правду о себе. Заполните пропуски справа именами одного или нескольких участников. Кроме того, для каждого участника напишите, является ли компонент частью уровня модели, представления или модели представления в архитектуре приложения.

Сегодняшние участники:

Активность Фрагмент Макет Модель представления База данных Сущность DAO

	Имя	Модель, представление или модель представления?
Я определяю внешний вид экранов приложения.		
Я сохраняю данные после завершения приложения.		
У меня есть жизненный цикл, но я не могу существовать сам по себе.		
Я используюсь для бизнес-логики.		
Room использует меня для создания таблицы.		
У меня есть жизненный цикл, и я являюсь разновидностью контекста.		
Я помогаю взаимодействовать с информацией в базе данных.		

→ Ответ на с. 657.

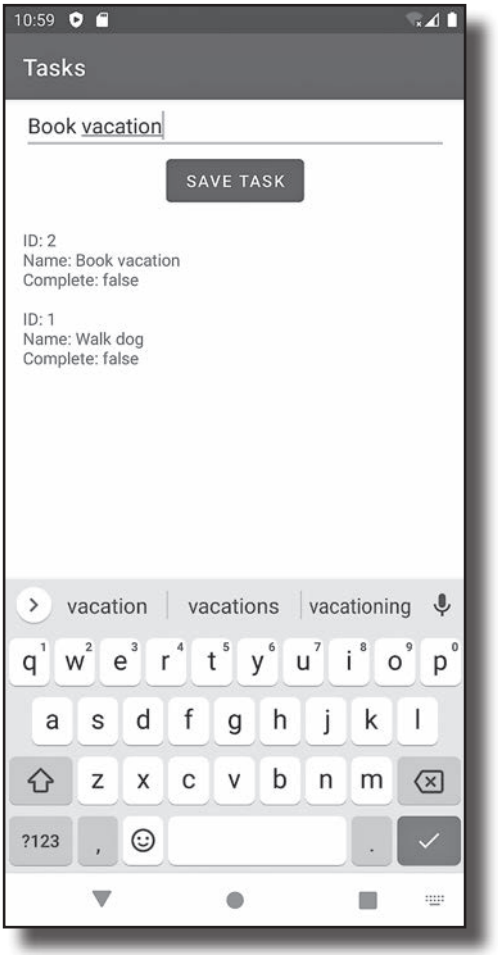
В TasksFragment должны выводиться записи

Вы узнали, как построить приложение, которое вставляет записи в базу данных Room. На следующем этапе необходимо обновить приложение так, чтобы в `TasksFragment` выводился список всех вставленных записей.

Вот как должна выглядеть новая версия `TasksFragment`:

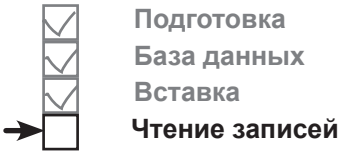
Пользователь должен иметь возможность вводить записи, как и прежде.

Все введенные записи должны отображаться в списке.



Мы воспользуемся методом `getAll()` объекта `TaskDao` для чтения всех записей из базы данных. Затем прочитанные записи отображаются в текстовом представлении `TasksFragment`, для чего они форматируются в строку.

Итак, за дело!

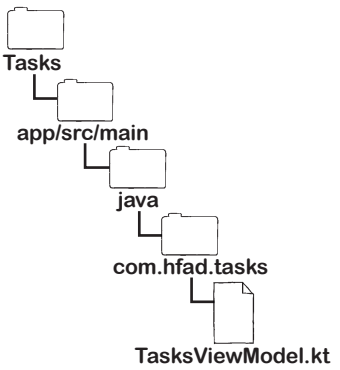


Чтение всех задач из базы данных методом getAll()

Начнем с добавления в `TasksViewModel` свойства с именем `tasks`, в котором будет храниться список всех задач в базе данных. Чтобы добавить задачи в свойство, присвойте ему результат вызова `getAll()` объекта `TaskDao`:

```
class TasksViewModel(val dao: TaskDao) : ViewModel() {
    ...
    val tasks = dao.getAll()
    ...
}
```

Вызывает метод getAll() объекта TaskDao и присваивает возвращаемое значение свойству tasks.

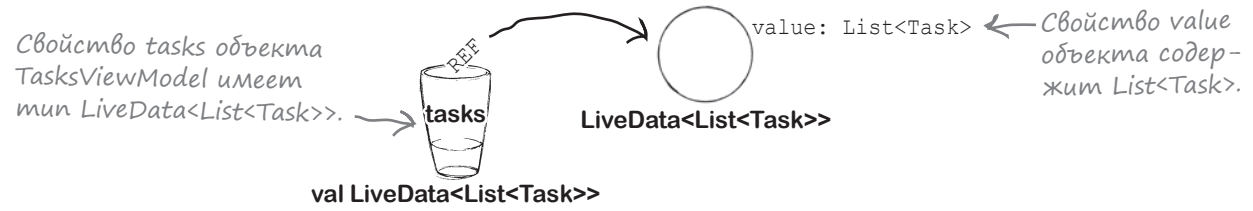


Как вы, возможно, помните, метод `getAll()` определяется в *TaskDao.kt* следующим кодом:

```
@Dao
interface TaskDao {
...
    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

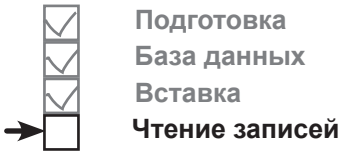
Метод getAll() получает все записи из базы данных. Он возвращает тип LiveData<List<Task>>.

Метод возвращает объект `LiveData<List<Task>>`: список объектов `Task` с данными Live Data. Это означает, что свойство `tasks` объекта `TasksViewModel` тоже относится к типу `LiveData<List<Task>>`:



Так как свойство `tasks` использует данные Live Data, оно всегда включает последние изменения, внесенные пользователем. Например, если в `task_table` добавляется новая задача, то значение свойства `tasks` автоматически обновляется и в него включается новая запись.

Такое использование Live Data весьма удобно для приложения Tasks, потому что оно может использоваться для отображения в `TasksFragment` актуального списка задач. Но прежде чем переходить к реализации этой возможности, необходимо еще кое-что узнать.



LiveData<List<Task>> — более сложный тип

При описании связывания даных мы показали, как связать представление со строкой или числом, включая значения Live Data. Это было возможно благодаря тому, что строки и числа — простые объекты, которые могут легко отображаться в текстовых представлениях.

Однако на этот раз ситуация иная. Свойство `tasks` имеет тип `LiveData<List<Task>>` — намного более сложный, чем, допустим, `LiveData<String>`. Из-за этого мы не можем просто связать текстовое представление макета со свойством напрямую, потому что текстовое представление не будет знать, как его отобразить. Что же делать?

Использование Transformations.map() для преобразования объектов Live Data

Прежде чем использовать связывание данных для отображения задач пользователя, необходимо сначала преобразовать `LiveData<List<Task>>` к более простой форме, которую умеет отображать текстовое представление. Для этого мы создадим новое свойство с именем `tasksString`, в котором будет храниться версия свойства `tasks` в формате `LiveData<String>`.

Transformations.map() наблюдает за объектом Live Data и преобразует его в другую разновидность объекта Live Data.

Для создания `LiveData<String>` будет использоваться метод **`Transformations.map()`**. Метод получает аргумент `LiveData` и лямбда-выражение, которое указывает, как должен преобразоваться объект Live Data. Он возвращает новый объект `LiveData`.

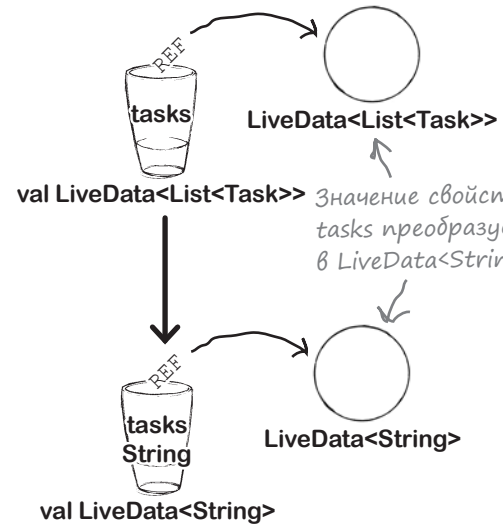
Например, для преобразования свойства `tasks` в `LiveData<String>` можно воспользоваться следующим кодом:

```
val tasksString = Transformations.map(tasks) {
    tasks -> formatTasks(tasks)
}
```

где `formatTasks()` — метод (который мы должны написать), форматирующий список задач из свойства `tasks` в `String`. Далее метод `Transformations.map()` упаковывает `String` в объект `LiveData`, чтобы он возвращал `LiveData<String>`.

Метод `Transformations.map()` наблюдает за переданным ему объектом `LiveData` и выполняет лямбда-выражение каждый раз, когда он получает оповещение об изменениях. Это означает, что в приведенном выше коде `tasksString` будет ***автоматически включать все новые записи задач, вводимые пользователем.***

На нескольких ближайших страницах мы добавим свойство `tasksString` в `TasksViewModel`, а затем воспользуемся связыванием данных для вывода его значения в макете `TasksFragment`. Такой подход означает, что все задачи пользователя будут отображаться в текстовом представлении, которое всегда остается актуальным.

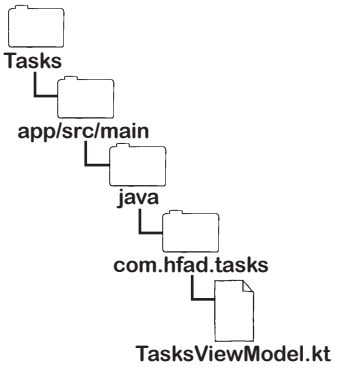


Подготовка
База данных
Вставка
Чтение записей

Обновление кода TasksViewModel

Начнем с обновления кода `TasksViewModel`, чтобы он включал новые свойства `tasks` и `tasksString`. Мы также добавим два метода: `formatTasks()` и `formatTask()`, которые упрощают преобразование `LiveData<List<Task>>` в `LiveData<String>`.

Ниже приведен полный код `TasksViewModel`. Обновите файл *TasksViewModel.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.tasks

import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""

    private val tasks = dao.getAll()
    val tasksString = Transformations.map(tasks) {
        tasks -> formatTasks(tasks)
    }

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }

    fun formatTasks(tasks: List<Task>): String {
        return tasks.fold("") {
                str, item -> str + '\n' + formatTask(item)
        }
    }

    fun formatTask(task: Task): String {
        var str = "ID: ${task.taskId}"
        str += '\n' + "Name: ${task.taskName}"
        str += '\n' + "Complete: ${task.taskDone}" + '\n'
        return str
    }
}
```

Импортируйте этот класс.

Получаем записи из базы данных.

Преобразует задачу в значение LiveData<String>, которое присваивается tasksString. Так как это объект Live Data, он будет автоматически обновляться при изменении данных.

Этот метод использует код Kotlin для форматирования списка задач в String. Здесь нет ничего специфического для Android.

Каждая задача форматируется в String.

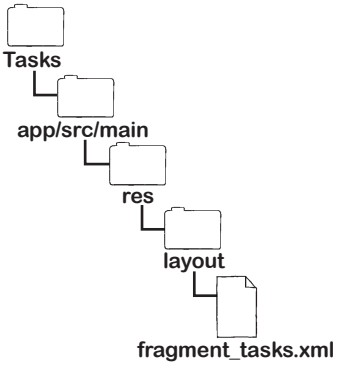
Подготовка
База данных
Вставка
Чтение записей

Свойство tasksString будет связано с текстовым представлением в макете

Затем мы используем механизм связывания данных для привязки текстового представления `tasks` в макете `TasksFragment` к свойству `tasksString` в `TasksViewModel`. Ниже приведен код решения этой задачи; обновите файл *fragment_tasks.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name"
            android:text="@={viewModel.newTaskName}" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task"
            android:onClick="@{() -> viewModel.addTask()}" />
```

Изменять код на этой странице не нужно.



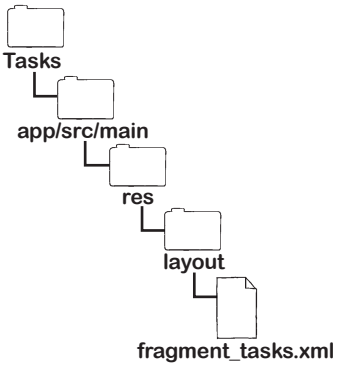
Продолжение на следующей странице.

Подготовка
База данных
Вставка
Чтение записей

fragment_tasks.xml (продолжение)...

```
        <TextView
            android:id="@+id/tasks"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@{viewModel.tasksString}" />
    </LinearLayout>
</layout>
```

Добавьте этот код.



И это все, что необходимо изменить в *fragment_tasks.xml* для вывода строки с задачами пользователя. Но прежде чем проводить тест-драйв приложения, необходимо внести еще одно изменение.

Макет должен реагировать на обновления данных Live Data

Как вы уже знаете, свойство `tasksString` относится к типу `LiveData<String>`; это означает, что оно автоматически учитывает любые обновления, которые вносятся в записи, хранящиеся в базе данных. Например, если пользователь вставляет новую запись, ее данные добавляются в значение `tasksString`.

Так как мы используем связывание данных для вывода значения свойства `tasksString` в текстовом представлении `tasks`, необходимо позаботиться о том, чтобы макет оповещался об обновлении значения свойства `tasksString`. Это означает, что новые записи начнут отображаться в текстовом представлении сразу же после их вставки.

Вы уже знаете, как заставить макет реагировать на обновления данных Live Data — назначением владельца жизненного цикла макета в коде фрагмента:

```
binding.lifecycleOwner = viewLifecycleOwner
```

Следовательно, код `TasksFragment` необходимо обновить и добавить в него эту строку. Полный код приведен на следующей странице, а потом мы рассмотрим, что же происходит при запуске приложения.

Часто Задаваемые Вопросы

В: Для чего еще используется метод `Transformations.map()`?

О: Так как результат `Transformations.map()` вычисляется заново при каждом получении новых данных, он может пригодиться, например, для вычислений.

Допустим, имеется свойство Live Data `rate`, и оно используется для некоторых вычислений. Например, можно выполнить вычисления внутри `Transformations.map()`, чтобы они выполнялись заново при каждом изменении свойства `rate`.

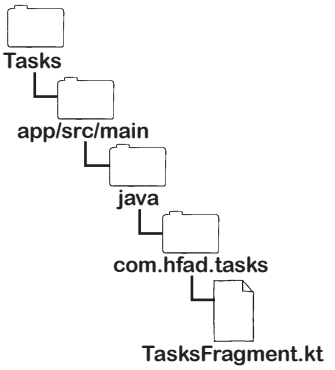
В: А списки всегда выводятся в виде отформатированных строк?

О: Нет. Мы выбрали этот вариант, потому что он позволяет относительно легко вывести набор записей. В этой главе будет описан более эффективный подход.

Подготовка
База данных
Вставка
Чтение записей

Полный код TasksFragment.kt

Ниже приведен полный код TasksFragment; обновите файл *TasksFragment.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

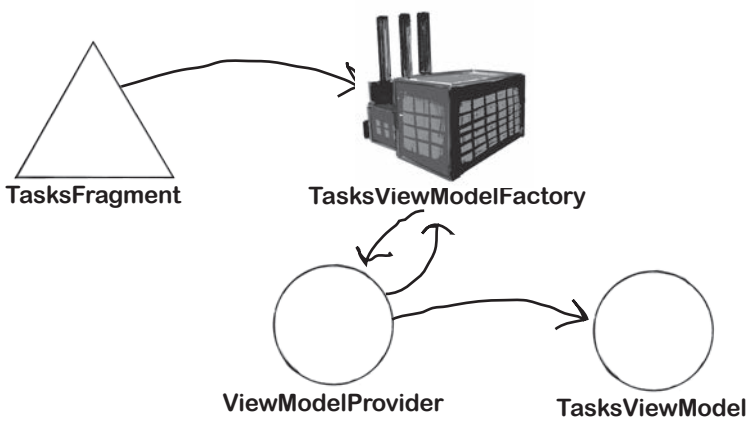
Назначаем владельца жизненного цикла макета, чтобы он мог реагировать на обновления данных Live Data.

Подготовка
База данных
Вставка
Чтение записей

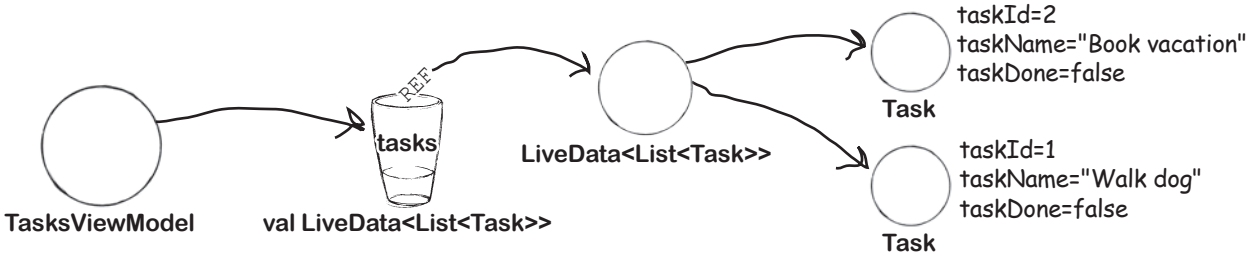
Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

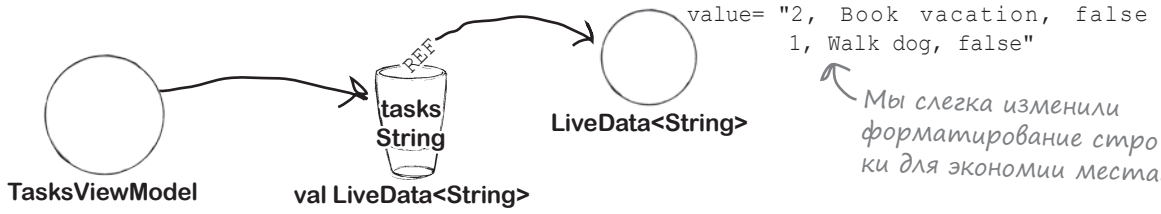
1. **TasksFragment создает объект TasksViewModelFactory, который используется провайдером модели представления для создания TasksViewModel.**



2. **Свойству tasks объекта TasksViewModel присваивается результат вызова метода getAll() объекта TaskDao, который возвращает LiveData<List<Task>>.** В нем содержится список всех записей из базы данных в виде данных Live Data.



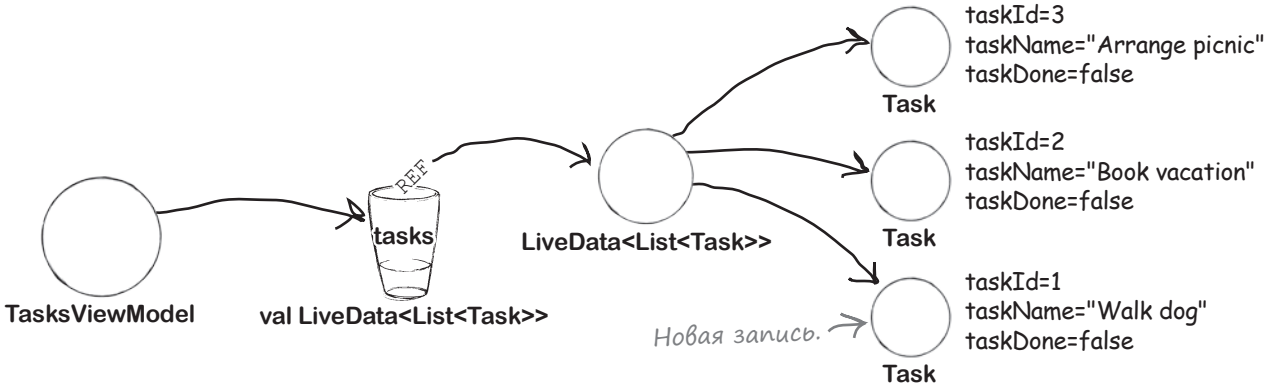
3. **Свойство tasksString объекта TasksViewModel использует метод Transformations.map() для преобразования значения свойства tasks в LiveData<String>.**
Метод возвращает записи свойства `tasks`, отформатированные в виде одной строки.



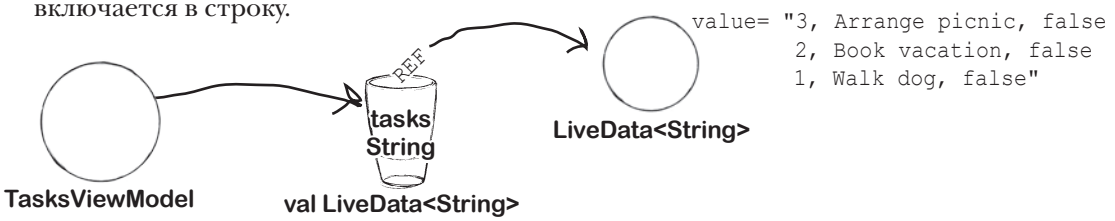
История продолжается

4 **Текстовое представление в макете использует связывание данных для вывода значения tasksString.**

5 **Пользователь вводит новую запись в базу данных.**
Свойство `tasks` объекта `TasksViewModel` автоматически обновляется (с использованием Live Data) для включения новой записи.



6 **Свойство tasksString реагирует на обновление свойства tasks.**
Благодаря использованию механизма Live Data новая запись автоматически включается в строку.



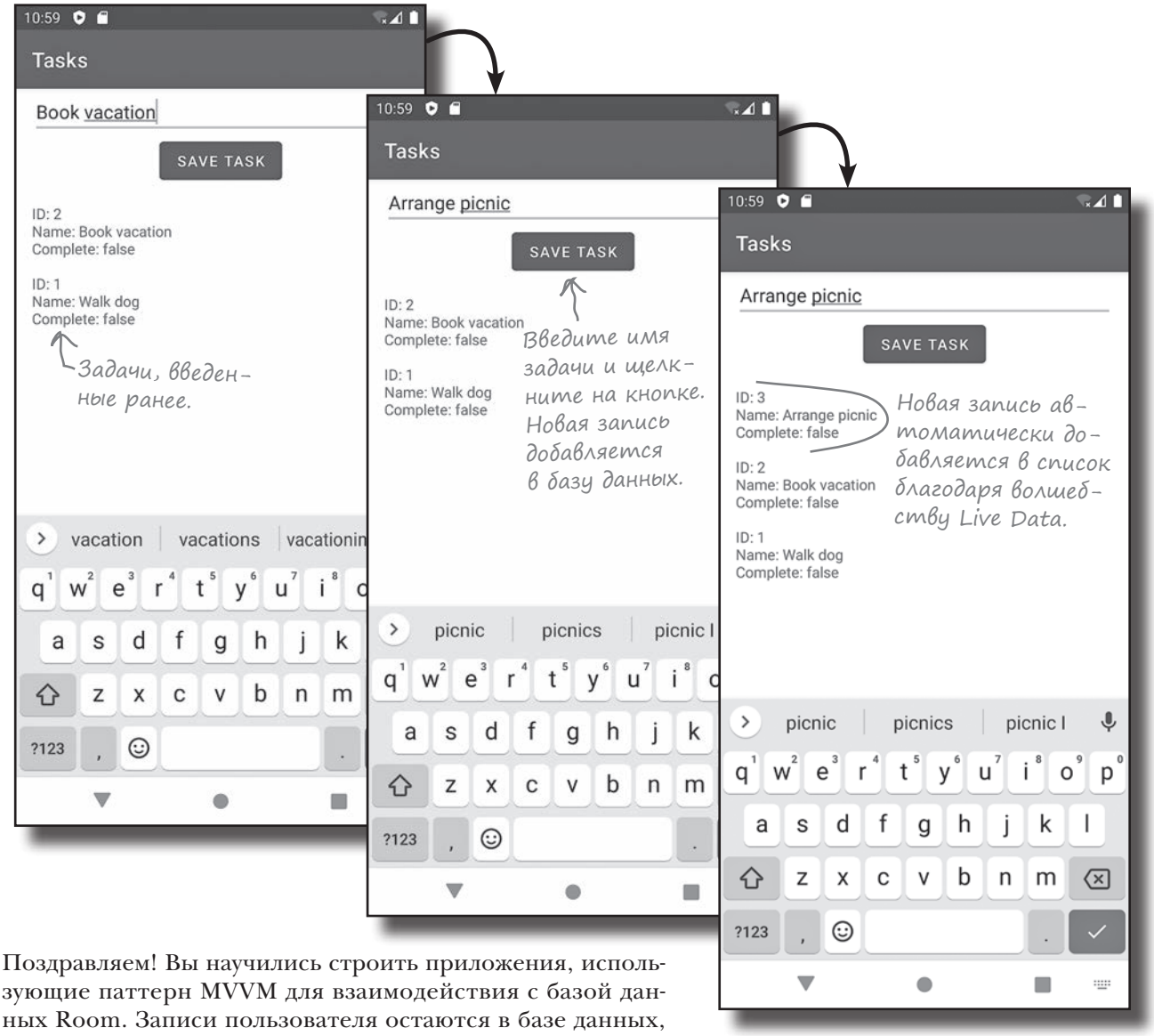
7 **Макет реагирует на обновление свойства tasksString.**
Запись, только что введенная пользователем, включается в текстовое представление.

Тест-драйв

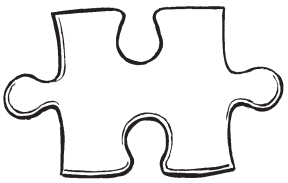
Подготовка
База данных
Вставка
Чтение записей

При запуске приложения, как и прежде, отображается фрагмент `TasksFragment`, но на этот раз выводится список задач, вводившихся ранее.

Когда вы вводите новую задачу, она добавляется в список сразу же после щелчка на кнопке Save Task.



Поздравляем! Вы научились строить приложения, использующие паттерн MVVM для взаимодействия с базой данных Room. Записи пользователя остаются в базе данных, чтобы они сохранялись при закрытии приложения, а приложение отображает новые записи сразу же после вставки.



Смешанные сообщения

Ниже приведен код модели представления, но код `refresh()` отсутствует. Соедините каждый потенциальный блок кода (слева) с итоговым значением свойства `result`, если блок-кандидат будет добавлен в метод `refresh()` и метод будет вызван однократно. Не все кандидаты будут использованы, а некоторые могут быть использованы многократно.

```
class MyViewModel() : ViewModel() {
    private val rate = MutableLiveData<Int>(1)
    private val a = MutableLiveData<Int>(0)
    private val b = MutableLiveData<Int>(0)
    val result = Transformations.map(rate) {
        rate -> rate * ((a.value ?: 0) + (b.value ?: 0))
    }

    fun refresh() {
        [                                    ]
    }
}
```

Здесь размещается код блока-кандидата.

Кандидаты:

```
a.value = 2
```

```
rate.value = 2
b.value = 2
```

```
a.value = 2
rate.value = 2
```

```
a.value = 2
rate.value = 2
b.value = 2
```

```
a.value = 2
b.value = 2
rate.value = 2
```

Возможные значения result:

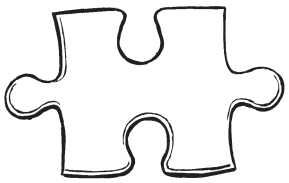
0

2

4

6

8



Смешанные сообщения. Решение

Ниже приведен код модели представления, но код `refresh()` отсутствует. Соедините каждый потенциальный блок кода (слева) с итоговым значением свойства `result`, если блок-кандидат будет добавлен в метод `refresh()` и метод будет вызван однократно. Не все кандидаты будут использованы, а некоторые могут быть использованы многократно.

```
class MyViewModel() : ViewModel() {
    private val rate = MutableLiveData<Int>(1)
    private val a = MutableLiveData<Int>(0)
    private val b = MutableLiveData<Int>(0)
    val result = Transformations.map(rate) {
        rate -> rate * ((a.value ?: 0) + (b.value ?: 0))
    }

    fun refresh() {
        [                                    ]
    }
}
```

Здесь размещается код блока-кандидата.

Кандидаты:

```
a.value = 2
```

```
rate.value = 2
b.value = 2
```

```
a.value = 2
rate.value = 2
```

```
a.value = 2
rate.value = 2
b.value = 2
```

```
a.value = 2
b.value = 2
rate.value = 2
```

Возможные значения result:

0

2

4

6

8

Значение result обновляется только при изменении значения rate. Если a или b изменится после обновления rate, новые значения не будут отражены в свойстве result.

Компания компонентов, облаченных в маскарадные костюмы, развлекается игрой «Кто я?». Игрок дает подсказку, а остальные на основании сказанного им пытаются угадать, кого он изображает. Будем считать, что игроки всегда говорят правду о себе. Заполните пропуски справа именами одного или нескольких участников. Кроме того, для каждого участника напишите, является ли компонент частью уровня модели, представления или модели представления в архитектуре приложения.

Сегодняшние участники:

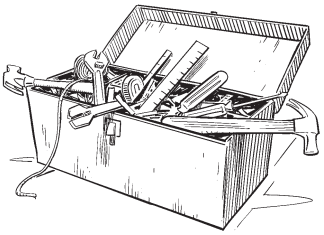
Активность Фрагмент Макет Модель представления База данных Сущность DAO

Решение

	Имя	Модель, представление или модель представления?
Я определяю внешний вид экранов представления.	Макет	Представление
Я сохраняю данные после завершения приложения.	База данных	Модель
У меня есть жизненный цикл, но я не могу существовать сам по себе.	Фрагмент	Представление
Я используюсь для бизнес-логики.	Модель представления	Модель представления
Room использует меня для создания таблицы.	Сущность	Модель
У меня есть жизненный цикл, и я являюсь разновидностью контекста.	Активность	Представление
Я помогаю взаимодействовать с информацией в базе данных.	DAO	Модель

ГЛАВА 14

Ваш инструментарий Android



Глава 14 осталась позади, а ваш инструментарий пополнился библиотекой хранения данных Room.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Room — библиотека хранения данных, работающая на базе SQLite.
- Структура приложения Room обычно строится по архитектурному паттерну проектирования MVVM, что означает «модель–представление–модель представления».
- Таблицы баз данных определяются классом данных с аннотациями. Аннотация `@Entity` отдает команду Room использовать класс для создания таблицы. Аннотация `@PrimaryKey` определяет столбец первичного ключа. Аннотация `@ColumnInfo` определяет имена столбцов.
- Имена методов доступа к данным определяются в интерфейсе с аннотациями. Аннотация `@Dao` отдает команду Room использовать методы для доступа к данным. Аннотации `@Insert`, `@Update`, `@Delete` и `@Query` определяют операции с данными.
- База данных определяется абстрактным классом с аннотацией `@Database`.
- При обновлении схемы базы данных необходимо увеличить номер версии базы данных.
- Код доступа к данным должен выполняться в фоновом режиме. Для этого можно преобразовать методы доступа к данным, которые не возвращают данные Live Data, в сопрограммы.
- Для преобразования объектов Live Data можно воспользоваться методом `Transformations.map()`. Метод наблюдает за объектом Live Data, выполняет лямбда-выражение при изменении значения объекта и возвращает другой объект Live Data.

Экономия и переработка

Списки данных являются важнейшей частью многих приложений. В этой главе мы покажем, как создать такой список с использованием **представлений с переработкой**: **невероятно гибкого** способа построения **прокручиваемых списков.** Вы научитесь создавать **гибкие макеты** для ваших списков, включая текстовые представления, флажки и многое другое. Вы узнаете, **как создавать адаптеры**, которые **отображают ваши данные** в представлениях с переработкой так, как вам нужно. Вы научитесь использовать **карточные представления** для оформления данных **с имитацией объема**. Наконец, мы покажем, как **менеджеры макетов могут полностью изменить внешний вид вашего списка всего одной или двумя строками кода**. Давайте займемся переработкой.

Как в настоящее время выглядит приложение Tasks

В предыдущей главе мы построили приложение Tasks, которое предоставляет возможность вводить записи в базе данных Room. Приложение выводит список записей в виде отформатированной строки, которая выглядит так:

TasksFragment — главный экран приложения, отображаемый внутри MainActivity. Он используется для ввода новых записей пользователем и их вывода.

Мы решили выводить записи с описаниями задач в отформатированной строке, потому что это относительно простой и быстрый способ просмотра записей, добавленных в базу данных.

Тем не менее список выглядит довольно уныло. Нельзя ли как-то украсить его?

Список можно преобразовать в представление с переработкой

Вместо того чтобы отображать список задач в виде отформатированной строки, его можно изменить, чтобы он выглядел примерно так:

Как видите, для данных каждой записи отображается текстовое представление и флажок (вместо простого текста). Элементы на карточках образуют сетку с возможностью прокрутки.

Для создания подобных списков используется **представление с переработкой.** Что это такое?

Для чего нужны представления с переработкой?

Представления с переработкой также являются частью Android Jetpack.

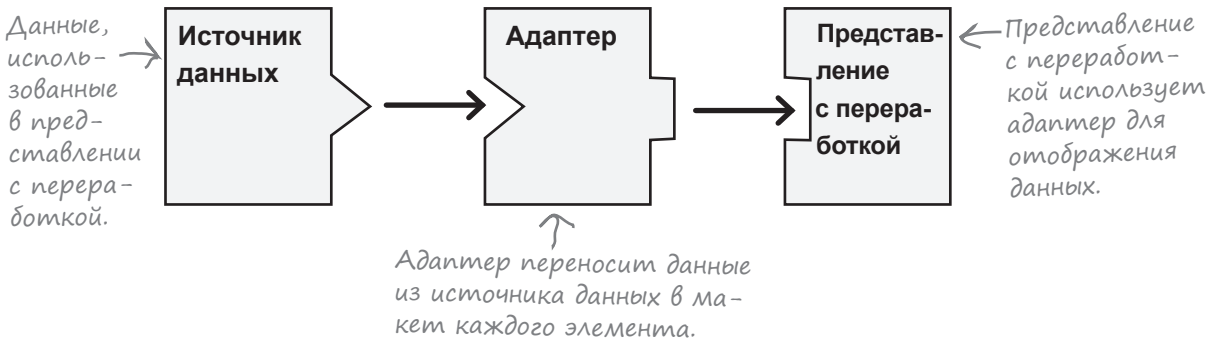
Представления с переработкой предоставляют более совершенный и гибкий способ отображения списка данных по сравнению с использованием простой отформатированной строки. Они обладают целым рядом преимуществ:

- **Более функциональный пользовательский интерфейс для элементов списка.** Каждый элемент отображается в макете, что позволяет использовать для вывода данных такие компоненты, как текстовые представления, графические представления и флажки.
- **Гибкие средства позиционирования элементов.** Представления с переработкой работают в сочетании с менеджерами макетов, которые позволяют позиционировать представления в вертикальном или горизонтальном списке, сетке или неравномерной сетке, у которой элементы имеют разную высоту.
- **Возможность использования для навигации.** Для элементов можно предусмотреть обработку щелчков, чтобы по щелчку происходил переход к другому фрагменту.
- **Эффективный механизм отображения больших наборов данных.** Представления с переработкой используют небольшое количество представлений для создания иллюзии большой группы представлений, выходящих за пределы экрана. Когда каждый элемент выходит за пределы экрана, его представление повторно используется — или *перерабатывается* — для элементов, которые вошли на экран в результате прокрутки.

Представления с переработкой получают свои данные от адаптера

Каждое представление с переработкой, которое вы создаете, использует адаптер для отображения своих данных. Адаптер использует данные из источника данных (такого, как база данных) и связывает их с представлениями в макете элемента. После этого представление с переработкой отображает элементы на экране в виде списка с возможностью прокрутки.

Схема взаимодействия источника данных, адаптера и представления с переработкой выглядит так:

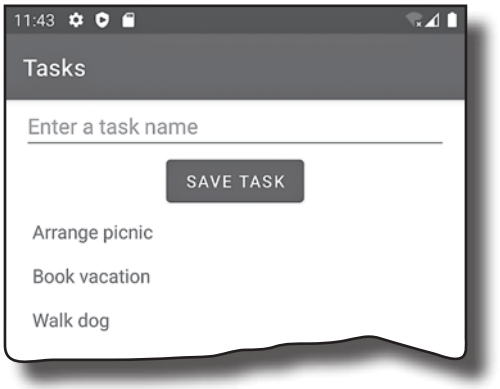


Мы добавим представление с переработкой в приложение Tasks. Давайте рассмотрим основные этапы решения этой задачи.

Что мы собираемся сделать

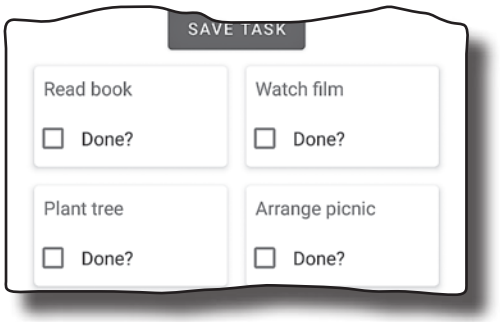
Чтобы добавить представление с переработкой в приложение Tasks, выполните следующие действия:

1. **Создание представления с переработкой, в котором выводится список имен задач.** Начнем с создания простого представления с переработкой, в котором выводятся только имена задач. С относительно простой первой версией вам будет проще понять, как конструируется каждая часть представления с переработкой и как они взаимодействуют друг с другом.



Первая версия представления с переработкой отображает задачи в виде списка строк.

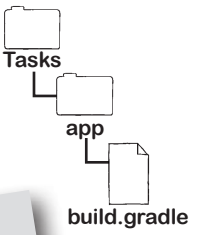
2. **Обновление представления с переработкой для отображения карточек, образующих сетку.** После того как простое представление с переработкой заработает, мы изменим его так, чтобы имя каждой задачи и признак ее завершения отображались в карточном представлении, а карточные представления формировали сетку.



Вторая версия представления с переработкой будет использовать более сложный макет.

Добавление зависимости для представления с переработкой в файл build.gradle приложения

Прежде чем строить представление с переработкой, необходимо добавить зависимость для библиотеки представлений с переработкой в файл *build.gradle* приложения. Откройте приложение Tasks, затем откройте файл *Tasks/app/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `dependencies`:



Добавляет библиотеку представлений с переработкой.

```
dependencies {
    ...
    implementation 'androidx.recyclerview:recyclerview:1.2.1'
    ...
}
```

Синхронизируйте это изменение с остальными частями приложения.

После того как зависимость будет добавлена, перейдем к построению представления с переработкой.

Не забудьте!

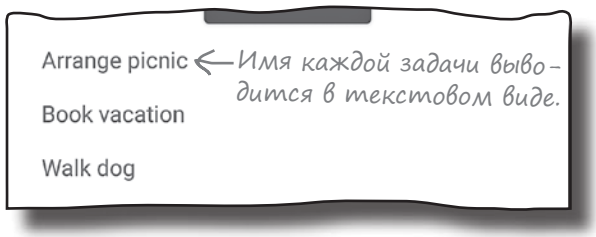
Мы собираемся изменить приложение Tasks, созданное в главе 14. Откройте проект этого приложения.

Список задач
Сетка карточек

Чтобы сообщить представлению с переработкой, как отображать каждый элемент...

Прежде всего необходимо сообщить представлению с переработкой, как выводить каждую запись.

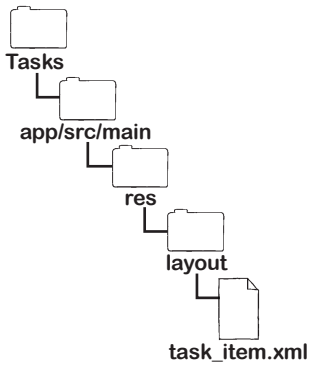
В первой версии приложения имя каждой задачи должно выводиться в представлении с переработкой, чтобы оно выглядело так:



Как же это сделать?

...следует определить файл макета

Для определения того, как должен быть оформлен каждый элемент в представлении с переработкой, используется файл макета. Представление с переработкой использует этот файл макета для отображения каждого элемента. Например, если файл макета состоит из единственного текстового представления, текстовое представление будет отображаться для каждого элемента в списке представления с переработкой.

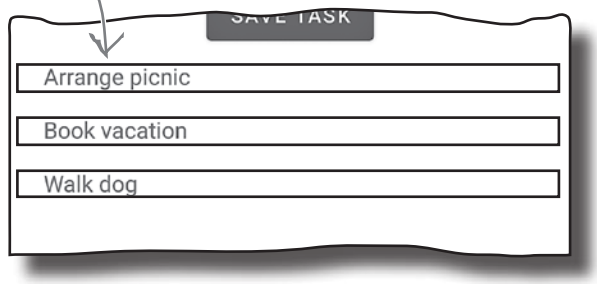


Чтобы создать файл макета, выделите папку *Tasks/app/src/main/res/layout* на панели проекта и выберите команду File→New→Layout Resource File. Введите имя файла «task_item» и щелкните на кнопке OK.

В первой версии приложения имя каждой задачи представления с переработкой должно отображаться в одном текстовом представлении; добавим текстовое представление в только что созданное текстовое представление. Для этого обновите код *task_item.xml* и приведите его к следующему виду:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:padding="8dp" />
```

Каждый элемент отображается в текстовом представлении.



И это весь код, необходимый для того, чтобы сообщить представлению с переработкой, как должен отображаться каждый элемент. На следующем этапе мы создадим адаптер представления с переработкой.

Адаптер добавляет данные в представление с переработкой

Как говорилось ранее, когда вы используете представление с переработкой в своем приложении, для него необходимо создать адаптер.

Адаптер представления с переработкой выполняет две основные функции: создание каждого из представлений, видимых в представлении с переработкой, и отображение данных в каждом представлении. Для приложения Tasks необходимо определить адаптер, который использует *task_item.xml* для создания набора текстовых представлений (по одному для каждой отображаемой записи) и помещает в каждое представление имя задачи.

Адаптер соединяет источник данных и представление с переработкой.

Адаптер будет построен на нескольких следующих страницах. Сначала выделим основные этапы его создания.

1. **Определение типа данных, с которыми должен работать адаптер.**
   Наш адаптер должен работать с данными `Task`, поэтому мы указываем, что он использует `List<Task>`.
2. **Определение держателя представления для адаптера.**
   Управляет тем, как должно заполняться каждое представление в макете элемента.
3. **Заполнение макета каждого элемента.**
   Когда представлению с переработкой потребуется вывести каждый элемент, мы заполняем экземпляр *task_item.xml* для этого элемента.
4. **Отображение данных каждого элемента в макете.**
   Для этого мы добавим значение свойства `taskName` каждого объекта `Task` в текстовое представление макета.

Начнем с создания файла для адаптера.

Создание файла адаптера

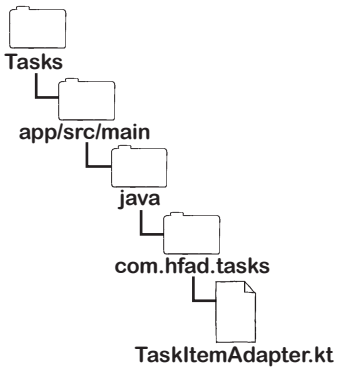
Мы создадим адаптер для представления с переработкой `TaskItemAdapter`. Для этого выделите пакет *com.hfad.tasks* в папке *app/src/main/java*, а затем выберите команду File→New→Kotlin Class/File. Введите имя файла «TaskItemAdapter» и выберите вариант Class.

После того как файл будет создан, обновите его код, чтобы он расширял класс `RecyclerView.Adapter`:

```
package com.hfad.tasks

import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter() {
}
```

Импортируйте этот класс.

Расширяет RecyclerView.Adapter.



Класс преобразуется в адаптер, который может использоваться представлением с переработкой.

Сообщаем адаптеру, с какими данными он должен работать

Когда вы определяете адаптер представления с переработкой, необходимо сообщить ему, какие данные должны добавляться в представление с переработкой. Для этого в адаптер будет добавлено свойство, определяющее тип данных.

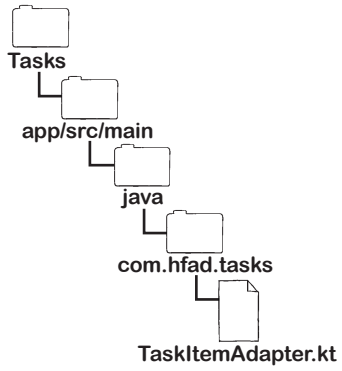
В приложении Tasks представление с переработкой должно отображать список записей задач, поэтому мы добавим в адаптер свойство `List<Task>` с именем `data`. Также будет добавлен специальный метод записи, который вызывает `notifyDataSetChanged()` при обновлении свойства; тем самым он сообщает представлению с переработкой, что данные изменились, чтобы оно могло перерисовать себя.

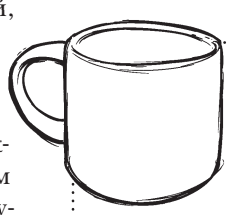
Ниже приведен обновленный код `TaskItemAdapter`; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):

```
...
class TaskItemAdapter : RecyclerView.Adapter() {

    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

}
```

Добавляет свойство data и его метод записи.

Сообщает представлению с переработкой, что данные изменились.





Расслабьтесь

Если в коде `TaskItemAdapter` будут обнаружены ошибки, не волнуйтесь.

Код еще не завершен, и сообщения об ошибках исчезнут после того, как мы его допишем.

Переопределение метода getItemCount()

Затем необходимо переопределить метод **`getItemCount()`** адаптера. Тем самым вы сообщаете адаптеру, сколько элементов данных будет отображаться, чтобы эта информация была доступна представлению с переработкой.

В коде `TaskItemAdapter` мы используем свойство `List<Task>` с именем `data` для элементов данных представления с переработкой, что позволяет использовать конструкцию `data.size` для определения количества элементов.

Ниже приведен метод `getItemCount()` (выделен жирным шрифтом), который добавляется в *TaskItemAdapter.kt*:

```
...
class TaskItemAdapter : RecyclerView.Adapter() {
    ...

    override fun getItemCount() = data.size

}
```

Это весь код, необходимый для метода getItemCount().

Итак, мы определили тип данных, с которыми работает адаптер. Теперь он будет использован для заполнения текстового представления макета. Для этого мы определим **держатель представления** адаптера.

Список задач
Сетка карточек

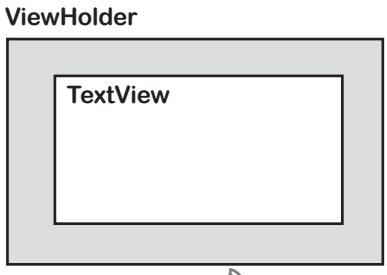
Определение держателя представления для адаптера

Держатель представления (view holder) содержит информацию о том, как должно отображаться представление в макете элемента, и о его позиции в представлении с переработкой. Его можно рассматривать как держатель корневого представления макета элемента — макета, определяющего, как представление с переработкой должно отображать каждый элемент.

В приложении Tasks представление с переработкой должно использовать файл макета *task_item.xml* для отображения записей с описаниями задач. Корневым представлением макета является `TextView`, а следовательно, необходимо определить держатель представления, который работает с текстовыми представлениями.

Чтобы определить держатель представления, добавьте внутренний класс в класс адаптера, расширяющий `RecyclerView.ViewHolder`. Он включает конструктор, задающий тип корневого представления макета (в данном случае `TextView`). Определение класса адаптера также должно обновляться для определения имени класса адаптера.

Ниже приведена новая версия кода `TaskItemAdapter`; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):



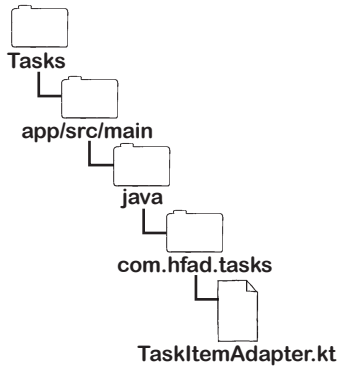
Каждый держатель представления содержит корневое представление макета для каждого элемента. В данном случае корневым является текстовое представление.

```
package com.hfad.tasks

import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    class TaskItemViewHolder(val rootView: TextView)
                : RecyclerView.ViewHolder(rootView) {
    }
}
```

Импортируем этот класс.

Используем обобщения для указания того, что TaskItemAdapter работает со своим внутренним классом TaskItemViewHolder.

Добавляется внутренний класс TaskItemViewHolder. Его код будет дополнен на нескольких ближайших страницах.



После определения держателя представления необходимо указать, какой макет он использует, переопределяя метод `onCreateViewHolder()` адаптера.

Список задач
Сетка карточек

Переопределение метода onCreateViewHolder()

Метод **onCreateViewHolder()** адаптера вызывается каждый раз, когда представлению с переработкой потребуется новый держатель представления. Представление с переработкой многократно вызывает метод в момент своего конструирования для построения набора держателей представлений, которые будут отображаться на экране.

Метод `onCreateViewHolder()` должен решать две задачи: заполнять макет, используемый для каждого элемента (в данном случае *task_item.xml*), и возвращать держатель представления. Ниже приведен код для адаптера `TaskItemAdapter`; обновите файл *TaskItemAdapter.kt* () (изменения выделены жирным шрифтом):

Расслабьтесь

Код адаптера на первый взгляд смотрится устрашающе, так как адаптер должен решать несколько разных задач.

Не торопитесь, понемногу двигайтесь вперед, и все будет нормально.

```
...
import android.view.LayoutInflater
import android.view.ViewGroup

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    class TaskItemViewHolder(val rootView: TextView)
                            : RecyclerView.ViewHolder(rootView) {
        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as TextView
                return TaskItemViewHolder(view)
            }
        }
    }
}
```

Импортируем эти классы.

Переопределяем этот метод.

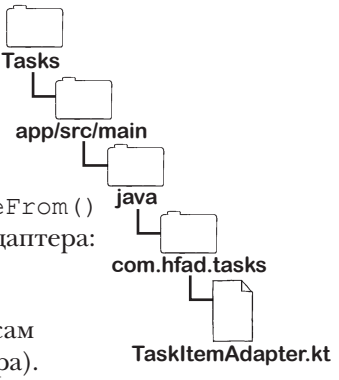
parent соответствует представлению с переработкой.

Позволяет использовать разные макеты для разных элементов.

onCreateViewHolder() будет вызывать метод inflateFrom() держателя представления.

Метод inflateFrom() добавляется в объект-компаньон. Это означает, что метод может вызываться без предварительного создания объекта TaskItemViewHolder.

Метод заполняет макет элемента и использует его для создания TaskItemViewHolder.



Как видите, мы поместили код заполнения *task_item.xml* в новый метод `inflateFrom()` в `TaskItemViewHolder`, который вызывается методом `onCreateViewHolder()` адаптера:

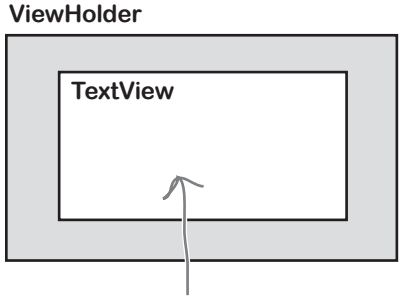
```
TaskItemViewHolder.inflateFrom(parent)
```

Этот подход возлагает ответственность за макет держателя представления на сам держатель представления (вместо заполнения макета в основном блоке кода адаптера).

Добавление данных в представление макета

Остается добавить в адаптер код, определяющий, как записи должны отображаться в макете держателя представления. Для этого мы переопределим метод **`onBindViewHolder()`** адаптера, который вызывается каждый раз, когда представлению с переработкой потребуется отобразить данные. Он получает два параметра: держатель представления, с которым связываются данные, и позицию данных в наборе данных.

В приложении Tasks требуется взять объект `Task` в определенной позиции свойства `data` адаптера (`List<Task>`) и отобразить его значение `taskName` в макете держателя представления. Ниже приведен код решения этой задачи для `TaskItemAdapter`; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):



При вызове метода onBindViewHolder() адаптера он должен добавить имя задачи в текстовое представление держателя представления.

```
...
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {

    var data = listOf<Task>()
    ...

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView: TextView)
                            : RecyclerView.ViewHolder(rootView) {
        ...

        fun bind(item: Task) {
            rootView.text = item.taskName
        }
    }
}
```

Переопределяем этот метод.

Представление, в которое метод должен добавить данные.

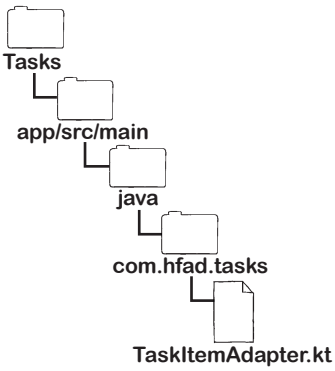
Позиция элемента в данных.

Получаем элемент, который необходимо отобразить.

Вызываем метод bind() держателя представления для этого элемента.

Этот метод добавляется в держатель представлений.

Имя задачи добавляется в корневое представление макета (текстовое представление).

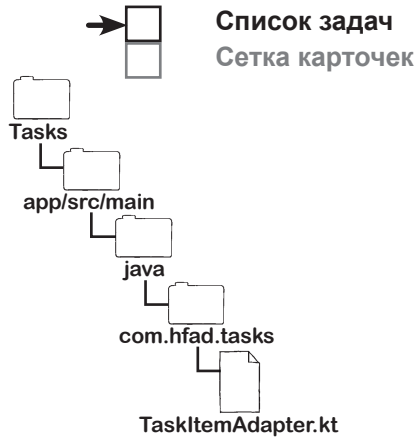


Как видите, текстовое представление макета задается в новом методе `bind()`, который добавляется в `TaskItemViewHolder`. Затем метод `onBindViewHolder()` адаптера вызывает его при каждом выполнении. Мы используем этот подход, потому что с ним держатель представления отвечает за заполнение своего макета (вместо адаптера).

И это весь код, необходимый для написания класса `TaskItemAdapter` и его внутреннего класса `TaskItemViewHolder`. Пора рассмотреть полный код.

Полный код TaskItemAdapter.kt

Ниже приведен полный код `TaskItemAdapter`; убедитесь в том, что в файл *TaskItemAdapter.kt* включены все показанные изменения:




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView: TextView)
                            : RecyclerView.ViewHolder(rootView) {
        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as TextView
                return TaskItemViewHolder(view)
            }
        }

        fun bind(item: Task) {
            rootView.text = item.taskName
        }
    }
}
```

Для данных представления с переработкой.

Количество элементов.

Вызывается каждый раз, когда потребуется создать держатель представления.

Вызывается, когда данные должны отображаться в держателе представления.

Определяет держатель представления.

Создает каждый держатель представления и заполняет его макет.

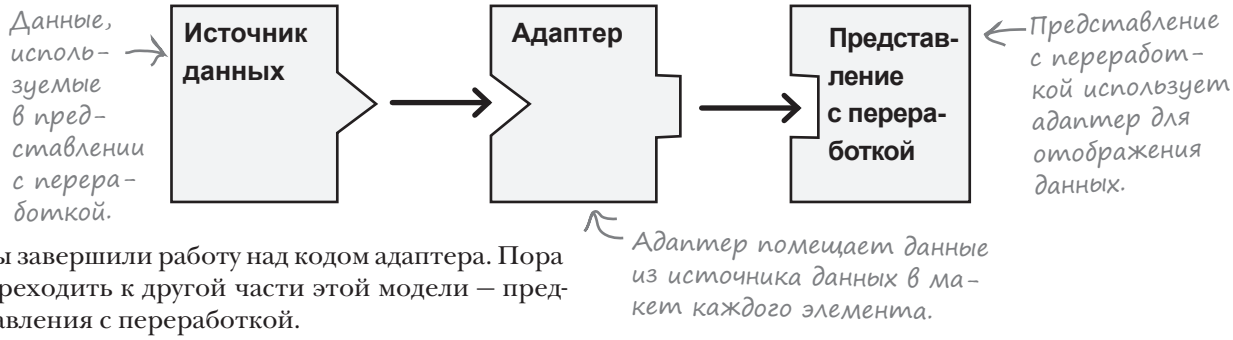
Данные добавляются в макет держателя представления.

Код адаптера готов

Мы написали весь код, необходимый для `TaskItemAdapter`.
Этот код:

1. **Сообщает, что работает с данными Task**
   Для этого мы определили свойство `List<Task>` с именем `data`.
2. **Использует держатель представлений с именем TaskItemViewHolder.**
   Класс `TaskItemViewHolder` был включен в `TaskItemAdapter` как внутренний класс.
3. **Заполняет макет каждого элемента.**
   Это происходит при вызове метода `onCreateViewHolder()`.
4. **Выводит данные каждого элемента в макете.**
   Для этого используется метод `onBindViewHolder()`.

Как говорилось ранее, адаптер соединяет источник данных и представление с переработкой. Источник данных, адаптер и представление с переработкой взаимодействуют по следующей схеме:



Мы завершили работу над кодом адаптера. Пора переходить к другой части этой модели — представления с переработкой.

Часто Задаваемые Вопросы

В: Почему представление называется представлением с переработкой?

О: Потому что оно перерабатывает, то есть повторно использует свои представления. Оно становится очень эффективным способом отображения списка данных.

В: Мы также добавили в адаптер внутренний класс держателя представления. Не напомните, что это такое?

О: Это держатель для корневого представления макета каждого элемента. Каждому адаптеру представления с переработкой, которое вы создадите, необходим держатель представления.

В: Он непременно должен быть внутренним классом?

О: Нет. Так как каждый держатель представления обычно связывается с одним адаптером, он часто определяется в виде внутреннего класса, но это не обязательно.

В: Почему мы добавили методы `inflateFrom()` и `bind()` в держатель представления? Разве нельзя было включить их код в методы `onCreateViewHolder()` и `onBindViewHolder()` адаптера?

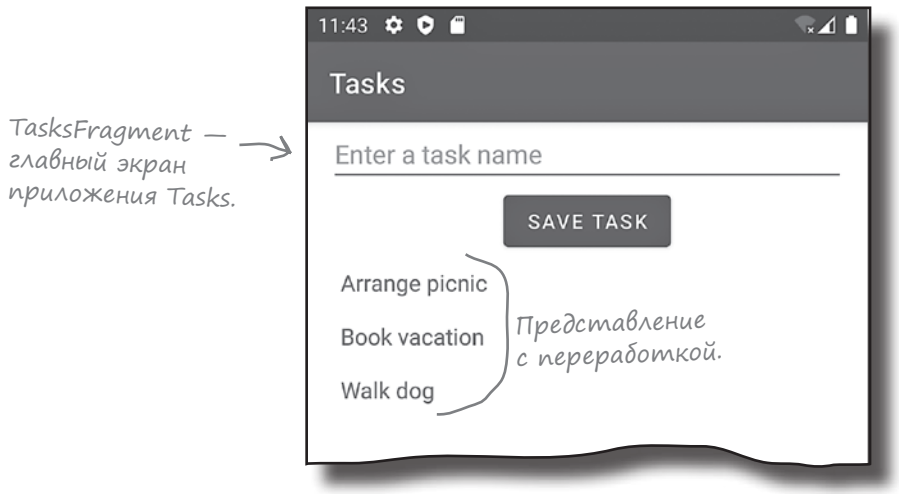
О: Мы добавили эти методы в держатель представления, потому что хотели хранить весь код, управляющий держателями представлений, в одном месте. Если бы код остался в основном теле адаптера, в программе не было бы такого четкого разделения обязанностей. Такой подход особенно полезен, если вы не хотите ограничиваться одним текстовым представлением для каждого элемента представления с переработкой.

Список задач
Сетка карточек

В приложении должно отображаться представление с переработкой

Следующее, что необходимо сделать, отобразить представление с переработкой в `TasksFragment` (главном экране приложения Tasks) и заставить его использовать только что созданный нами адаптер.

Кратко напомним, как должно выглядеть представление с переработкой:



Добавление представления с переработкой в макет

Чтобы в приложении отображалось представление с переработкой, добавьте элемент `<androidx.recyclerview.widget.RecyclerView>` в файл макета фрагмента. Код должен выглядеть так:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/tasks_list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

Представление с переработкой добавляется в макет.

Элементы образуют вертикальный список.

Строка:

```
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
```

указывает, что в представлении с переработкой используется менеджер макета, который определяет, как представление с переработкой позиционирует свои элементы. В данном случае используется менеджер линейного макета; это означает, что элементы представления с переработкой будут отображаться в вертикальном списке со строками полной длины.

Менеджеры макетов более подробно рассматриваются далее в этой главе.

И это весь код, который необходим для добавления представления с переработкой в макет `TasksFragment`. Давайте посмотрим, как он выглядит в целом.

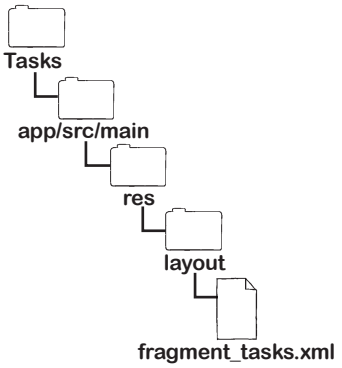
Список задач
Сетка карточек

Полный код fragment_tasks.xml

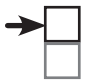
Ниже приведен полный код *fragment_tasks.xml* (макет `TasksFragment`). Как видите, мы заменили текстовое представление представлением с переработкой; обновите код *fragment_tasks.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:hint="Enter a task name"
            android:text="@={viewModel.newTaskName}" />

        <Button
            android:id="@+id/save_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Save Task"
            android:onClick="@{() -> viewModel.addTask()}" />
```

Добавляем пространство имен приложения.



Продолжение на следующей странице.

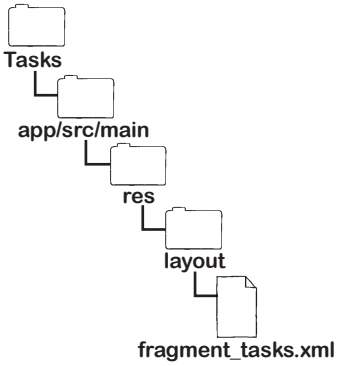


Список задач
Сетка карточек

fragment_tasks.xml (продолжение)

Текстовое представление удаляется.

```
<TextView
        android:id="@+id/tasks"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@{viewModel.tasksString}" />
```



Добавляем представление с переработкой.

```
<androidx.recyclerview.widget.RecyclerView
        android:id="@+id/tasks_list"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
    </LinearLayout>
</layout>
```

И это весь код, необходимый для добавления представления с переработкой в макет `TasksFragment`. Затем нужно отдать команду представлению с переработкой использовать созданный нами адаптер.

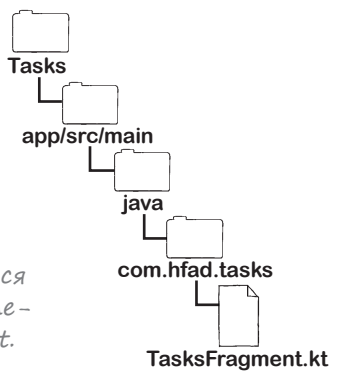
Использование адаптера представлением с переработкой

Чтобы отдать команду представлению с переработкой использовать адаптер, следует создать экземпляр адаптера и присоединить его к представлению с переработкой. Это делается в коде Kotlin фрагмента.

В нашем случае нужно отдать команду представлению с переработкой использовать `TaskItemAdapter`. Для этого следует включить в метод `onCreateView()` объекта `TasksFragment` следующую строку (выделенную жирным шрифтом):

```
...
override fun onCreateView(...): View? {
    ...
    val adapter = TaskItemAdapter()
    binding.tasksList.adapter = adapter
    ...
}
```

Адаптер добавляется в представление с переработкой tasksList.



Этот код будет добавлен в `TasksFragment` на следующей странице.

Список задач
Сетка карточек

Обновленный код TasksFragment.kt

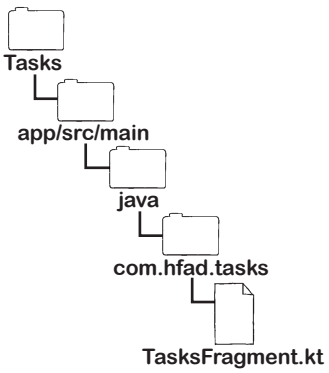
Ниже приведен код `TasksFragment`; обновите файл *TasksFragment.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.tasks
...
class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Команды импортирования не приводятся, потому что ничего нового импортировать не нужно.

Создание TaskItemAdapter.

Адаптер присоединяется к представлению с переработкой.



Что же дальше?

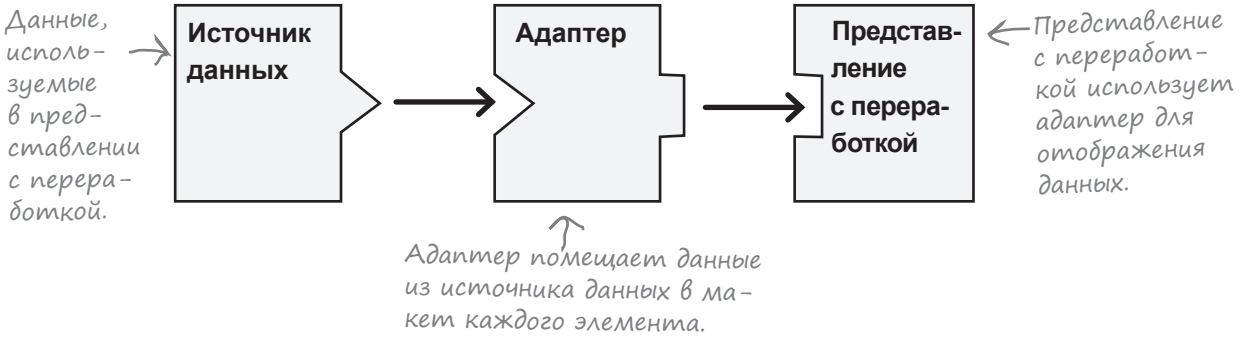
Список задач
Сетка карточек

Представление с переработкой добавляется в макет TasksFragment

Мы написали весь код, необходимый для отображения представления с переработкой в макете `TasksFragment`, и отдали ему команду использовать адаптер `TaskItemAdapter`. Остается сделать последний шаг: соединить адаптер с источником данных.

Как вы узнали ранее, адаптер использует данные из источника данных (например, базы данных) и связывает их с представлениями в макете элемента. Тогда представление с переработкой отображает элементы на экране устройства.

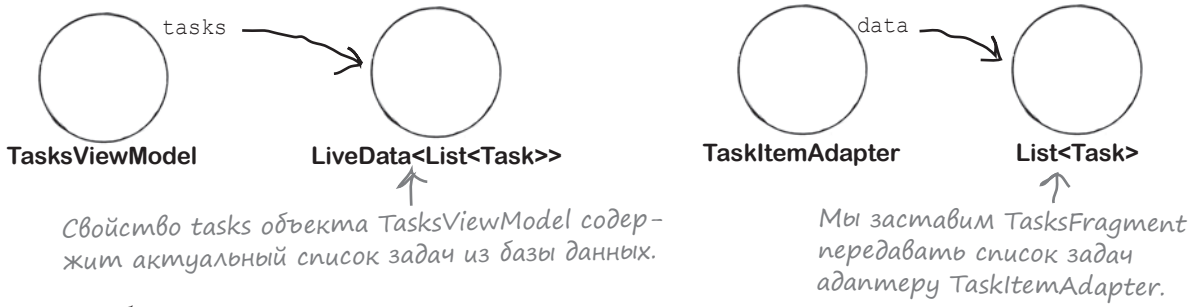
Источник данных, адаптер и представление с переработкой взаимодействуют по следующей схеме:



Таким образом, чтобы отображать данные в представлении с переработкой приложения Tasks, необходимо сообщить объекту `TaskItemAdapter`, какие данные он должен использовать.

TasksFragment будет поставлять данные задач адаптеру TaskItemAdapter

Чтобы сообщить `TaskItemAdapter`, какие данные задач следует использовать, мы заставим `TasksFragment` обновлять его свойство `data` с `List<Task>`. `TasksFragment` получает этот список из свойства `tasks` объекта `TasksViewModel`:



Для этого необходимо сначала предоставить `TasksFragment` доступ к свойству `tasks` объекта `TasksViewModel`.

Обновление кода TasksViewModel.kt

Как вы помните, свойство `tasks` объекта `TasksViewModel` в настоящее время помечено модификатором private. Этот модификатор необходимо удалить, чтобы код `TasksFragment` мог получить значение свойства.

Также будет удален весь код, добавленный в предыдущей главе для преобразования данных задач в отформатированную строку: этот код стал лишним, так как мы используем представление с переработкой для отображения `List<Task>`.

Ниже приведена новая версия кода `TasksViewModel`; обновите файл *TasksViewModel.kt* (изменения выделены жирным шрифтом):

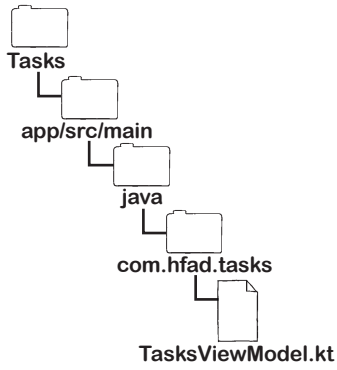
```
package com.hfad.tasks

import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""
    private val tasks = dao.getAll()
    val tasksString = Transformations.map(tasks) {
            tasks -> formatTasks(tasks)
    }

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }

    fun formatTasks(tasks: List<Task>): String {...}

    fun formatTask(task: Task): String {...}
}
```

Удалите эту строку, она более не нужна.

Удалите модификатор private.

Свойство tasksString более не понадобится, его можно удалить.

Удалите эти два метода.



Фрагмент `TasksFragment` получил доступ к свойству `tasks`, теперь он должен передавать объект `List<Task>` из свойства адаптеру `TaskItemAdapter`.

Список задач
Сетка карточек

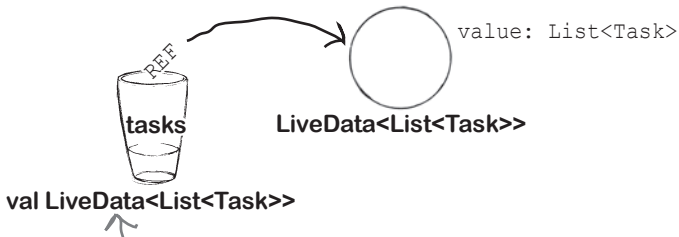
Фрагмент TasksFragment должен обновлять свойство данных TaskItemAdapter

Как вы уже знаете, свойство `tasks` объекта `TasksViewModel` содержит список задач в виде данных Live Data, которые читаются из базы данных с помощью кода:

```
val tasks = dao.getAll()
```

Здесь метод getAll() объекта TaskDao используется для получения LiveData<List<Task>> из базы данных.

Так как это свойство использует механизм Live Data, можно отдать команду `TasksFragment` наблюдать за ним, чтобы при каждом изменении его значения фрагмент получал оповещение. Тогда `TasksFragment` может присвоить новейшую версию списка свойству `data` адаптера, гарантируя, что отображаемые в представлении с переработкой данные всегда остаются актуальными.



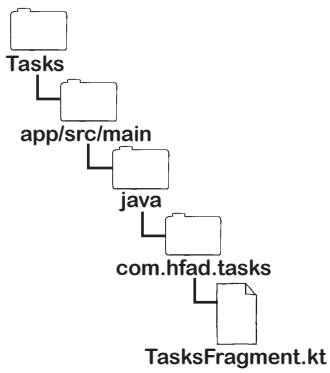
Свойство tasks содержит список задач Live Data, который загружается из базы данных.

Код наблюдения за свойствами Live Data вам уже знаком, поэтому мы просто приведем код, который необходимо добавить в `TasksFragment` (выделен жирным шрифтом):

```
class TasksFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        ...

        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.data = it
            }
        })
        ...
    }
    ...
}
```

Наблюдаем за свойством tasks модели представления.

Если список задач изменился, он присваивается свойству data адаптера.



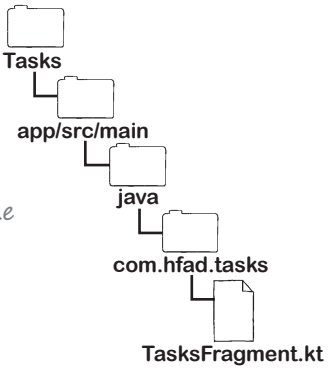
Обновим файл *TasksFragment.kt* и включим в него это изменение.

Полный код TasksFragment.kt

Ниже приведена новая версия `TasksFragment`; обновите файл *TasksFragment.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter
```

Импортируйте этот класс.



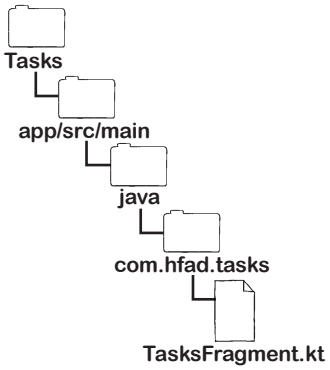
Продолжение на следующей странице.

Список задач
Сетка карточек

TasksFragment.kt (продолжение)

Данные передаются адаптеру.

```
        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.data = it
            }
        })

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```



Работа над кодом представления с переработкой завершена.

Это заняло некоторое время, но мы написали весь код, необходимый для отображения имен задач в представлении с переработкой. Основные этапы этой работы:

1. **Создание адаптера с именем TaskItemAdapter.**
   Адаптер соединяет представление с переработкой с его источником данных. В приложении Tasks источником данных является база данных Room, содержащая записи с задачами.

2. **Присоединение TaskItemAdapter в представлении с переработкой.**
   Мы добавили представление с переработкой в макет `TasksFragment` и отдали ему команду использовать `TaskItemAdapter` в коде Kotlin.

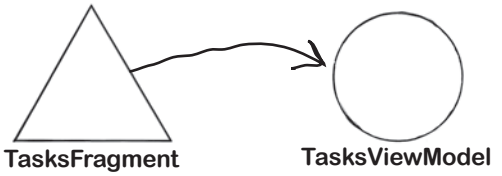
3. **Передача актуального списка List<Task> адаптеру TaskItemAdapter.**
   Для этого `TasksFragment` задает значение свойства data объекта `TaskItemAdapter` при каждом обновлении списка задач Live Data из объекта `TasksViewModel`.

Прежде чем мы проведем тест-драйв приложения и посмотрим, как выглядит представление с переработкой, разберемся, что же происходит при выполнении кода.

Что происходит при выполнении кода

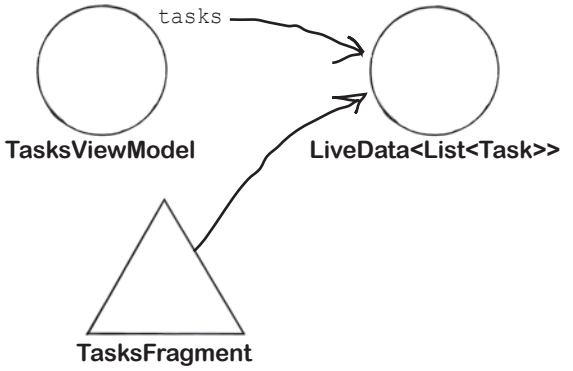
При выполнении кода происходят следующие события:

1. **При запуске приложения MainActivity отображает TasksFragment.**
`TasksFragment` использует `TasksViewModel` как свою модель представления.

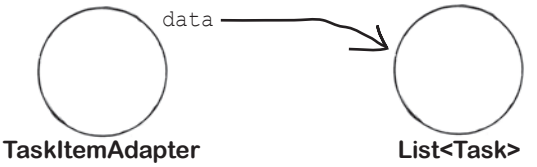


2. **TasksFragment создает объект TaskItemAdapter и присваивает его представлению с переработкой в качестве адаптера.**

3. **TasksFragment наблюдает за свойством tasks объекта TasksViewModel.**
Это свойство имеет тип `LiveData<List<Task>>` и содержит актуальный список записей из базы данных.



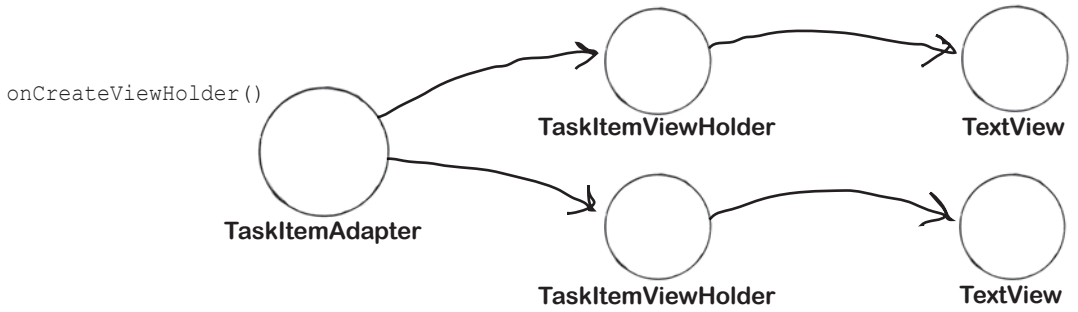
4. **TasksFragment присваивает значение свойства data объекта TaskItemAdapter списку List<Task>.**



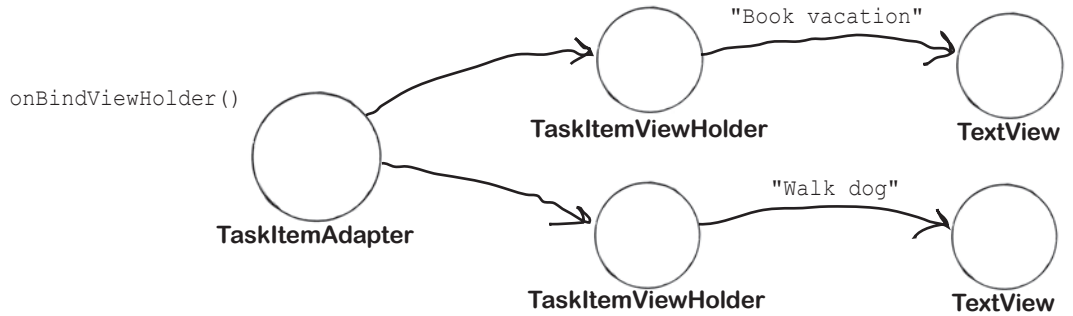
Список задач
Сетка карточек

История продолжается

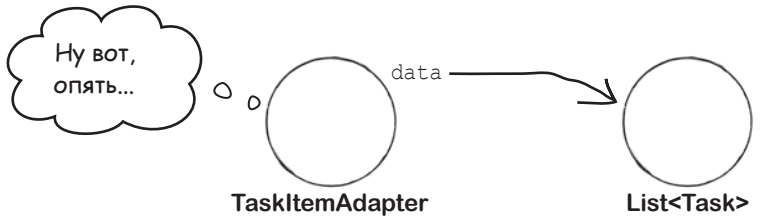
5 **Метод onCreateViewHolder() объекта TaskItemAdapter вызывается для каждого элемента, который должен отображаться в представлении с переработкой.**
При этом для каждого элемента создается объект `TaskItemViewHolder` (держатель представления). Для каждого держателя представления заполняется макет (определяемый файлом *task_item.xml*).



6 **Метод onBindViewHolder() объекта TaskItemAdapter вызывается для каждого объекта TaskItemViewHolder.**
Данные связываются с текстовым представлением в макете каждого держателя представления.



7 **При каждом обновлении свойства tasks объекта TasksViewModel фрагмент TasksFragment передает обновленный список List<Task> объекту TaskItemAdapter.**
Этапы 5 и 6 повторяются, чтобы представление с переработкой оставалось актуальным.

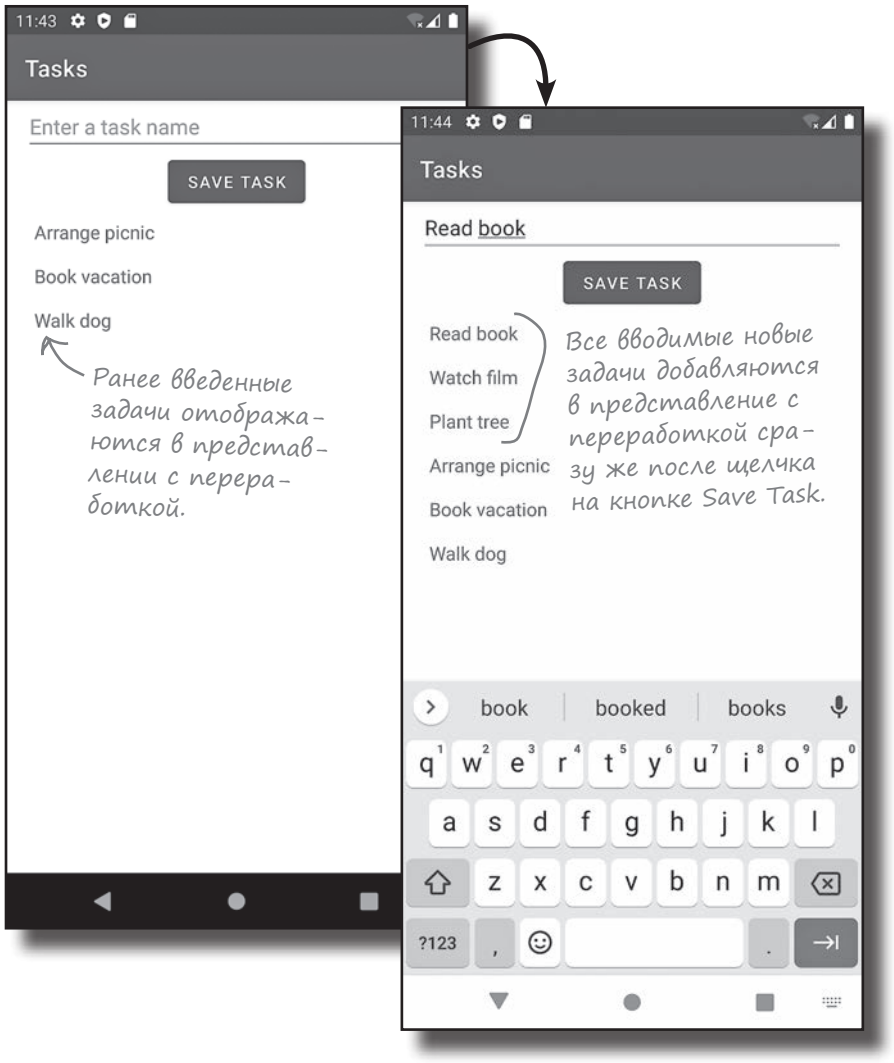


Давайте проведем тест-драйв приложения.

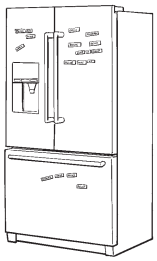
Тест-драйв

При запуске приложения `TasksFragment` отображает имена всех задач в представлении с переработкой.

Если ввести новую задачу, она будет добавлена в список представления с переработкой. Приложение работает так, как было задумано.



Вы узнали, как создать простейшее представление с прокруткой. Прежде чем настраивать его, чтобы изменить способ отображения записей, опробуйте свои силы на следующем упражнении.



Развлечения с магнитами

Приложение Bits and Pizzas содержит класс данных `Pizza`, который выглядит так:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

Приложение должно включать представление с переработкой, в котором для каждого элемента `Pizza` отображается значение `pizzaName` в следующем макете (с именем *pizza_item.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

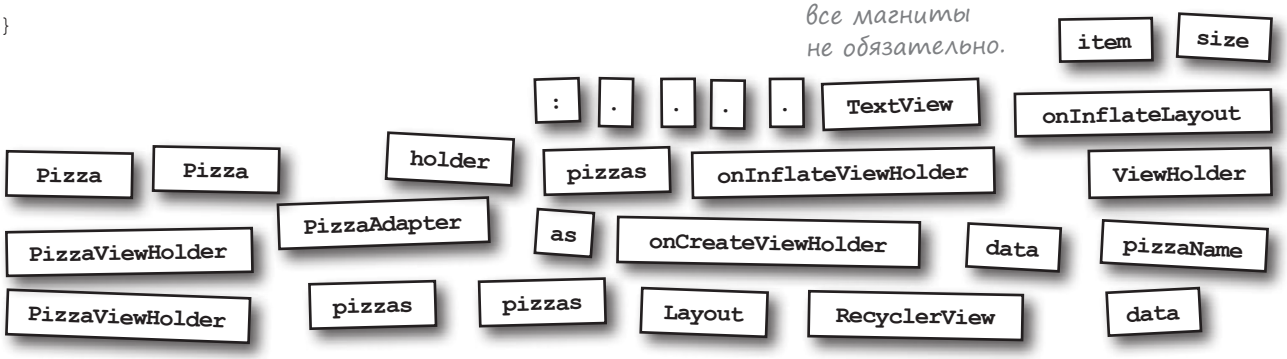
Удастся ли вам завершить код адаптера представления с переработкой (см. ниже)?

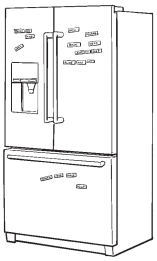
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class PizzaAdapter : RecyclerView.Adapter<                                    >() {

    var pizzas = listOf<          >()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() =                                              
```

Продолжение на следующей странице.

```
    override fun _____(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(_____, position: Int) {
        val item = _____ [position]
        holder.bind(item)
    }

    class PizzaViewHolder(val rootView: TextView)
                            : _____(rootView) {
        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                            .inflate(R.layout.pizza_item, parent, false)  _____
                return PizzaViewHolder(view)
            }
        }

        fun bind(item:_____) {
            rootView.text = _____
        }
    }
}
```

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Приложение Bits and Pizzas содержит класс данных `Pizza`, который выглядит так:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

Приложение должно включать представление с переработкой, в котором для каждого элемента `Pizza` отображается значение `pizzaName` в следующем макете (с именем *pizza_item.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Удастся ли вам завершить код адаптера представления с переработкой (см. ниже)?

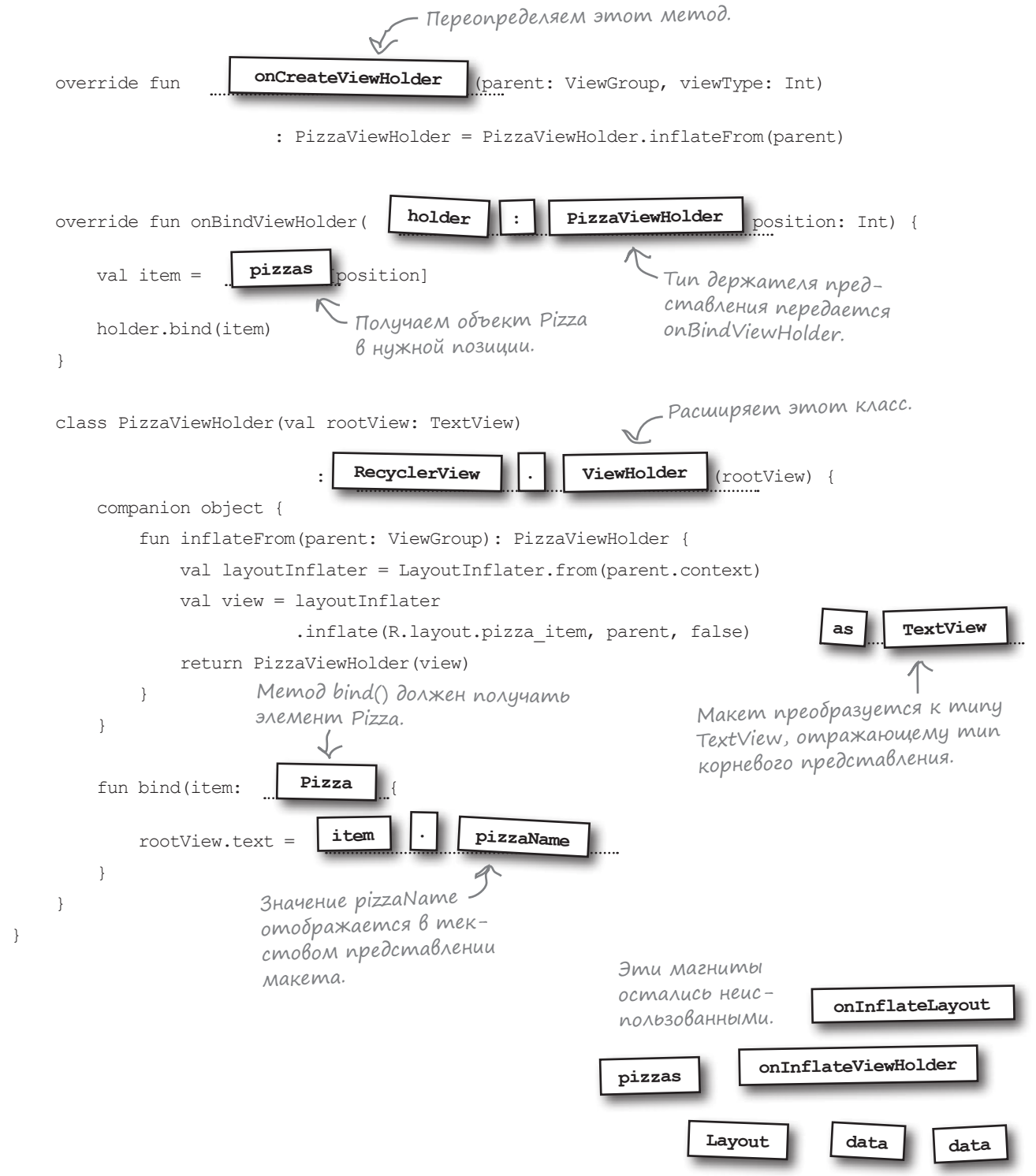
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class PizzaAdapter : RecyclerView.Adapter<  PizzaAdapter . PizzaViewHolder  {

    var pizzas = listOf<  Pizza  >()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() =  pizzas . size
```

Адаптер должен использовать внутренний класс PizzaViewHolder в качестве держателя представления.

Адаптер использует список объектов Pizza.

Определяет количество элементов.

Продолжение на следующей странице.


Переопределяем этот метод.
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
: PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)
override fun onBindViewHolder(holder : PizzaViewHolder position: Int) {
val item = pizzas[position]
holder.bind(item)
}
Получаем объект Pizza в нужной позиции.
Тип держателя представления передается onBindViewHolder.
class PizzaViewHolder(val rootView: TextView)
Расширяет этот класс.
: RecyclerView . ViewHolder(rootView) {
companion object {
fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
val layoutInflater = LayoutInflater.from(parent.context)
val view = layoutInflater
.inflate(R.layout.pizza_item, parent, false)
return PizzaViewHolder(view)
}
}
as TextView
Макет преобразуется к типу TextView, отражающему тип корневого представления.
Метод bind() должен получать элемент Pizza.
fun bind(item: Pizza) {
rootView.text = item . pizzaName
}
}
}
Значение pizzaName отображается в текстовом представлении макета.
Эти магниты остались неиспользованными.
onInflateLayout
pizzas
onInflateViewHolder
Layout
data
data

Список задач
Сетка карточек

Представления с переработкой чрезвычайно гибки

К настоящему моменту в этой главе вы узнали, как построить простое представление с переработкой, которое отображает список имен задач. Для этого мы создали макет, который используется каждым элементом списка, определили адаптер для заполнения его данными и добавили представление с переработкой в макет `TaskFragment`.

А я хочу знать, почему они сделали представления с переработкой **такими сложными**! Почему я должна создавать **дополнительный макет** и **адаптер**? Разве не **проще** было бы написать небольшой код фрагмента?

Представления с переработкой могут показаться сложными, но они чрезвычайно гибки.

В этом приложении мы создадим представление с переработкой для вывода простого списка имен задач, но это только начало. Представления с переработкой также могут использоваться для других целей, в том числе:

- **Вывода списков изображений с включением графических представлений в макет элемента.**
- **Использования других менеджеров макетов для отображения элементов в виде сетки (вместо вертикального списка).**
- **Программирования реакции на щелчки, чтобы представление могло использоваться для навигации.**

Чтобы показать, насколько гибкими могут быть представления с переработкой, мы изменим только что созданное представление с переработкой, чтобы в нем выводилась более подробная информация о каждой задаче.

Посмотрим, как будет выглядеть новая версия представления с переработкой.

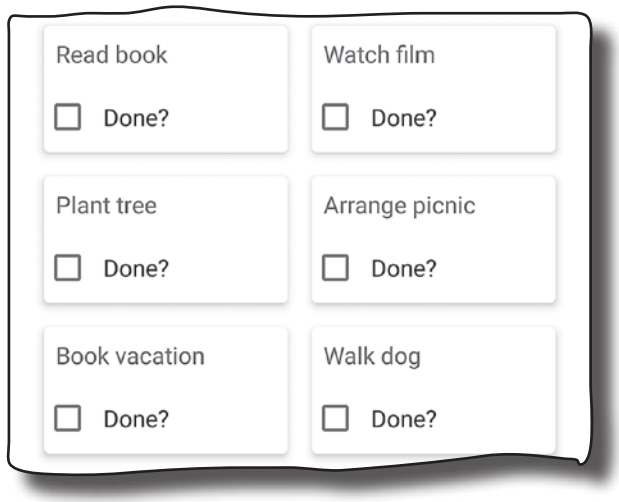
Представление с переработкой 2.0

Список задач
Сетка карточек

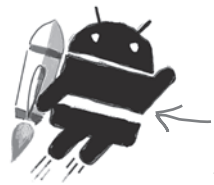
Мы обновим представление с переработкой, чтобы имя каждой задачи отображалось в текстовом представлении, рядом с которым находится флажок — признак ее завершения. Данные записи каждой задачи отображаются на отдельной карточке, которая кажется слегка приподнятой. Карточки образуют сетку.

Новая версия представления с переработкой должна выглядеть так:

Элементы с описаниями задач должны отображаться в карточках, образующих сетку. Каждая карточка содержит текстовое представление для имени задачи, а также флажок — признак завершения.



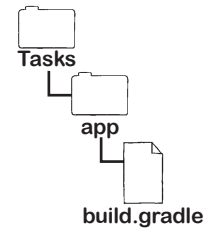
Чтобы создать эту версию представления с переработкой, мы изменим файл *task_item.xml* (макет, используемый элементами представления с переработкой), чтобы в нем использовалось **карточное представление**. Это разновидность фреймового макета с закругленными углами и тенями, из-за которых он кажется слегка приподнятым над фоном.



Карточные представления являются частью Android Jetpack.

Добавление зависимости для карточного представления в файл build.gradle приложения

Чтобы использовать карточное представление, прежде всего необходимо добавить зависимость для его библиотеки в файл *build.gradle* приложения. Откройте файл *Tasks/app/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `dependencies`:




```
...
dependencies {
    ...
    implementation 'androidx.cardview:cardview:1.0.0'
    ...
}
```

Добавляет библиотеку карточных представлений в приложение.

Не забудьте синхронизировать это изменение с остальными частями приложения.

Список задач
Сетка карточек

Создание карточного представления

Мы используем карточное представление из файла *task_item.xml*, которое включает текстовое представление и флажок.

Чтобы создать карточное представление, добавьте элемент `<androidx.cardview.widget.CardView>` в код макета. Код типичного карточного представления выглядит так:

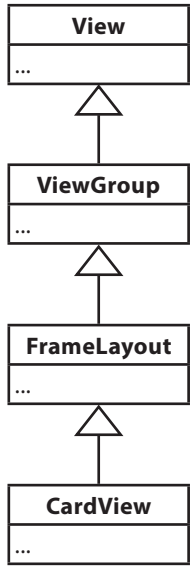
```
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardElevation="4dp"
    app:cardCornerRadius="4dp" >

    ...

</androidx.cardview.widget.CardView>
```

Определяет карточное представление (CardView).

Это свойство создает эффект отбрасываемой тени.

Создает закругленные углы у CardView.

Здесь добавляются все представления, которые должны отображаться в CardView.



CardView является разновидностью FrameLayout. Класс включает все атрибуты FrameLayout, а также содержит ряд дополнительных атрибутов.

Как видно из этого кода, в него включается дополнительное пространство имен:

```
xmlns:app="http://schemas.android.com/apk/res-auto"
```

Это позволяет добавить атрибуты, с которыми карточка имеет закругленные углы и эффект тени, из-за которого она кажется приподнятой над окружающим фоном. Закругленные углы создаются атрибутом `app:cardCornerRadius`, а атрибут `app:cardElevation` создает эффект объема и добавляет отбрасываемые тени:

Атрибут app:cardBackgroundColor изменяет цвет фона карточки.

Пустое карточное представление с закругленными углами и эффектом объема.

После того как вы определите карточное представление, в него можно добавить любые представления, которые в нем должны присутствовать. В приложении Tasks в карточное представление следует добавить текстовое представление для имени задачи и флажок для признака ее завершения. Давайте посмотрим, как это будет выглядеть в коде.

Полный код task_item.xml

Ниже приведена новая версия *task_item.xml*; обновите содержимое файла (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardElevation="4dp"
    app:cardCornerRadius="4dp" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >

        <TextView
            xmlns:android="http://schemas.android.com/apk/res/android"
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:padding="8dp" />

        <CheckBox
            android:id="@+id/task_done"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:padding="8dp"
            android:clickable="false"
            android:text="Done?" />
    </LinearLayout>
</androidx.cardview.widget.CardView>
```

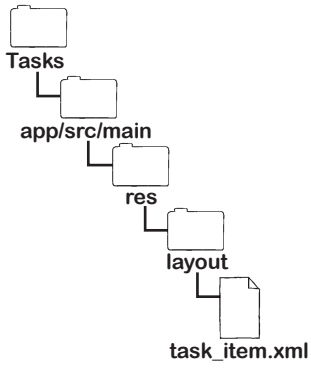
Добавляем карточное представление.

В карточное представление добавляется текстовое представление и флажок.

Эта строка перемещается в элемент карточного представления.

Текстовому представлению назначается идентификатор, чтобы к нему можно было обращаться из кода Kotlin приложения.

Добавляем флажок.



Затем необходимо обновить держатель представления адаптера, чтобы он работал с новым макетом, и заполнить представления карточки.

Список задач
Сетка карточек

Держатель представления адаптера должен работать с новым кодом макета

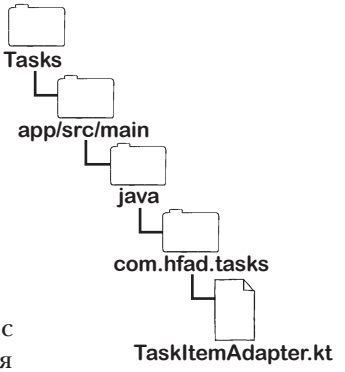
При определении `TaskItemAdapter` (адаптера представления с переработкой) был включен внутренний класс `TaskItemViewHolder`. Мы использовали его для заполнения макета (текстового представления), который ассоциировался с каждым элементом представления с переработкой, и для заполнения имени задачи.

Напомним, как выглядел код исходного внутреннего класса:

```
class TaskItemViewHolder(val rootView: TextView)
                        : RecyclerView.ViewHolder(rootView) {
    companion object {
        fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater
                       .inflate(R.layout.task_item, parent, false) as TextView
            return TaskItemViewHolder(view)
        }
    }

    fun bind(item: Task) {
        rootView.text = item.taskName
    }
}
```

Заполняет макет элемента и использует его для создания держателя представления.

Добавляет данные в представления элемента.



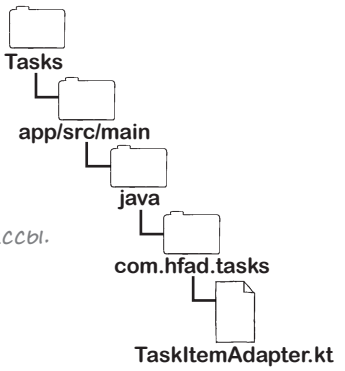
После изменения *task_item.xml* следует обновить класс `TaskItemViewHolder`, чтобы он работал с новым макетом. Для этого необходимо внести три изменения:

1. **Обновить конструктор держателя представления, чтобы он использовал CardView вместо TextView.**
2. **Изменить метод inflateFrom(), чтобы он заполнял макет каждого элемента с CardView.**
3. **Обновить метод bind(), чтобы он заполнял текстовое представление и флажок в макете значениями свойств taskName и taskDone.**

Код, необходимый для внесения этих изменений, вам уже знаком, поэтому мы просто приведем обновленный код `TaskItemAdapter` — и внутреннего класса `TaskItemViewHolder` — на следующей странице.

Полный код TaskItemAdapter.kt

Ниже приведен обновленный код `TaskItemAdapter`, который работает с новым кодом макета; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView

class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item)
    }
```

Импортируйте эти классы.

Продолжение на следующей странице.

Список задач
Сетка карточек

TaskItemAdapter.kt (продолжение)

```
class TaskItemViewHolder(val rootView: ~~TextView~~ CardView)
                                     : RecyclerView.ViewHolder(rootView) {
    val taskName = rootView.findViewById<TextView>(R.id.task_name)
    val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)

    companion object {
        fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater
                       .inflate(R.layout.task_item, parent, false) as ~~TextView~~ CardView
            return TaskItemViewHolder(view)
        }
    }

    fun bind(item: Task) {
        ~~rootView~~ taskName.text = item.taskName
        taskDone.isChecked = item.taskDone
    }
}
}
```

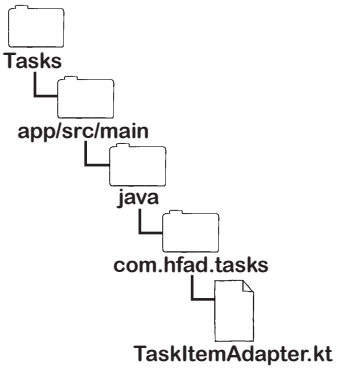
TextView заменяется на CardView.

Получаем ссылки на текстовое представление и флажок из макета.

Здесь тоже необходимо заменить TextView на CardView.

Имя задачи выводится в текстовом представлении.

Признак завершения задачи представляется флажком.



Часто Задаваемые Вопросы

В: Можно ли выводить графику в представлениях с переработкой?

О: Да! Для этого включите графическое представление в макет элемента и заполните его в методе `bind()` держателя представления.

В: Представления с переработкой работают только с информацией из базы данных?

О: Нет, они также могут использоваться с данными из других источников — при условии, что адаптер имеет доступ к данным.

В: Вижу, в коде держателя представления вызывается метод `findViewById()`. Но ведь мы отказались от его использования?

О: В коде этот метод используется для получения ссылки на каждое представление в макете элемента, который должен быть заполнен данными. Мы воспользовались им как относительно простым способом добавления данных в каждое представление. В следующей главе вы узнаете, как заменить эти вызовы связыванием данных.

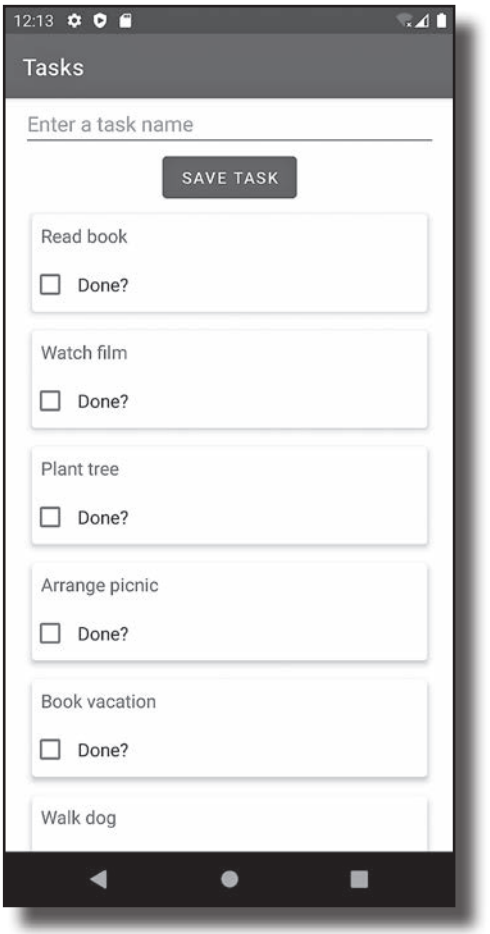
В: Могут ли представления с переработкой использовать разные макеты?

О: Да! Например, можно сделать так, чтобы первый элемент списка использовал собственный макет — не тот, который используется большинством элементов. Для этого следует создать все макеты, которые вы собираетесь использовать, и определить держатель представления для каждого макета. Затем переопределите метод `getItemViewType()` адаптера, чтобы он сообщал, какой держатель представления должен использоваться для той или иной позиции представления с переработкой. Наконец, обновите метод `onCreateViewHolder()`, чтобы он проверял параметр типа представления и создавал соответствующий держатель представления.

Как представление с переработкой выглядит сейчас

Если запустить приложение после обновления кода *task_item.xml* и *TaskItemAdapter.kt*, вы увидите представление с переработкой, которое выводит данные задач в вертикальном списке карточных представлений:

С внесенными изменениями задачи выводятся в вертикальном списке карточек.



Представление с переработкой размещает карточки именно так, потому что в файле *fragment_tasks.xml* указано, что в нем должен использоваться менеджер линейного макета:

```
<androidx.recyclerview.widget.RecyclerView
    ...
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
```

В текущей версии приложения используется менеджер линейного макета для вывода элементов в вертикальном списке.

По умолчанию этот менеджер макета размещает элементы в вертикальном списке с записями, распространяющимися на всю ширину. Однако вы можете включить дополнительные настройки — или выбрать другую разновидность менеджера макета, — чтобы изменить способ отображения элементов.

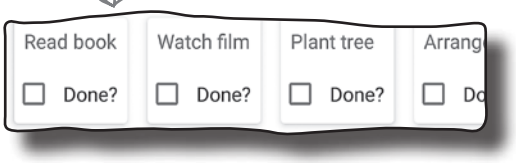
Рассмотрим некоторые возможные варианты.

Список задач
Сетка карточек

Галерея менеджеров макетов

Ниже описаны некоторые альтернативные возможности размещения элементов в представлениях с переработкой, а также способы их создания.

Используется менеджер линейного макета с горизонтальной ориентацией.



Размещение элементов по горизонтали

Менеджер линейного макета по умолчанию отображает элементы в вертикальном списке. Однако элементы также можно разместить по горизонтали, для этого достаточно переключиться на горизонтальную ориентацию:

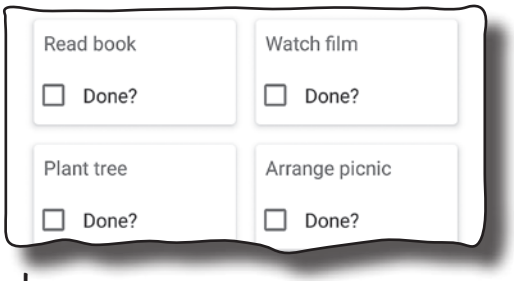
```
<androidx.recyclerview.widget.RecyclerView ...
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
    android:orientation="horizontal" />
```

Использование GridLayoutManager для размещения элементов в сетке

Если вы хотите выстроить элементы в сетку, попробуйте воспользоваться классом `GridLayoutManager`. Атрибут `app:spanCount` определяет количество столбцов в сетке:

```
<androidx.recyclerview.widget.RecyclerView ...
    app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
    app:spanCount="2" />
```

Используется менеджер сеточного макета с двумя столбцами.

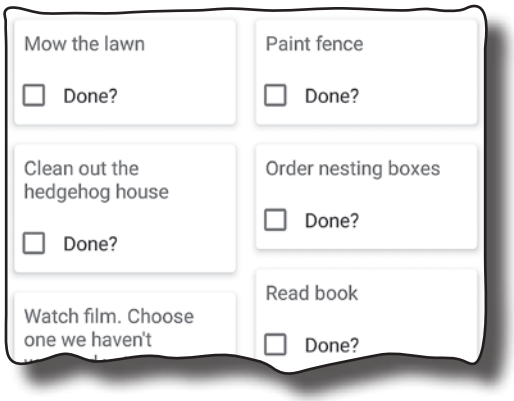


Размещение элементов в неравномерной сетке

Если ваши элементы имеют разные размеры, можно воспользоваться классом `StaggeredGridLayoutManager`:

```
<androidx.recyclerview.widget.RecyclerView ...
    app:layoutManager="androidx.recyclerview.widget.StaggeredGridLayoutManager"
    app:spanCount="2" />
```

Менеджер неравномерного сетчатого макета хорошо подходит для элементов с разными размерами.



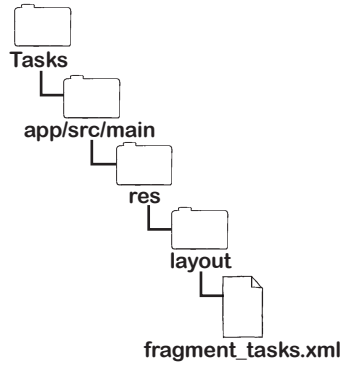
Применим один из этих стилей к представлению с переработкой в приложении Tasks и посмотрим, что происходит при запуске приложения.

Список задач
Сетка карточек

Обновление fragment_tasks.xml для размещения элементов в сетке

Мы обновим представление с переработкой, чтобы элементы размещались в сетке из двух столбцов. Ниже приведена новая версия кода макета; обновите файл *fragment_tasks.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".TasksFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.TasksViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <EditText
            ... />

        <Button
            ... />

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/tasks_list"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:gravity="top"
            app:layoutManager="androidx.recyclerview.widget.~~Linear~~Grid LayoutManager"
            app:spanCount="2" />
    </LinearLayout>
</layout>
```



Используется менеджер сетчатого макета вместо линейного.

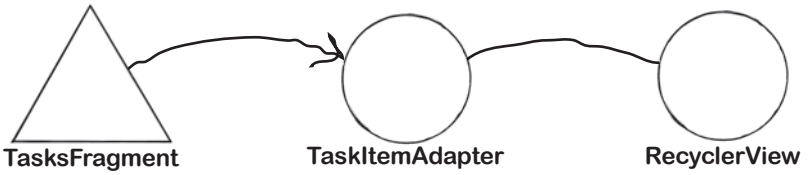
Сетка состоит из двух столбцов.

Список задач
Сетка карточек

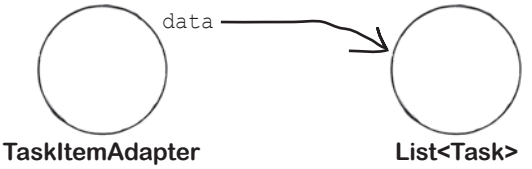
Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

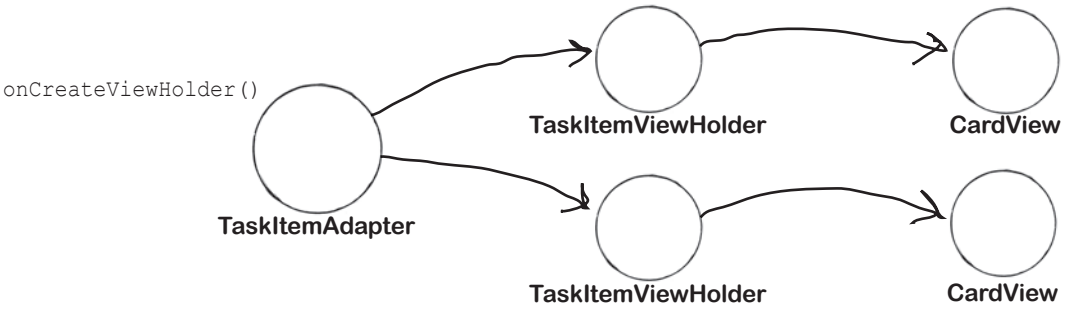
1. **TasksFragment создает объект TaskItemAdapter и назначает его адаптером представления с переработкой.**



2. **TasksFragment присваивает свойству data объекта TaskItemAdapter список List<Task>.**
`TasksFragment` получает `List<Task>`, наблюдая за свойством `tasks` объекта `TasksViewModel`.

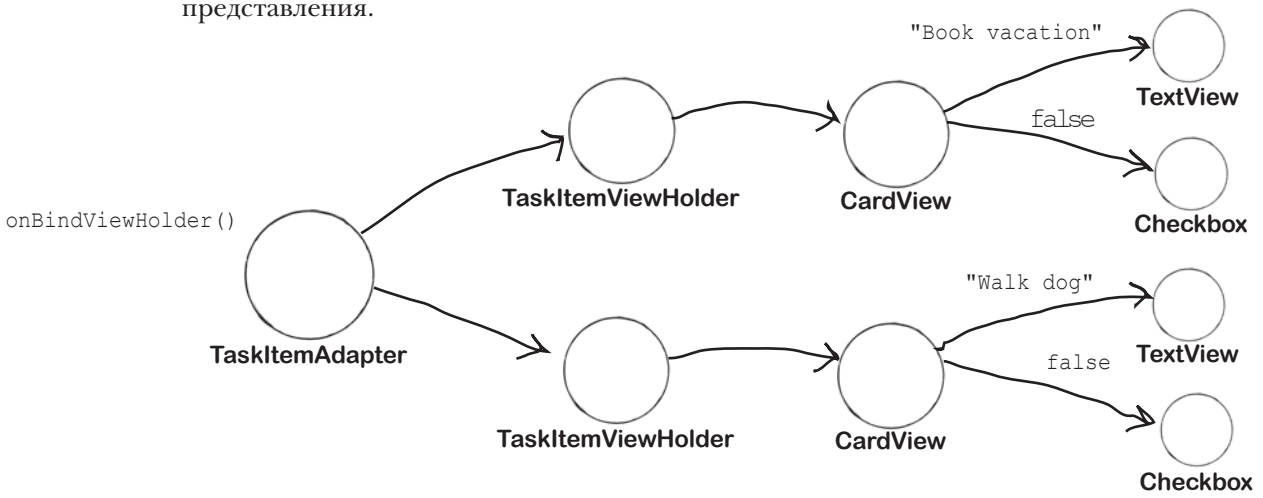


3. **Метод onCreateViewHolder() объекта TaskItemAdapter вызывается для каждого элемента, который должен отображаться в представлении с переработкой.**
В результате для каждого элемента создается объект `TaskItemViewHolder`. Макет (определенный в файле *task_item.xml*) заполняется для каждого держателя представления.

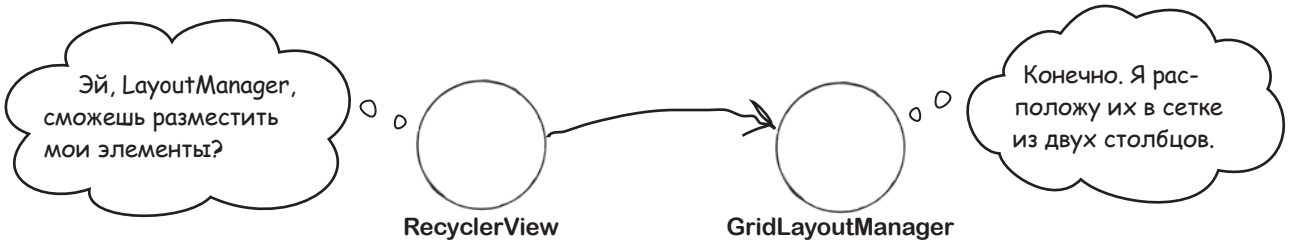


История продолжается

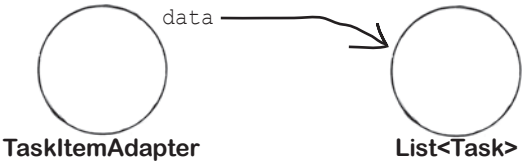
4 **Метод onBindViewHolder() объекта TaskItemAdapter вызывается для каждого TaskItemViewHolder.**
Вызов связывает данные с представлениями в макете каждого держателя представления.



5 **Представление с переработкой использует менеджер макета для размещения своих элементов.**
Так как представление с переработкой использует `GridLayoutManager` со значением `spanCount`, равным 2, элементы выстраиваются в сетку с двумя столбцами.



6 **При каждом обновлении свойства tasks объекта TasksViewModel, TasksFragment передает обновленный список List<Task> объекту TaskItemAdapter.**
Этапы с 3-го по 5-й повторяются, чтобы представление с переработкой оставалось актуальным.



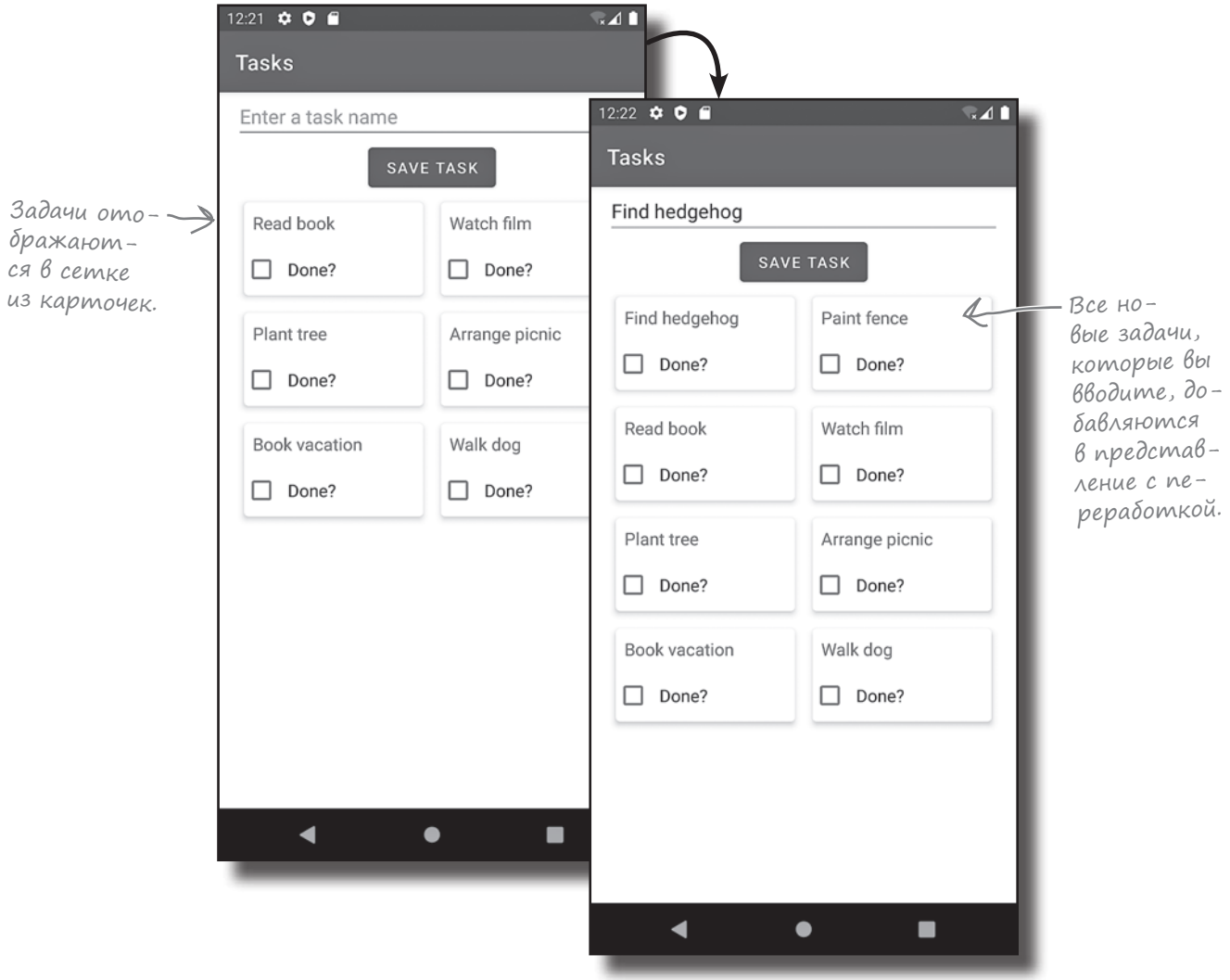
Проведем тест-драйв приложения.

Тест-драйв

Список задач
Сетка карточек

При запуске приложения представление с переработкой фрагмента `TasksFragment` отображает сетку карточек; в каждой карточке выводится имя задачи и признак ее завершения.

Когда вы вводите новую задачу, она появляется в представлении с переработкой сразу же после того, как вы щелкнете на кнопке Save Task.



Поздравляем! Вы научились управлять внешним видом представлений с переработкой при помощи менеджеров макетов, а также отображать данные в сетке карт с поддержкой прокрутки.

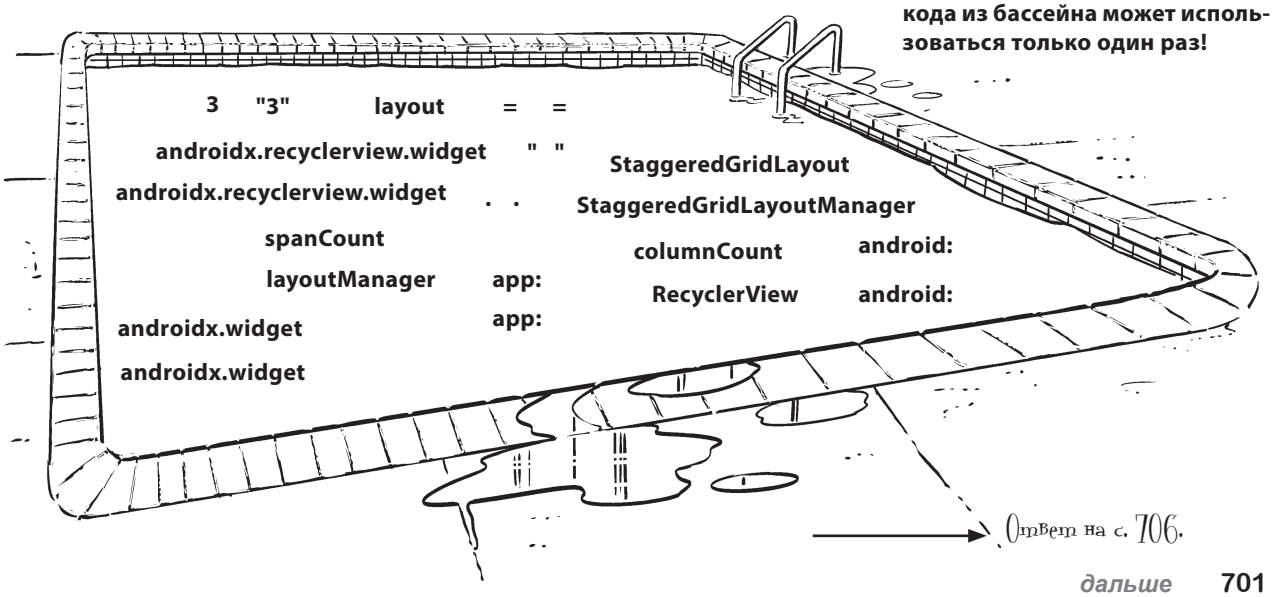
В следующей главе мы расширим новые знания и внесем дальнейшие улучшения в представление с переработкой.

У бассейна

Ваша **задача** — обновить приведенный ниже код макета, чтобы в нем отображалось представление с переработкой, которое выводит свои элементы в неравномерной сетке из трех столбцов. Выловите из бассейна сегменты кода и расставьте их в пустых строках в коде. Каждый фрагмент кода может использоваться только один раз, при этом все фрагменты использовать не обязательно.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <.....
        android:id="@+id/recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        ..........
        ...../>
</LinearLayout>
```

Внимание: каждый сегмент кода из бассейна может использоваться только один раз!



Ответ на с. 706.

Упражнение

Приложение Bits and Pizzas включает класс данных `Pizza`, который выглядит так:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

Приложение должно включать представление с переработкой, которое отображает свойства `pizzaName` и `PizzaDescription` для каждого элемента `Pizza` в следующем линейном макете (с именем *pizza_item.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="8dp" >

    <TextView
        android:id="@+id/pizza_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/pizza_description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
 </LinearLayout>
```

Удастся ли вам обновить код адаптера на следующей странице, чтобы он отображал правильные свойства в приведенном выше макете?

```
...
class PizzaAdapter : RecyclerView.Adapter<PizzaAdapter.PizzaViewHolder>() {
    var pizzas = listOf<Pizza>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = pizzas.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = pizzas[position]
        holder.bind(item)
    }

    class PizzaViewHolder(val rootView: TextView)
                                        : RecyclerView.ViewHolder(rootView) {

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater.inflate(R.layout.pizza_item, parent, false)
                            as TextView
                return PizzaViewHolder(view)
            }
        }

        fun bind(item: Pizza) {
            rootView.text = item.pizzaName
        }
    }
}
```

Это код адаптера, обновленный в предыдущем упражнении. Сможете ли вы заставить его работать с новой версией pizza_item.xml?

Упражнение Решение

Приложение Bits and Pizzas включает класс данных `Pizza`, который выглядит так:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

Приложение должно включать представление с переработкой, которое отображает свойства `pizzaName` и `PizzaDescription` для каждого элемента `Pizza` в следующем линейном макете (с именем *pizza_item.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="8dp" >

    <TextView
        android:id="@+id/pizza_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/pizza_description"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
 </LinearLayout>
```

Удастся ли вам обновить код адаптера на следующей странице, чтобы он отображал правильные свойства в приведенном выше макете?

```
...
class PizzaAdapter : RecyclerView.Adapter<PizzaAdapter.PizzaViewHolder>() {
    var pizzas = listOf<Pizza>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = pizzas.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = pizzas[position]
        holder.bind(item)
    }

    class PizzaViewHolder(val rootView: ~~TextView~~ LinearLayout)
                                        : RecyclerView.ViewHolder(rootView) {
        val pizzaName = rootView.findViewById<TextView>(R.id.pizza_name)
        val pizzaDescription = rootView.findViewById<TextView>(R.id.pizza_description)

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater.inflate(R.layout.pizza_item, parent, false)
                           as ~~TextView~~ LinearLayout
                return PizzaViewHolder(view)
            }
        }

        fun bind(item: Pizza) {
            ~~rootView~~ pizzaName.text = item.pizzaName
            pizzaDescription.text = item.pizzaDescription
        }
    }
}
```

Если ваш код в чем-то отличается от нашего, не беспокойтесь. Это всего лишь один из возможных ответов.

Корневым представлением макета является LinearLayout.

Получаем ссылки на представления макета.

Представление преобразуется к типу LinearLayout в соответствии с корневым представлением макета.

Данные включаются в представления макета.

У бассейна. Решение

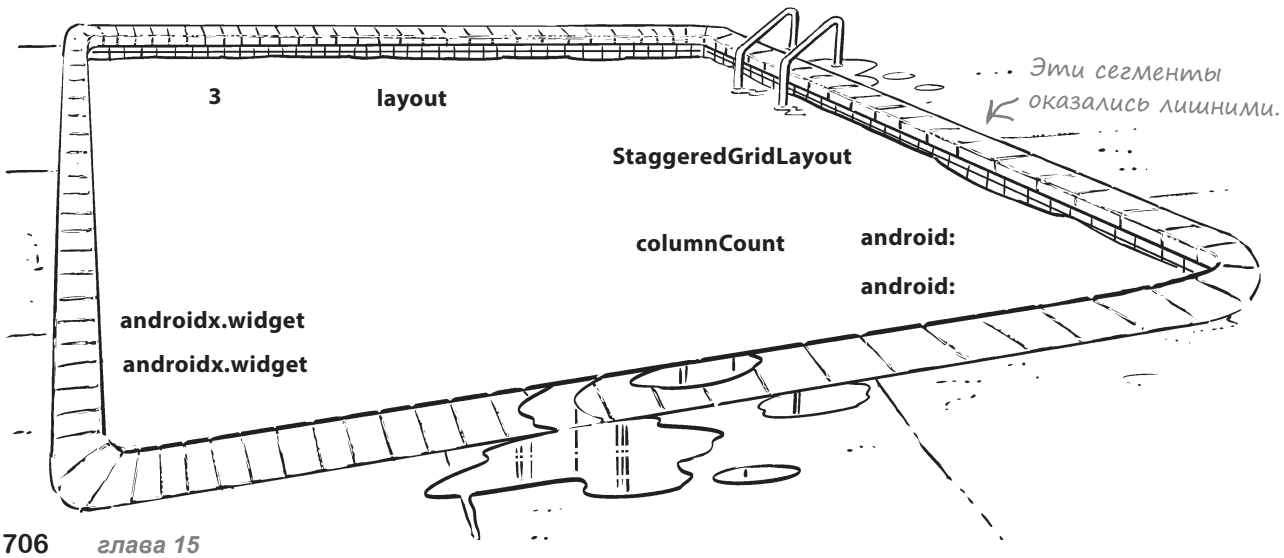
Ваша **задача** — обновить приведенный ниже код макета, чтобы в нем отображалось представление с переработкой, которое выводит свои элементы в неравномерной сетке из трех столбцов. Выловите из бассейна сегменты кода и расставьте их в пустых строках в коде. Каждый фрагмент кода может использоваться только один раз, при этом все фрагменты использовать не обязательно.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    < androidx.recyclerview.widget . RecyclerView
        android:id="@+id/recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app: layoutManager = " androidx.recyclerview.widget . StaggeredGridLayoutManager "
        app: spanCount = "3" />
</LinearLayout>
```

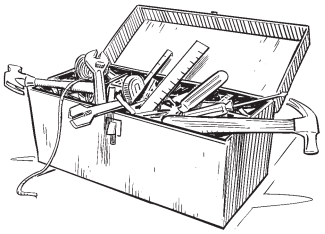
Этот элемент используется для добавления представления с переработкой.

Эта разновидность менеджера макета используется для отображения элементов в неравномерной сетке.

Неравномерная сетка должна состоять из трех столбцов.



Ваш инструментарий Android



Глава 15 осталась позади, а ваш инструментарий пополнился представлениями с переработкой.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Представление с переработкой — гибкое средство для вывода списка данных с возможностью прокрутки.
- Зависимость для библиотеки представлений с переработкой добавляется в файл *build.gradle* приложения.
- Для каждого представления с переработкой вы определяете макет для его элементов и адаптер, который помещает данные в представления каждого элемента. Затем представление с переработкой отображает эти элементы.
- Адаптер должен расширять класс `RecyclerView.Adapter`.
- Адаптер использует держатель представления, который расширяет `RecyclerView.ViewHolder`. Он содержит информацию о том, как должен отображаться макет каждого элемента, и обычно определяется как внутренний класс адаптера.
- Метод `onCreateViewHolder()` адаптера заполняет макет каждого элемента и создает держателей представлений.
- Метод `onBindViewHolder()` адаптера связывает данные с представлениями в макете каждого элемента.
- Для добавления представления с переработкой используется элемент `<androidx.recyclerview.widget.RecyclerView>`.
- Карточное представление — разновидность фреймового макета с закругленными углами и имитацией рельефа.
- Зависимость для библиотеки карточных представлений добавляется в файл *build.gradle* приложения.
- Менеджер макета управляет отображением элементов.
- `LinearLayoutManager` размещает элементы по вертикали или по горизонтали.
- `GridLayoutManager` размещает элементы в сетке.
- `StaggeredGridLayoutManager` напоминает `GridLayoutManager`, но поддерживает элементы с различающимися размерами.

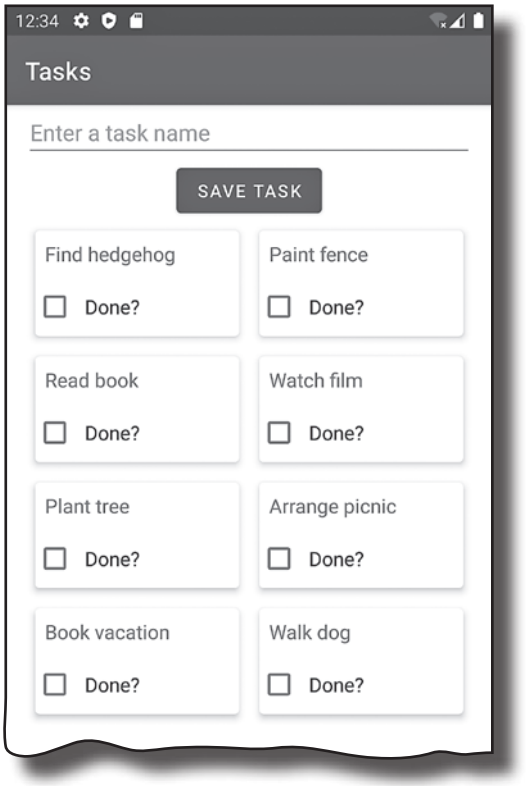
На полных оборотах

Приложение должно быть настолько плавным и быстрым, насколько это возможно. Но если действовать неосторожно, большие и сложные наборы данных могут привести к сбоям в вашем представлении с переработкой. В этой главе мы представим ***DiffUtil***: вспомогательный класс, который **расширяет возможности представлений с переработкой**. Вы научитесь использовать его для **эффективного обновления** представлений с переработкой. Вы узнаете, как объекты ***ListAdapter*** упрощают работу с ***DiffUtil***. А заодно будет показано, **как полностью избавиться от *findViewById()*** реализацией **связывания данных в коде представления с переработкой.**

Представление с переработкой правильно отображает данные задач...

В предыдущей главе мы добавили в приложение Tasks представление с переработкой, которое отображает свои данные точно так, как нам требуется. Каждая задача отображается на отдельной карточке, а на каждой карточке выводится имя задачи и признак ее завершения. Карточки выстраиваются в сетку из двух столбцов:

Представление с переработкой, добавленное в TasksFragment. Записи задач отображаются в сетке из двух столбцов.



...но при обновлении данных в представлении с переработкой происходит переход

Каждый раз, когда мы добавляем новую задачу, представление с переработкой перерисовывается, чтобы оно включало новую запись и оставалось актуальным. Однако при этом в представлении с переработкой происходит резкий переход.

Каждый раз, когда возникает необходимость в обновлении представления с переработкой, **перерисовывается весь список**. Нет плавных переходов, которые бы указывали на изменения, и если список окажется очень длинным, пользователь перестает понимать, в какой позиции он сейчас находится. Кроме того, такой подход неэффективен для больших наборов данных и может привести к проблемам с быстродействием.

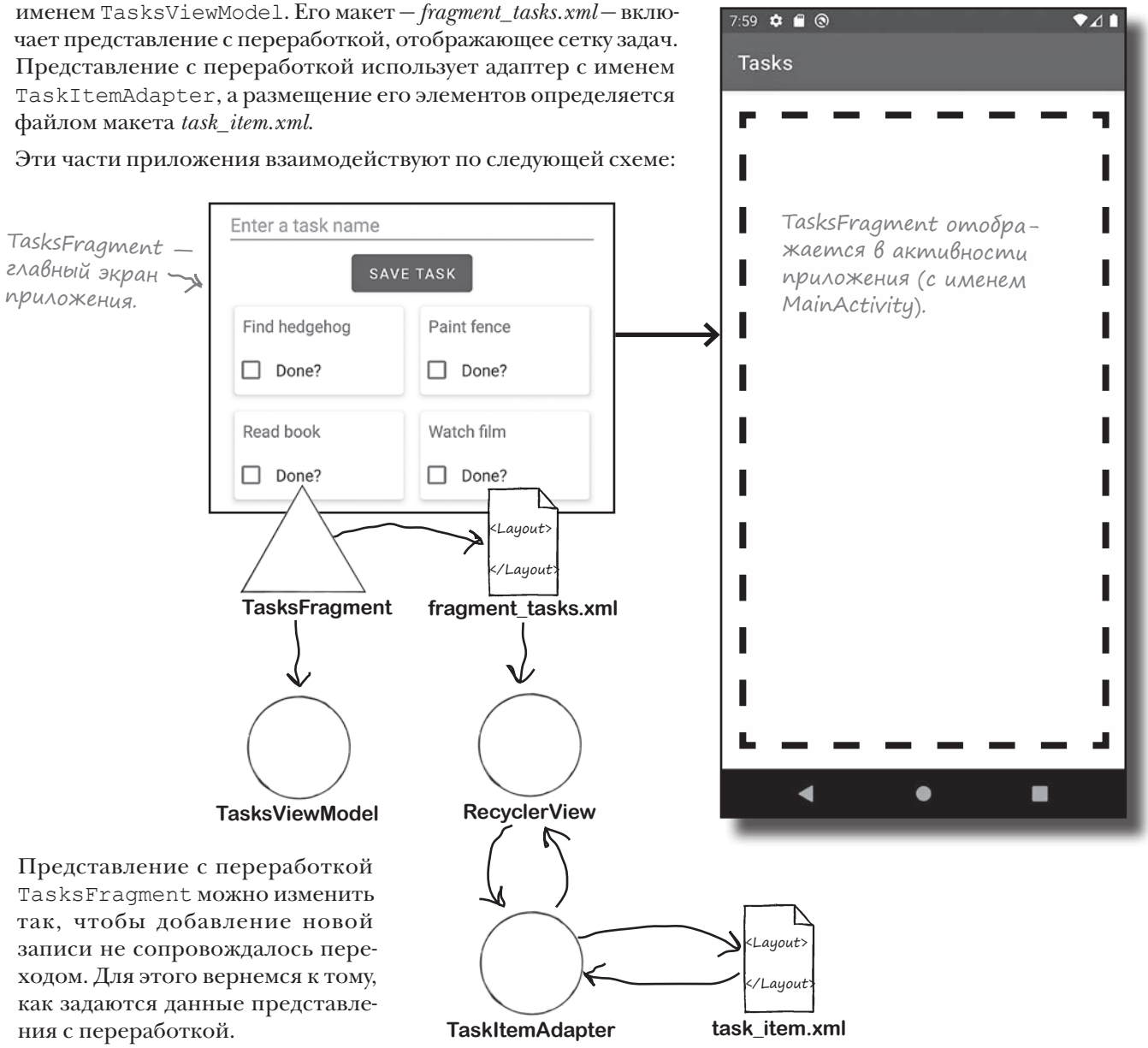
Прежде чем браться за решение этих проблем, вспомним структуру приложения Tasks.

Снова о приложении Tasks

Как говорилось ранее, приложение Tasks предоставляет пользователю возможность вводить записи, которые сохраняются в базе данных Room. Приложение включает представление с переработкой, в котором отображаются все введенные записи.

Главный экран приложения определяется фрагментом `TasksFragment`, который использует модель представления с именем `TasksViewModel`. Его макет — *fragment_tasks.xml* — включает представление с переработкой, отображающее сетку задач. Представление с переработкой использует адаптер с именем `TaskItemAdapter`, а размещение его элементов определяется файлом макета *task_item.xml*.

Эти части приложения взаимодействуют по следующей схеме:

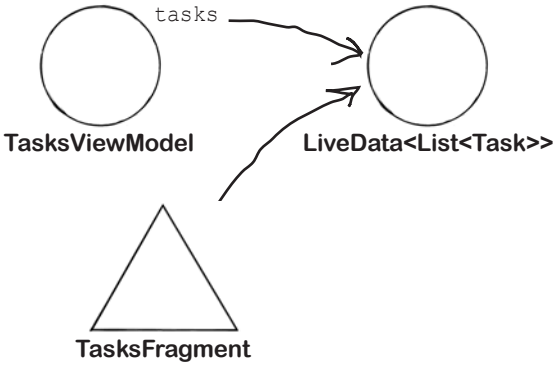


Представление с переработкой `TasksFragment` можно изменить так, чтобы добавление новой записи не сопровождалось переходом. Для этого вернемся к тому, как задаются данные представления с переработкой.

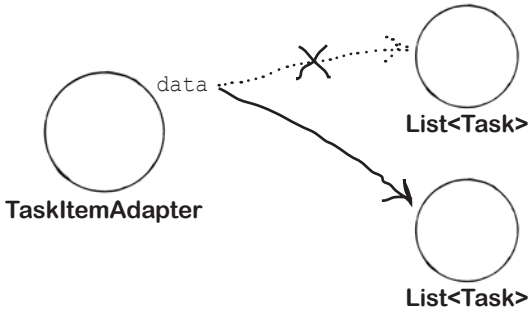
Как представление с переработкой получает свои данные

Когда возникает необходимость в обновлении данных представления с переработкой, происходит следующее:

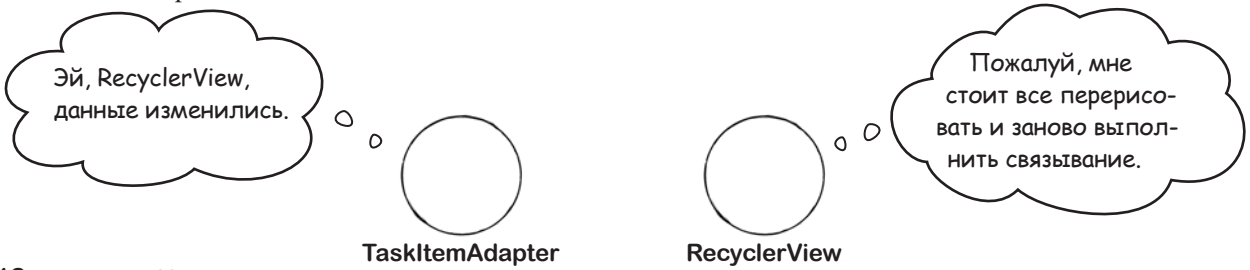
1. **TasksFragment оповещается о добавлении записи в базу данных.**
Это происходит из-за того, что он наблюдает за свойством `tasks` объекта `TasksViewModel`: списком `LiveData<List<Task>>`, который получает свои данные из базы данных.



2. **TasksFragment задает свойство data объекта TaskItemAdapter, который содержит данные представления с переработкой.**
Ему присваивается новый список `List<Task>` (который он получает из свойства `tasks`), который включает последние изменения в записях.



3. **TaskItemAdapter оповещает представление с переработкой о том, что данные изменились.**
Представление с переработкой реагирует на оповещение перерисовкой и повторным связыванием каждого элемента в списке.



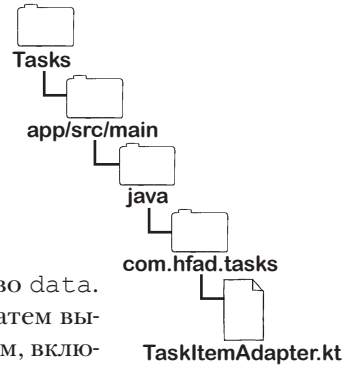
Метод записи свойства data вызывает notifyDataSetChanged()

Представление с переработкой перерисовывает и заново связывает весь список из-за метода записи, который был добавлен к свойству `data` объекта `TaskItemAdapter`. Напомним, как выглядит код:

```
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }
    ...
}
```

Метод записи свойства data.

Этот метод сообщает представлению с переработкой о том, что данные изменились.



Метод записи вызывается каждый раз, когда потребуется обновить свойство `data`. Как видно из листинга, он присваивает новое значение свойству `data`, а затем вызывает `notifyDataSetChanged()`. Этот метод сообщает всем наблюдателям, включая представление с переработкой, о том, что набор данных изменился, поэтому представление с переработкой перерисовывается с учетом последних изменений.

notifyDataSetChanged() перерисовывает весь список

Тем не менее использование `notifyDataSetChanged()` создает проблемы. При каждом вызове метод сообщает, что свойство `data` устарело в каком-то отношении, *но не указывает, в каком именно.* Так как представление с переработкой не знает, что изменилось, оно реагирует на оповещение **повторным связыванием и перерисовкой каждого элемента в списке**.

Когда все представление с переработкой заново связывает и перерисовывает элементы подобным образом, оно теряет текущую позицию пользователя в списке. Если список содержит хотя бы десяток записей, это может привести к переходу и смене отображаемой части списка.

Кроме того, такой подход неэффективен для больших наборов данных. Если представление с переработкой содержит много элементов, повторное связывание и перерисовка создают большой объем лишней работы и могут вызывать проблемы с быстродействием.

Каждый раз, когда вызывается метод notifyDataSetChanged(), представление с переработкой заново связывает и перерисовывает весь список. Такое решение неэффективно, особенно для больших наборов данных.

Мозговой штурм

Как предотвратить перерисовку всего списка представления с переработкой каждый раз, когда пользователь вводит новую запись?

Передача представлению с переработкой информации о том, что должно измениться

В другом, более эффективном варианте метод `notifyDataSetChanged()` вызывается для оповещения представления с переработкой о том, какие элементы в списке изменились, чтобы представление обновило только эти элементы. Например, если в базу данных добавляется новая запись, то для представления с переработкой будет более эффективно просто добавить новый элемент, чем заново связывать и перерисовывать весь список.

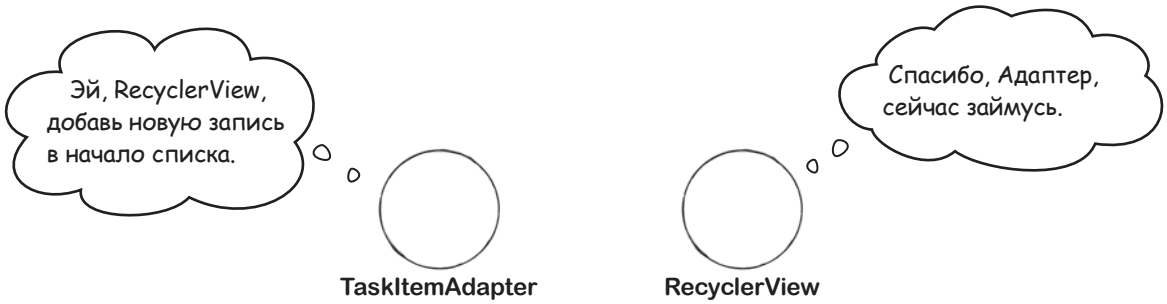
Проверять все изменения вручную было бы слишком хлопотно и потребовало бы слишком большого объема кода. К счастью, библиотека представлений с переработкой включает вспомогательный класс с именем **`DiffUtil`**, который выполняет всю черную работу за вас.

DiffUtil определяет отличия между списками

Класс `DiffUtil` специализируется на определении отличий между двумя списками, чтобы вам не приходилось заниматься этой работой.

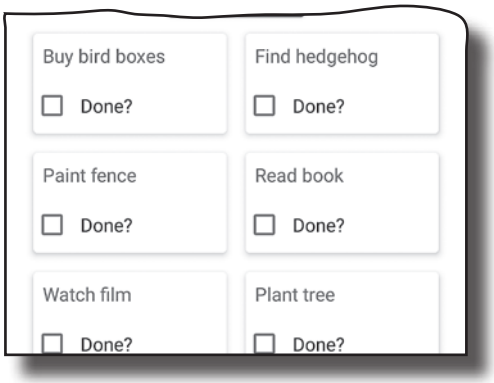
Каждый раз, когда адаптеру передается новая версия списка, используемого его представлением с переработкой, `DiffUtil` сравнивает ее со старой версией. Класс определяет, какие элементы были добавлены, удалены или обновлены, и сообщает представлению с переработкой, что следует изменить, наиболее эффективным способом из всех возможных:

Если выражаться точнее, для определения того, что изменилось, используется хитроумный алгоритм Юджина У. Майерса.



Так как представлению с переработкой уже не нужно перерисовывать и проводить повторное связывание всего списка, использование `DiffUtil` намного эффективнее обновления данных представления с переработкой. Также это означает, что пользователь не теряет текущую позицию в списке, и представление с переработкой даже может обеспечивать плавные переходы с анимацией, которые ясно показывают, какие изменения были внесены.

Когда пользователь вводит новую запись, она добавляется в список. Другие элементы перемещаются, чтобы освободить для нее место.

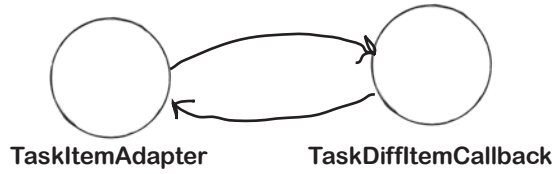


Что мы собираемся сделать

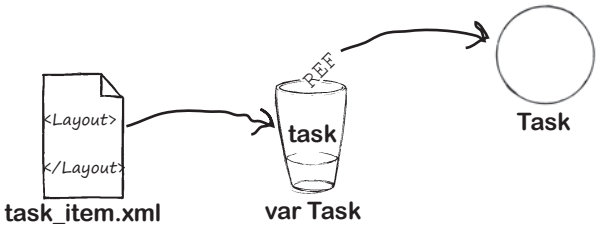
В этой главе мы внесем изменение в представление с переработкой приложения Tasks, чтобы в нем использовался класс `DiffUtil`, а представления заполнялись посредством связывания данных. Эти изменения повышают эффективность представления с переработкой и улучшают впечатления пользователя от работы с ним.

Для этого необходимо:

1. **Использовать DiffUtil в представлении с прокруткой**
   Мы создадим новый класс с именем `TaskDiffItemCallback`, который использует `DiffUtil` для сравнения элементов в списке. Затем мы обновим код `TaskItemAdapter`, чтобы в нем использовался этот новый класс. Эти изменения повышают эффективность представления с переработкой, а работа пользователя с представлением становится более плавной и приятной.



2. **Реализовать связывание данных в макете представления с переработкой.**
   Мы удалим вызовы `findViewById()` в коде `TaskItemAdapter` и заполним представления каждого элемента с использованием связывания данных.



Для начала заставим представление с переработкой использовать `DiffUtil`.

Не забудьте!

Мы собираемся изменить приложение Tasks. Откройте проект этого приложения.

Реализация DiffUtil.ItemCallback

Чтобы использовать `DiffUtil` для представления с переработкой приложения Tasks, необходимо создать новый класс (назовем его `TaskDiffItemCallback`), реализующий абстрактный класс **`DiffUtil.ItemCallback`**. Этот класс используется для вычисления разности между двумя элементами списка, отличными от null, и помогает значительно повысить эффективность представлений с переработкой.

Реализуя `DiffUtil.ItemCallback`, сначала необходимо задать тип объектов, с которыми он работает. Для этого будет использоваться обобщение:

Позволяет классу работать с объектами Task.

```
class TaskDiffItemCallback : DiffUtil.ItemCallback<Task>() {...}
```

Также необходимо переопределить два метода: `areItemsTheSame()` и `areContentsTheSame()`.

Метод **`areItemsTheSame()`** используется для проверки того, ссылаются ли два переданных ему объекта на один *элемент*. Для его реализации будет использоваться следующий код:

Если oldItem и newItem ссылаются на один элемент, метод должен возвращать true.

```
override fun areItemsTheSame(oldItem: Task, newItem: Task)
    = (oldItem.taskId == newItem.taskId)
```

Если оба объекта имеют одинаковые значения `taskId`, это означает, что они ссылаются на один элемент, и метод возвращает *true*.

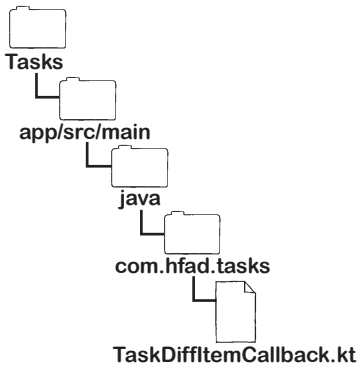
Метод **`areContentsTheSame()`** используется для проверки совпадения *содержимого* двух объектов и вызывается только в том случае, если `areItemsTheSame()` возвращает *true*. Так как `Task` является классом данных, этот метод может быть реализован следующим кодом:

Если oldItem и newItem имеют одинаковое содержимое, метод должен возвращать true. Так как Task является классом данных, oldItem == newItem при совпадении значений их свойств.

```
override fun areContentsTheSame(oldItem: Task, newItem: Task) = (oldItem == newItem)
```

Создание TaskDiffItemCallback.kt

Чтобы создать новый класс, выделите пакет *com.hfad.tasks* в папке *app/src/main/java*, затем выберите команду File→New→Kotlin Class/File. Введите имя файла «TaskDiffItemCallback» и выберите вариант Class.



После того как файл будет создан, обновите его код:

```
package com.hfad.tasks

import androidx.recyclerview.widget.DiffUtil

class TaskDiffItemCallback : DiffUtil.ItemCallback<Task>() {
    override fun areItemsTheSame(oldItem: Task, newItem: Task)
        = (oldItem.taskId == newItem.taskId)

    override fun areContentsTheSame(oldItem: Task, newItem: Task) = (oldItem == newItem)
}
```

ListAdapter получает аргумент DiffUtil.ItemCallback

ListAdapter — разновидность RecyclerView.Adapter, предоставляющая собственный резервный список. Хорошо работает в сочетании с DiffUtil.

После определения `TaskDiffItemCallback` необходимо использовать его в коде адаптера. Для этого мы обновим класс `TaskItemAdapter`, чтобы он расширял класс **`ListAdapter`** вместо `RecyclerView.Adapter`.

`ListAdapter` — разновидность `RecyclerView.Adapter`, спроектированная для работы со списками. Он предоставляет собственный резервный список, чтобы вам не приходилось определять его самостоятельно, и получает `DiffUtil.ItemCallback` в конструкторе.

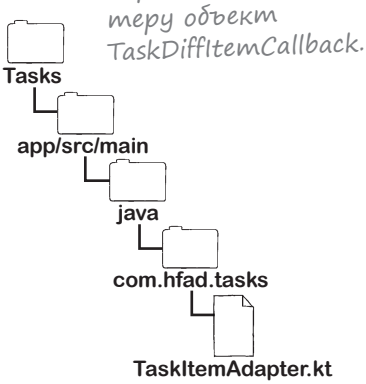
Мы укажем, что `TaskItemAdapter` является разновидностью `ListAdapter` с собственным списком `List<Task>`, и передадим ему экземпляр `TaskDiffItemCallback`. Для этого используется следующий код:

```
...
import androidx.recyclerview.widget.ListAdapter

class TaskItemAdapter
        : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
    ...
}
```

Использует класс ListAdapter из библиотеки androidx.recyclerview.

Превращает TaskItemAdapter в List Adapter, работающий с объектами Task и TaskItemViewHolder.

Передает адаптеру объект TaskDiffItemCallback.



Остальной код TaskItemAdapter можно упростить

После того как класс `TaskItemAdapter` был изменен для расширения `ListAdapter`, можно удалить его свойство `data` типа `List<Task>` вместе с его методом записи. Это свойство стало ненужным, потому что `ListAdapter` содержит собственный резервный список, так что вам не придется определять его самостоятельно.

Также можно удалить метод `getItemCount()` объекта `TaskItemAdapter`. Он был необходим, когда адаптер расширял `RecyclerView.Adapter`, но `ListAdapter` предоставляет собственную реализацию, поэтому этот метод стал лишним.

Наконец, необходимо обновить метод `onBindViewHolder()` адаптера, чтобы вместо

```
val item = data[position]
```

для получения элемента в определенной позиции свойства `data` использовался код

```
val item = getItem(position)
```

Этот код получает элемент в заданной позиции резервного списка адаптера.

Весь этот код будет приведен на следующей странице.

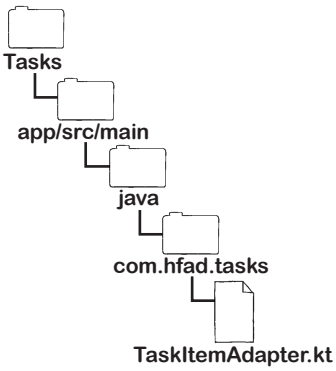
DiffUtil
Связывание данных

Обновленный код TaskItemAdapter.kt

Ниже приведен обновленный код `TaskItemAdapter`; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import androidx.recyclerview.widget.ListAdapter
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView
```

Импортируйте этот класс. В Android есть несколько разных классов ListAdapter; проследите за тем, чтобы импортировался правильный класс.




```
class TaskItemAdapter : RecyclerView.Adapter<TaskItemAdapter.TaskItemViewHolder>() {
    : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {
    var data = listOf<Task>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun getItemCount() = data.size

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = data[position] getItem(position)
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView: CardView) : RecyclerView.ViewHolder(rootView) {
        ...
    }
}
```

Свойство data можно удалить, оно стало лишним.

Измените определение класса, чтобы TaskItemAdapter расширял ListAdapter вместо RecyclerView.Adapter.

Этот метод более не нужен.

Получение элемента из резервного списка адаптера.

Подробности кода держателя представления опущены, так как изменять его не нужно.

Заполнение списка ListAdapter...

Последнее, что осталось сделать, — передать список записей `Task` резервному списку `TaskItemAdapter`.

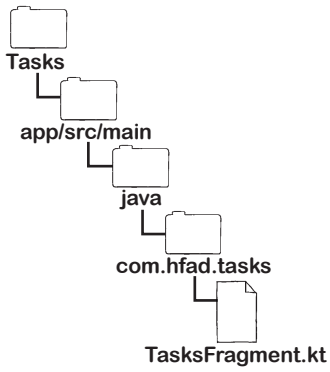
Ранее для этого мы отдавали команду `TasksFragment` наблюдать за свойством `tasks` объекта `TasksViewModel`. Каждый раз, когда свойство изменялось, фрагмент обновлял свойство `data` объекта `TaskItemAdapter` новым значением свойства `tasks`.

Напомним, как выглядел код, который для этого использовался:

```
viewModel.tasks.observe(viewLifecycleOwner, Observer {
    it?.let {
        adapter.data = it
    }
})
```

Присваивает свойству data адаптера новый список задач.

Итак, теперь адаптер использует резервный список вместо свойства `data`, и решение необходимо слегка изменить.



...с помощью submitList()

Для передачи списка задач резервному списку `TaskItemAdapter` будет использоваться метод `submitList()`. Этот метод обновляет резервный список `ListAdapter` новым объектом `List`, поэтому он идеально подходит для этой ситуации.

Ниже приведен новый код, который необходимо добавить в `TasksFragment` (он выделен жирным шрифтом); мы добавим его на следующей странице:

```
viewModel.tasks.observe(viewLifecycleOwner, Observer {
    it?.let {
        ~~adapter.data = it~~
        adapter.submitList(it)
    }
})
```

Метод submitList() будет использоваться для передачи новых данных резервному списку адаптера.

Когда адаптер получает новый список, он использует класс `TaskDiffItemCallback` для того, чтобы сравнить его со старой версией. Затем представление с переработкой обновляется найденными различиями — вместо замены всего списка. Такой подход более эффективен, а представление работает более плавно.

Давайте посмотрим, как выглядит обновленный код `TasksFragment`.

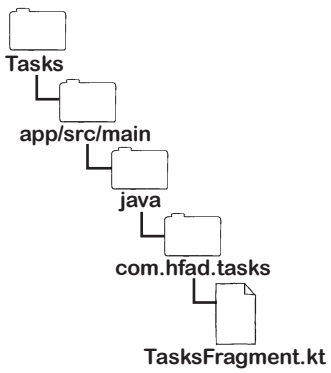
DiffUtil
Связывание данных

Обновленный код TasksFragment.kt

Ниже приведен обновленный код `TasksFragment`; обновите файл *TasksFragment.kt* (изменения выделены жирным шрифтом):

```
...
class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
        val adapter = TaskItemAdapter()
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.~~data = it~~ submitList(it)
            }
        })
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Обновите эту строку.

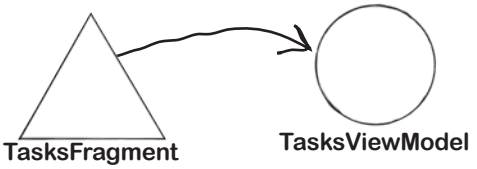


Посмотрим, что происходит при выполнении приложения.

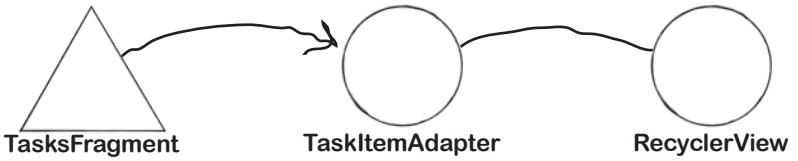
Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

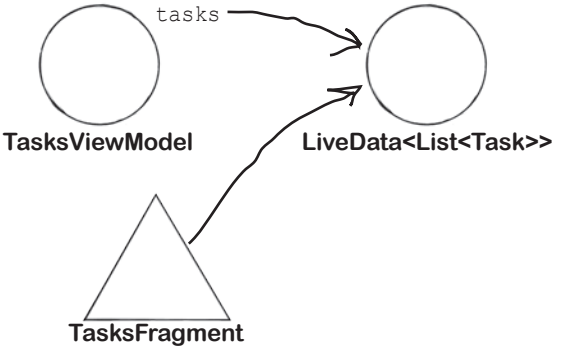
1. **При запуске приложения MainActivity отображает TasksFragment.** `TasksFragment` использует `TasksViewModel` в качестве модели представления.



2. **TasksFragment создает объект TaskItemAdapter и назначает его адаптером для представления с переработкой.**

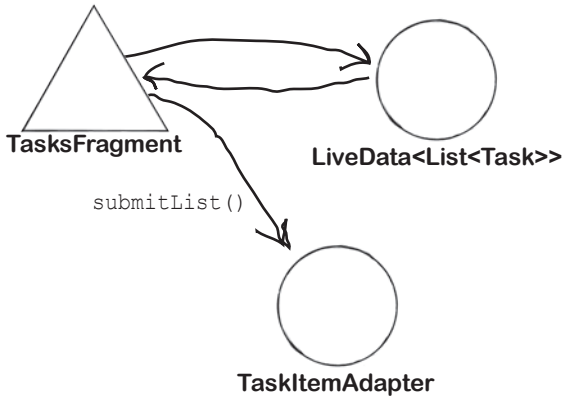


3. **TasksFragment наблюдает за свойством tasks объекта TasksViewModel.** Это свойство содержит тип `LiveData<List<Task>>`, в котором хранится актуальный список записей из базы данных.



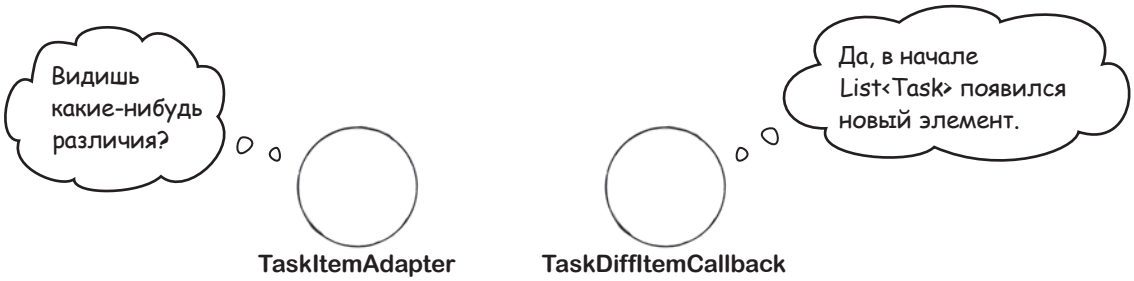
История продолжается

4 **Каждый раз, когда свойство tasks получает новое значение, TasksFragment передает свой список List<Task> адаптеру TaskItemAdapter.**



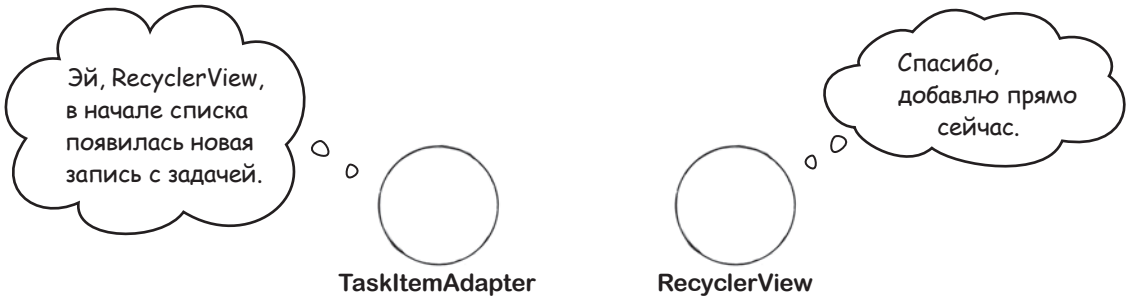
5 **TaskItemAdapter использует TaskDiffItemCallback для сравнения старых данных с новыми.**

Для определения того, что изменилось в данных, используются методы `areItemsTheSame()` и `areContentsTheSame()` объекта `TaskDiffItemCallback`.



6 **TaskItemAdapter передает представлению с переработкой информацию об изменениях.**

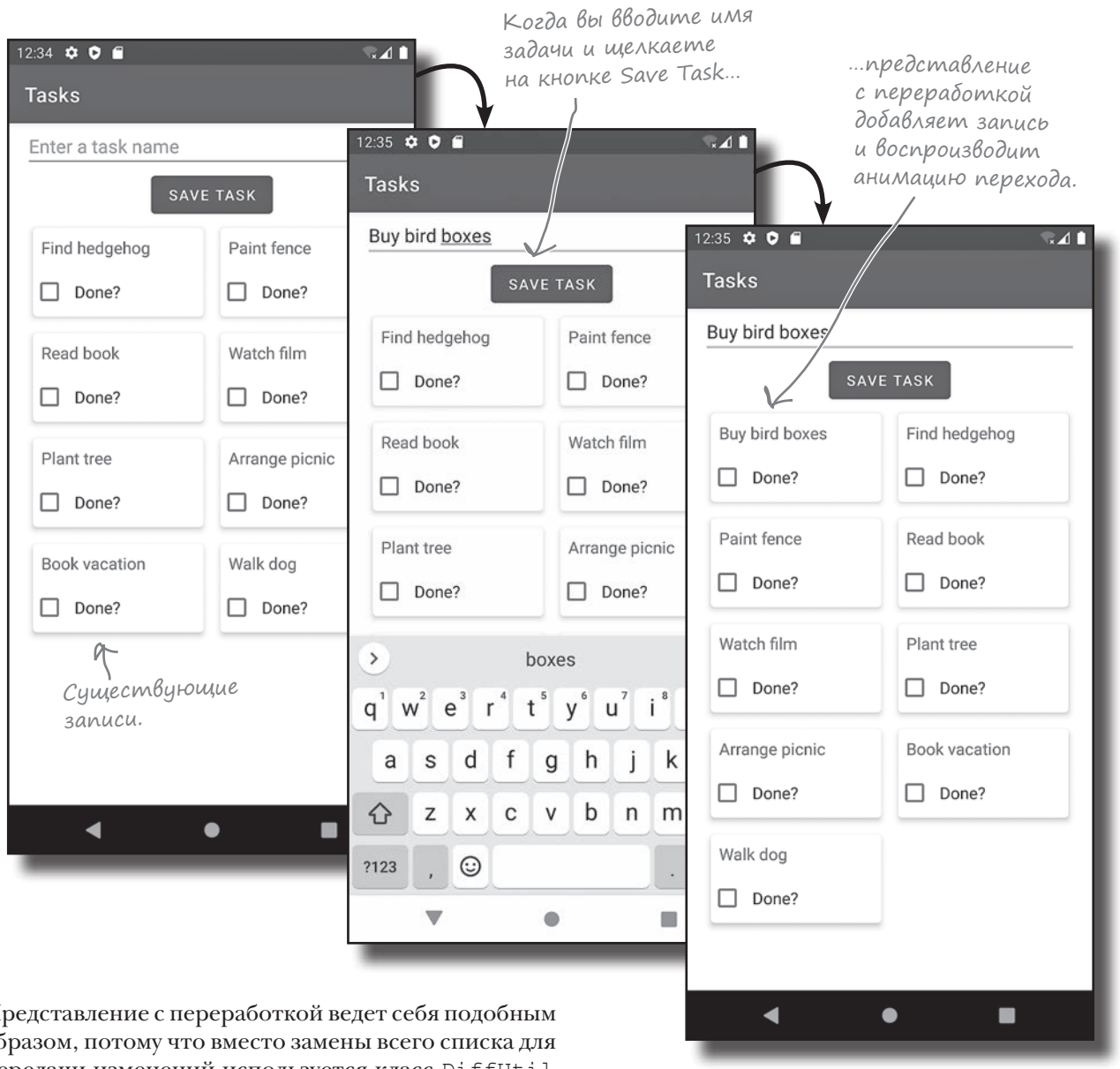
Представление с переработкой заново связывает и перерисовывает необходимые элементы.



Тест-драйв

Когда вы запускаете приложение, `TasksFragment`, как и прежде, отображает сетку карточек в представлении с переработкой.

Если ввести имя новой задачи и щелкнуть на кнопке, в представление с переработкой добавляется новая карточка задачи, а существующие карточки смещаются, чтобы освободить для нее место.



Представление с переработкой ведет себя подобным образом, потому что вместо замены всего списка для передачи изменений используется класс `DiffUtil`.

СТАНЬ ListAdapter

Класс ListAdapter представления с переработкой содержит резервный список объектов Drink. Он использует класс Drink, приведенный справа. Представьте себя на месте ListAdapter и скажите, будут ли приведенные ниже классы ItemCallback правильно обнаруживать изменения при получении нового списка ListAdapter. Почему?

Подсказка: свойство drinkId содержит уникальный идентификатор для каждого объекта Drink.

```
class Drink(
    var drinkId: Long = 0L,
    var drinkName: String = ""
)
```

A

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)
}
```

B

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = ((oldItem.drinkId == newItem.drinkId) &&
           (oldItem.drinkName == newItem.drinkName))
}
```

C

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkName == newItem.drinkName)
}
```

Ответ на с. 740.

Представления с переработкой могут использовать связывание данных

Другое возможное усовершенствование представления с переработкой приложения Tasks — переход на связывание данных.

Как вы помните, класс `TaskItemViewHolder`, являющийся внутренним классом `TaskItemAdapter`, использует метод `findViewById()` для получения ссылок на представления всех элементов в представлении с переработкой. Затем метод `bind()` держателя представления использует эти ссылки для добавления данных в каждое представление.

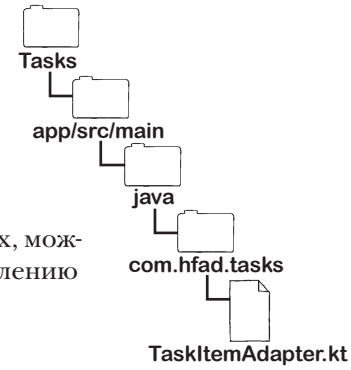
Напомним, как выглядит код:

```
class TaskItemViewHolder(val rootView: CardView)
                                    : RecyclerView.ViewHolder(rootView) {
    val taskName = rootView.findViewById<TextView>(R.id.task_name)
    val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)
    ...

    fun bind(item: Task) {
        taskName.text = item.taskName
        taskDone.isChecked = item.taskDone
    }
}
```

Получает ссылки на представления из макета элемента.

Заполняет представления данными.

Если перевести представление с переработкой на связывание данных, можно удалить вызовы `findViewById()` и поручить каждому представлению загрузку его собственных данных.



Как реализовать связывание данных

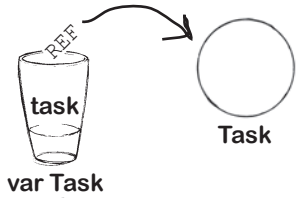
Представление с переработкой переводится на связывание данных примерно по той же схеме, как и для фрагментов. Необходимо сделать следующее:

1. **Добавить переменную связывания данных в task_item.xml.**
   Мы сделаем `<layout>` корневым элементом макета и создадим переменную связывания данных с именем `task` и типом `Task`. При этом будет сгенерирован класс связывания с именем `TaskItemBinding`.

2. **Присвоить значение переменной связывания данных в TaskItemAdapter.**
   Объект `TaskItemBinding` используется для заполнения макета каждого элемента, а его переменной связывания данных присваивается объект `Task` для этого элемента.

3. **Использовать переменную связывания данных для заполнения представлений данными.**
   Наконец, мы обновим файл *task_item.xml*, чтобы каждое представление загружало свои данные из объекта `Task` макета.

Начнем с определения переменной связывания данных.



Чтобы перевести представление с переработкой на связывание данных, следует добавить переменную связывания данных Task (с именем task) в макет, используемый его элементами. Тогда каждое представление сможет получать данные из объекта Task макета.

Включение переменной связывания данных в task_item.xml

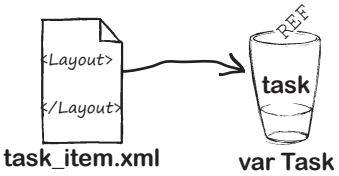
Начнем с добавления элемента `<layout>` в корневой элемент *task_item.xml* и определения переменной связывания данных. Но вместо того чтобы использовать его для привязки представлений к модели представления, мы укажем, что оно имеет тип `Task`.

Ниже приведен код для решения этой задачи; обновите файл *task_item.xml* (изменения выделены жирным шрифтом):

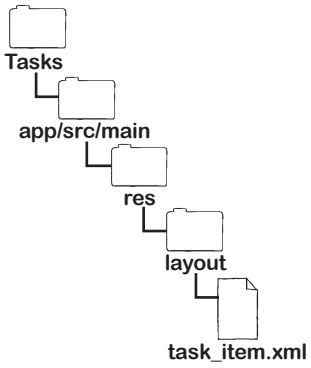
```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable
            name="task"
            type="com.hfad.tasks.Task" />
    </data>

    <androidx.cardview.widget.CardView
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        app:cardElevation="4dp"
        app:cardCornerRadius="4dp" >

        ...

    </androidx.cardview.widget.CardView>
</layout>
```

Добавляем элемент <layout>.

Определяем переменную связывания данных с именем task.



Эти строки перемещаются в <layout>.



Закрывающий тег элемента <layout>.

Назначение элемента `<layout>` корневым элементом файла *task_item.xml* сообщает Android, что макет должен использоваться со связыванием данных, поэтому он генерирует новый класс связывания с именем `TaskItemBinding`. Мы воспользуемся этим классом для заполнения приведенного выше макета и присвоим его переменной связывания данных объект `Task`.

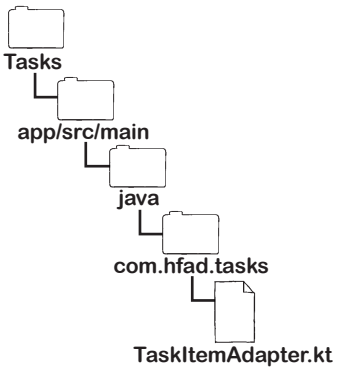
Макет заполняется в коде держателя представления адаптера

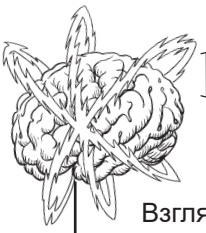
При создании представления с переработкой мы заполнили файл макета *task_item.xml* в `TaskItemViewHolder` — внутреннем классе `TaskItemAdapter`. Теперь необходимо изменить код, чтобы он работал с классом связывания `TaskItemBinding`. Но прежде чем браться за дело, напомним текущую версию кода.

```
class TaskItemAdapter...{
    ...
    class TaskItemViewHolder(val rootView: CardView)
                                        : RecyclerView.ViewHolder(rootView) {
        val taskName = rootView.findViewById<TextView>(R.id.task_name)
        val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)

        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as CardView
                return TaskItemViewHolder(view)
            }
        }

        fun bind(item: Task) {
            taskName.text = item.taskName
            taskDone.isChecked = item.taskDone
        }
    }
}
```

Заполняет макет и возвращает объект TaskItemViewHolder.

Добавляет данные в каждое представление.





Мозговой штурм

Взгляните на приведенный выше код `TaskItemViewHolder`. Как вы думаете, какие изменения необходимо внести, чтобы связать *task_item.xml* с объектом `Task`?

DiffUtil
Связывание данных

Использование класса связывания для заполнения макета

Первое изменение, которое мы внесем в `TaskItemViewHolder`, — заполнение *task_item.xml* с использованием класса `TaskItemBinding`. Для этого мы воспользуемся методом `inflateFrom()` держателя представления:

```
fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
    val layoutInflater = LayoutInflater.from(parent.context)
    val view = layoutInflater.inflate(R.layout.task_item, parent, false) as CardView
    val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
    return TaskItemViewHolder(view binding)
}
```

Переменная связывания используется для создания TaskItemViewHolder.

Макет заполняется с использованием TaskItemBinding.

Обратите внимание на передачу переменной связывания — объекта `TaskItemBinding` — конструктору `TaskItemViewHolder`. А значит, также нужно обновить определение класса `TaskItemViewHolder` и привести его к следующему виду:

```
class TaskItemViewHolder(val rootView: CardView) : RecyclerView.ViewHolder(rootView) {
class TaskItemViewHolder(val binding: TaskItemBinding)
        : RecyclerView.ViewHolder(binding.root) {
    ...
}
```

TaskItemViewHolder должен получать объект TaskItemBinding.

Становится корневым представлением макета.

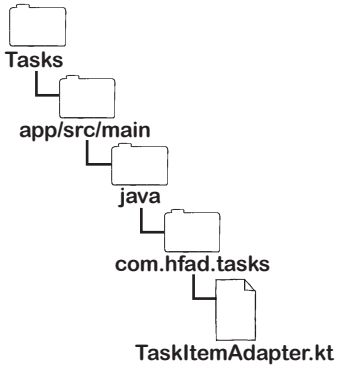
Присваивание объекта Task переменной связывания данных макета

Теперь в приложении класс `TaskItemBinding` используется для заполнения макета *task_item.xml*, и мы можем воспользоваться им для присваивания переменной связывания данных `task`. Для этого мы изменим метод `bind()` объекта `TaskItemViewHolder`, чтобы он присваивал `task` текущему элементу `Task` представления с переработкой:

```
fun bind(item: Task) {
    taskName.text = item.taskName
    taskDone.isChecked = item.taskDone
    binding.task = item
}
```

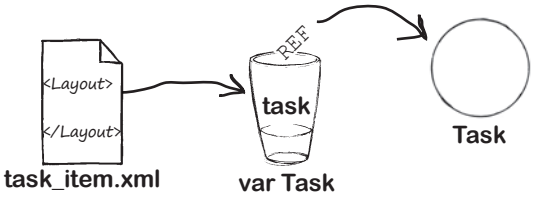
Держателю представления уже не нужно добавлять данные в представления макета.

Переменной связывания данных присваивается Task.



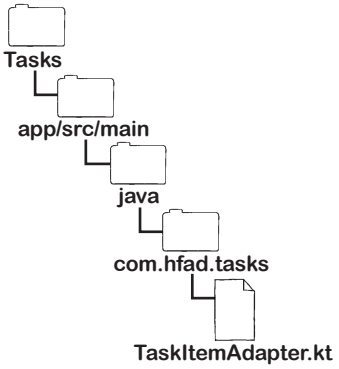
Строки, в которых заполняются представления `task_name` и `task_done` макета, стали лишними. Это означает, что мы также можем удалить свойства `taskName` и `taskDone` из держателя представления.

Полный код `TaskItemAdapter` (включая внутренний класс `TaskItemViewHolder`) приведен на следующей странице.



Полный код TaskItemAdapter.kt

Ниже приведен обновленный код `TaskItemAdapter`; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.CheckBox
import androidx.recyclerview.widget.ListAdapter
import android.widget.TextView
import androidx.cardview.widget.CardView
import androidx.recyclerview.widget.RecyclerView
import com.hfad.tasks.databinding.TaskItemBinding

class TaskItemAdapter
      : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }

    class TaskItemViewHolder(val rootView binding: CardView TaskItemBinding)
                            : RecyclerView.ViewHolder(rootView binding.root) {
        val taskName = rootView.findViewById<TextView>(R.id.task_name)
        val taskDone = rootView.findViewById<CheckBox>(R.id.task_done)
```

Удалите эту строку.

И эти строки тоже.

Добавьте эту команду импортирования.

Надо переименовать.

Теперь TaskItemViewHolder передается объект TaskItemBinding вместо CardView.

Тоже изменяем.

Удалите эти строки.

Продолжение на следующей странице.

TaskItemAdapter.kt (продолжение)

DiffUtil
Связывание данных

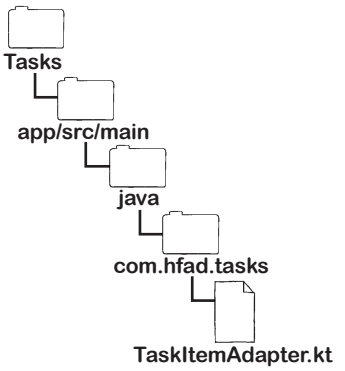
```
        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val view = layoutInflater
                        .inflate(R.layout.task_item, parent, false) as CardView
                val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
                return TaskItemViewHolder(view binding)
            }
        }

        fun bind(item: Task) {
            taskName.text = item.taskName
            taskDone.isChecked = item.taskDone
            binding.task = item
        }
    }
}
```

Макет заполняется с использованием TaskItemBinding.

Переменная связывания передается конструктору TaskItemViewHolder.

Удалите эти строки.

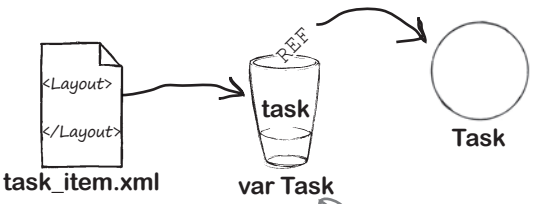
Присваиваем значение переменной связывания данных макета.



Использование связывания данных для заполнения представлений макета

Теперь, когда мы присвоили переменной связывания данных `task` макета *task_item.xml* элемент `Task` держателя представления, мы можем воспользоваться связыванием данных для заполнения представлений макета.

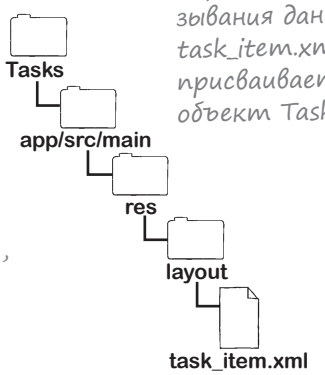
Код решения этой задачи вам уже знаком. Например, для заполнения текста представления `task_name` именем задачи можно использовать следующий код:



Переменной связывания данных task_item.xml присваивается объект Task.

```
<TextView
    android:id="@+id/task_name"
    ...
    android:text="@{task.taskName}" />
```

Представление заполняется именем задачи.



а состояние флажка `task_done` может задаваться следующим кодом:

```
<CheckBox
    android:id="@+id/task_done"
    ...
    android:checked="@{task.taskDone}" />
```

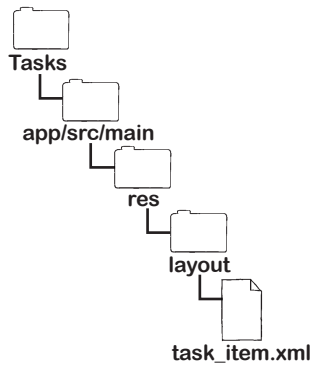
Если задача завершена, флажок будет установлен.

Полный код приведен на следующей странице.

Полный код task_item.xml

Ниже приведен обновленный код *task_item.xml*; обновите этот файл (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable
            name="task"
            type="com.hfad.tasks.Task" />
    </data>

    <androidx.cardview.widget.CardView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        app:cardElevation="4dp"
        app:cardCornerRadius="4dp" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" >

            <TextView
                android:id="@+id/task_name"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textSize="16sp"
                android:padding="8dp"
                android:text="@{task.taskName}" />
```



Выводится имя задачи.

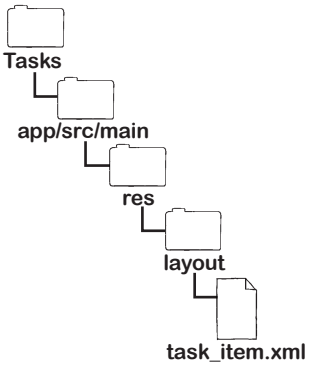
Продолжение на следующей странице.

DiffUtil
Связывание данных

Код task_item.xml (продолжение)

```
        <CheckBox
            android:id="@+id/task_done"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:padding="8dp"
            android:clickable="false"
            android:text="Done?"
            android:checked="@{task.taskDone}" />
    </LinearLayout>
</androidx.cardview.widget.CardView>
</layout>
```

Показывает, завершена ли задача.



Посмотрим, что происходит при выполнении кода, и проведем тест-драйв.

Часто Задаваемые Вопросы

В: Почему мы использовали `RecyclerView.Adapter` в предыдущей главе? Можно ли было использовать `ListAdapter`?

О: В предыдущей главе мы использовали `RecyclerView.Adapter`, чтобы вы получили представление о том, как работают представления с переработкой. `ListAdapter` — дополнительное усовершенствование, которое хорошо работает в сочетании с `DiffUtil`; именно поэтому мы представили его в этой главе.

В: Можно ли использовать `DiffUtil` без реализации `ListAdapter`?

О: Можно, но это более сложный путь. Проще воспользоваться `ListAdapter`.

В: Почему мы заменили эти вызовы `findViewById()`?

О: Метод `findViewById()` обладает рядом недостатков, которые были описаны в главе 10.

Во-первых, он может быть достаточно неэффективным. При каждом вызове метода Android приходится искать в иерархии макета представление с подходящим идентификатором. Если такое представление не будет найдено, приложение аварийно завершается.

Во-вторых, метод findViewById() небезопасен по отношению к типам. Если указать для искомого представления неправильный тип, то при попытке его использования в приложении может произойти ошибка.

В: Да, кажется, я припоминаю эти недостатки. А у связывания данных их нет?

О: Нет. Связывание данных безопаснее, потому что оно существенно снижает вероятность подобных ошибок времени выполнения.

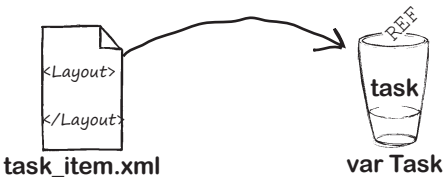
В: В файле *task_item.xml* вы определили переменную связывания данных с типом `Task`. Почему?

О: Потому что мы хотели отображать значения из объекта `Task` каждого элемента, а создание переменной этого типа позволяет задавать эти значения с применением связывания данных.

Что происходит при выполнении кода

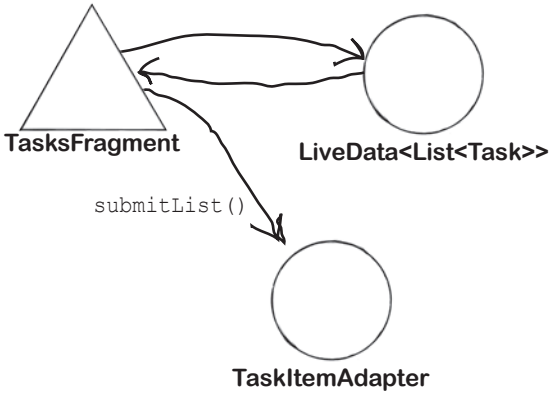
При выполнении приложения происходят следующие события:

1 **task_item.xml определяет переменную связывания данных Task с именем task.**
Так как *task_item.xml* содержит корневой элемент `<layout>`, для этого макета генерируется класс связывания с именем `TaskItemBinding`.



2 **TasksFragment создает объект TaskItemAdapter и назначает его адаптером представления с переработкой.**

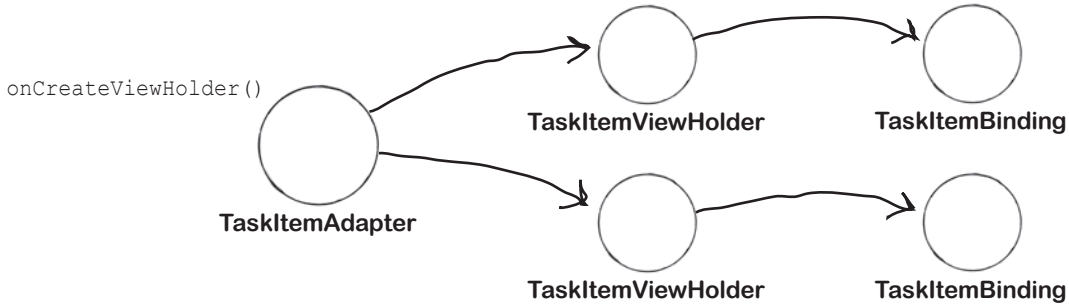
3 **TasksFragment передает List<Task> объекту TaskItemAdapter.**
`List<Task>` содержит обновленный список записей из базы данных.



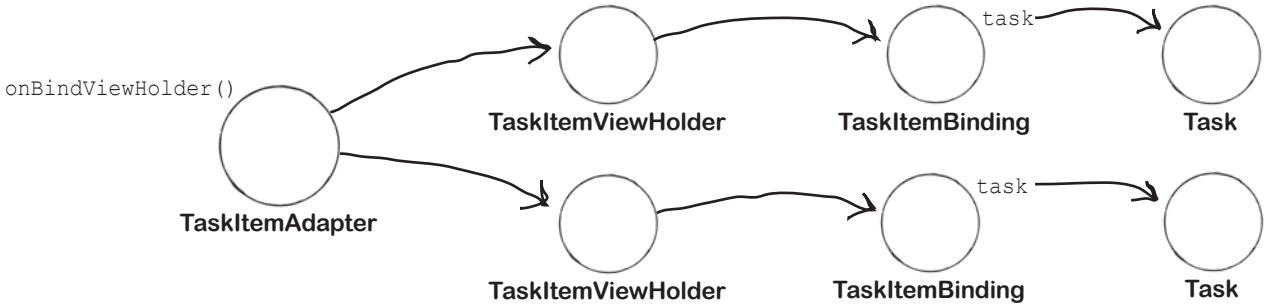
DiffUtil
Связывание данных

История продолжается

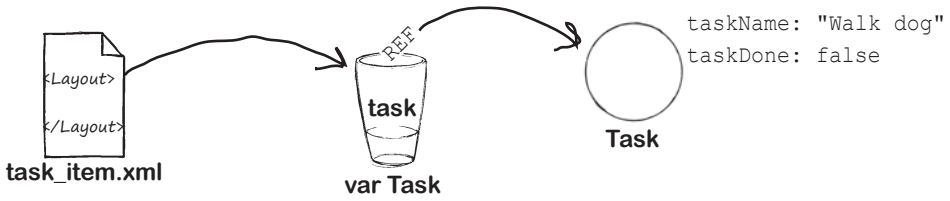
4 **Метод onCreateViewHolder() объекта TaskItemAdapter вызывается для каждого элемента, который должен отображаться в представлении с переработкой.**
`onCreateViewHolder()` вызывает метод `TaskItemViewHolder.inflateFrom()`, который создает объект `TaskItemBinding`. Он заполняет макет объекта и использует его для создания объекта `TaskItemViewHolder`.



5 **Метод onBindViewHolder() объекта TaskItemAdapter вызывается для каждого объекта TaskItemViewHolder.**
При этом вызывается метод `bind()` объекта `TaskItemViewHolder`, который использует объект `TaskItemBinding`, чтобы присвоить переменной `task` объект `Task` элемента.

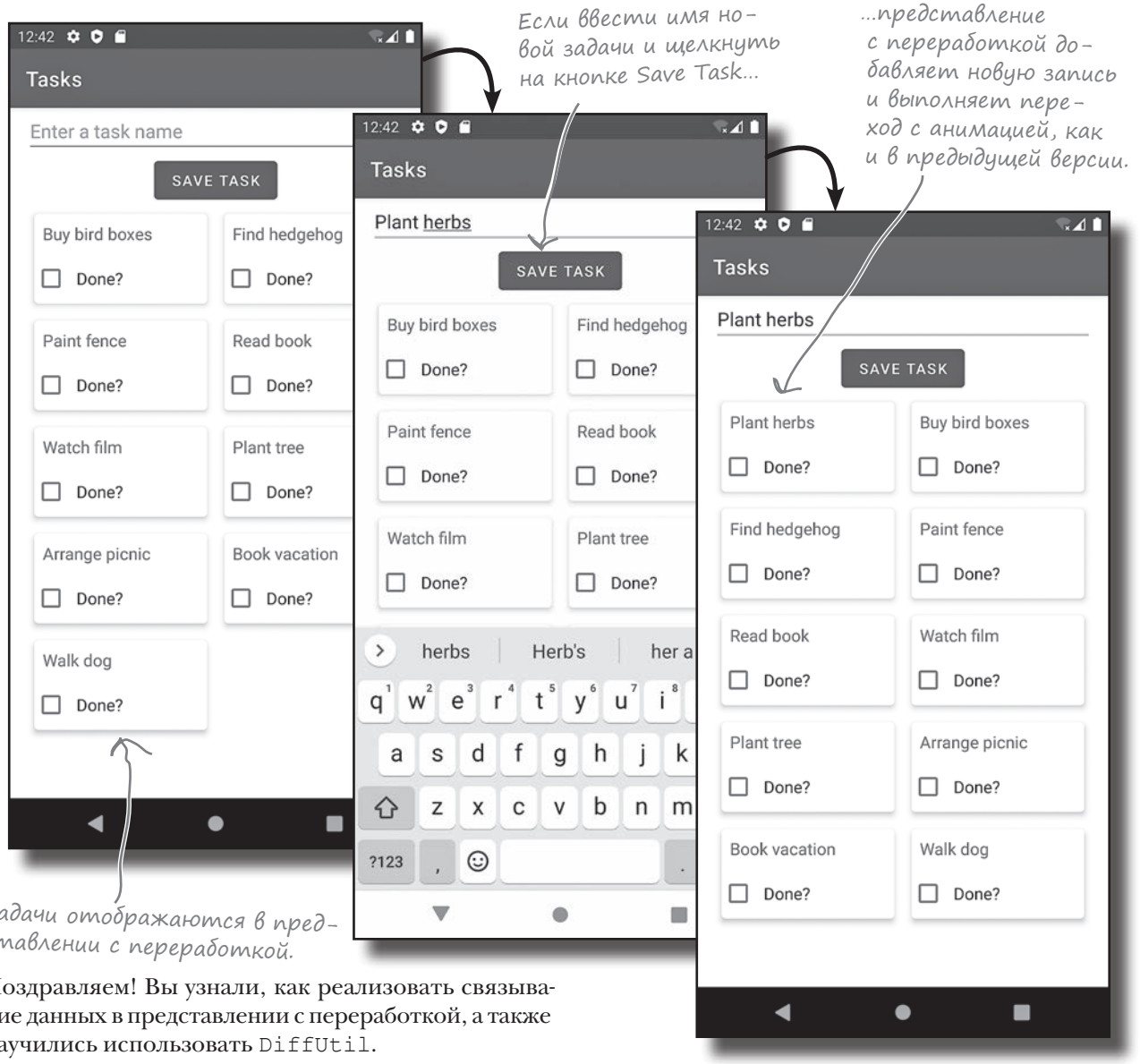


6 **Код связывания данных в task_item.xml использует свойство task для назначения представлений для каждого элемента.**
Свойству `text` представления `task_name` присваивается `task.taskName`, а свойству `checked` представления `task_done` присваивается `task.taskDone`.



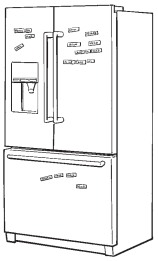
Тест-драйв

При запуске приложения `TasksFragment` отображает сетку карточек в представлении с переработкой. Представление ведет себя точно так же, как и в предыдущей версии, но на этот раз в его внутренней реализации используется связывание данных.



Поздравляем! Вы узнали, как реализовать связывание данных в представлении с переработкой, а также научились использовать `DiffUtil`.

В следующей главе мы воспользуемся новыми знаниями, чтобы организовать переход к отдельным записям в представлении с переработкой.



Развлечения с магнитами

Приложение Bits and Pizzas включает представление с переработкой, которое использует для своих элементов макет *pizza_item.xml*. Макет определяет переменную связывания данных (с именем `pizza`):

```
<layout...>
    <data>
        <variable
            name="pizza"
            type="com.hfad.bitsandpizzas.Pizza" />
    </data>
    ...
</layout>
```

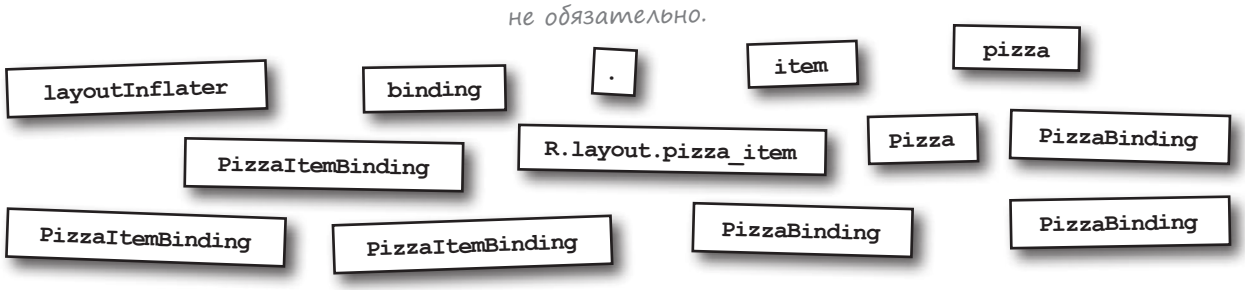
Представление с переработкой использует адаптер с именем `PizzaAdapter`, приведенный ниже. Попробуйте завершить код адаптера, чтобы в нем задавалось значение переменной связывания данных `pizza`.

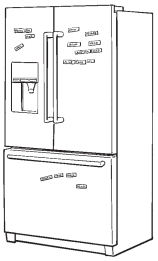
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView

import com.hfad.bitsandpizzas.databinding.  .....

class PizzaAdapter
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
```

Продолжение на следующей странице.

```
    class PizzaViewHolder(val binding: ..........)
                                    : RecyclerView.ViewHolder(binding.root) {

        companion object {
            fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)

                val binding = ..........

                            .inflate(..........., parent, false)

                return PizzaViewHolder(binding)
            }
        }

        fun bind(item: Pizza) {

         .......... = ..........
        }
    }
}
```

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Приложение Bits and Pizzas включает представление с переработкой, которое использует для своих элементов макет *pizza_item.xml*. Макет определяет переменную связывания данных (с именем `pizza`):

```
<layout...>
    <data>
        <variable
            name="pizza"
            type="com.hfad.bitsandpizzas.Pizza" />
    </data>
    ...
</layout>
```

Представление с переработкой использует адаптер с именем `PizzaAdapter`, приведенный ниже. Попробуйте завершить код адаптера, чтобы в нем задавалось значение переменной связывания данных `pizza`.

```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView

import com.hfad.bitsandpizzas.databinding.PizzaItemBinding

class PizzaAdapter
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
```

Импортируем класс связывания макета.

Продолжение на следующей странице.

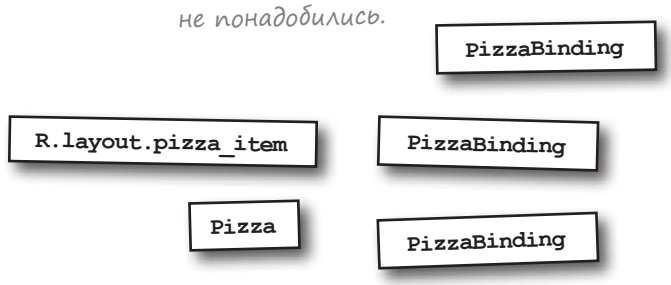
Объект PizzaViewHolder должен получать объект PizzaItemBinding.

```
class PizzaViewHolder(val binding: PizzaItemBinding)
                                   : RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)

            val binding = PizzaItemBinding.inflate(layoutInflater, parent, false)
            return PizzaViewHolder(binding)
        }
    }

    fun bind(item: Pizza) {
        binding.pizza = item
    }
}
```

PizzaItemBinding используется для заполнения макета.

Задается значение переменной связывания данных макета.

Эти магниты не понадобились.



СТАНЬ ListAdapter. Решение

Класс ListAdapter представления с переработкой содержит резервный список объектов Drink. Он использует класс Drink, приведенный справа. Представьте себя на месте ListAdapter и скажите, будут ли приведенные ниже классы ItemCallback правильно обнаруживать изменения при получении нового списка ListAdapter. Почему?

Подсказка: свойство drinkId содержит уникальный идентификатор для каждого объекта Drink.

```
class Drink(
    var drinkId: Long = 0L,
    var drinkName: String = ""
)
```

A

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)
}
```

Так как Drink не является классом данных, это выражение вернет true только в том случае, если oldItem и newItem ссылаются на один объект.

B

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem == newItem)
    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = ((oldItem.drinkId == newItem.drinkId) &&
           (oldItem.drinkName == newItem.drinkName))
}
```

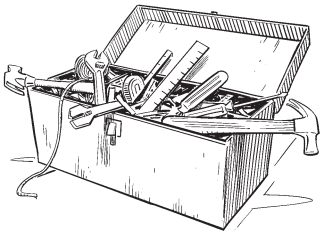
Это выражение должно проверять, имеют ли два элемента одинаковые идентификаторы. Вместо этого оно проверяет, ссылаются ли они на один объект.

C

```
class DrinkDiffItemCallback : DiffUtil.ItemCallback<Drink>() {
    override fun areItemsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkId == newItem.drinkId)

    override fun areContentsTheSame(oldItem: Drink, newItem: Drink)
        = (oldItem.drinkName == newItem.drinkName)
}
```

Это правильное выражение проверяет, что два элемента имеют одинаковые идентификаторы и содержимое.

Ваш инструментарий Android



Глава 16 осталась позади, а ваш инструментарий пополнился классом DiffUtil и связыванием данных в представлениях с переработкой.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Каждый раз, когда вызывается метод `notifyDataSetChanged()`, представление с переработкой выполняет повторное связывание и перерисовку каждого элемента.
- `DiffUtil` определяет, какие элементы изменились, и обновляет представление с переработкой наиболее эффективным образом.
- `ListAdapter` — разновидность `RecyclerView.Adapter`, работающая со списками. Он предоставляет собственный резервный список, а его конструктор получает `DiffUtil.ItemCallback`.
- `DiffUtil.ItemCallback` определяет различия между двумя элементами списка, отличными от null.
- Чтобы указать, ссылаются ли два объекта на один и тот же элемент, переопределите метод `areItemsTheSame()` объекта `DiffUtil.ItemCallback`.
- Чтобы указать, имеют ли два объекта одинаковое содержимое, переопределите метод `areContentsTheSame()` объекта `DiffUtil.ItemCallback`.
- Используйте метод `submitList()` для передачи новой версии списка `ListAdapter`.
- Представления с переработкой могут использовать связывание данных.
- Определите переменную связывания данных в макете, используемом для элементов представления с переработкой.
- Значение переменной связывания данных задается в коде адаптера представления с переработкой.

Карточные фокусы

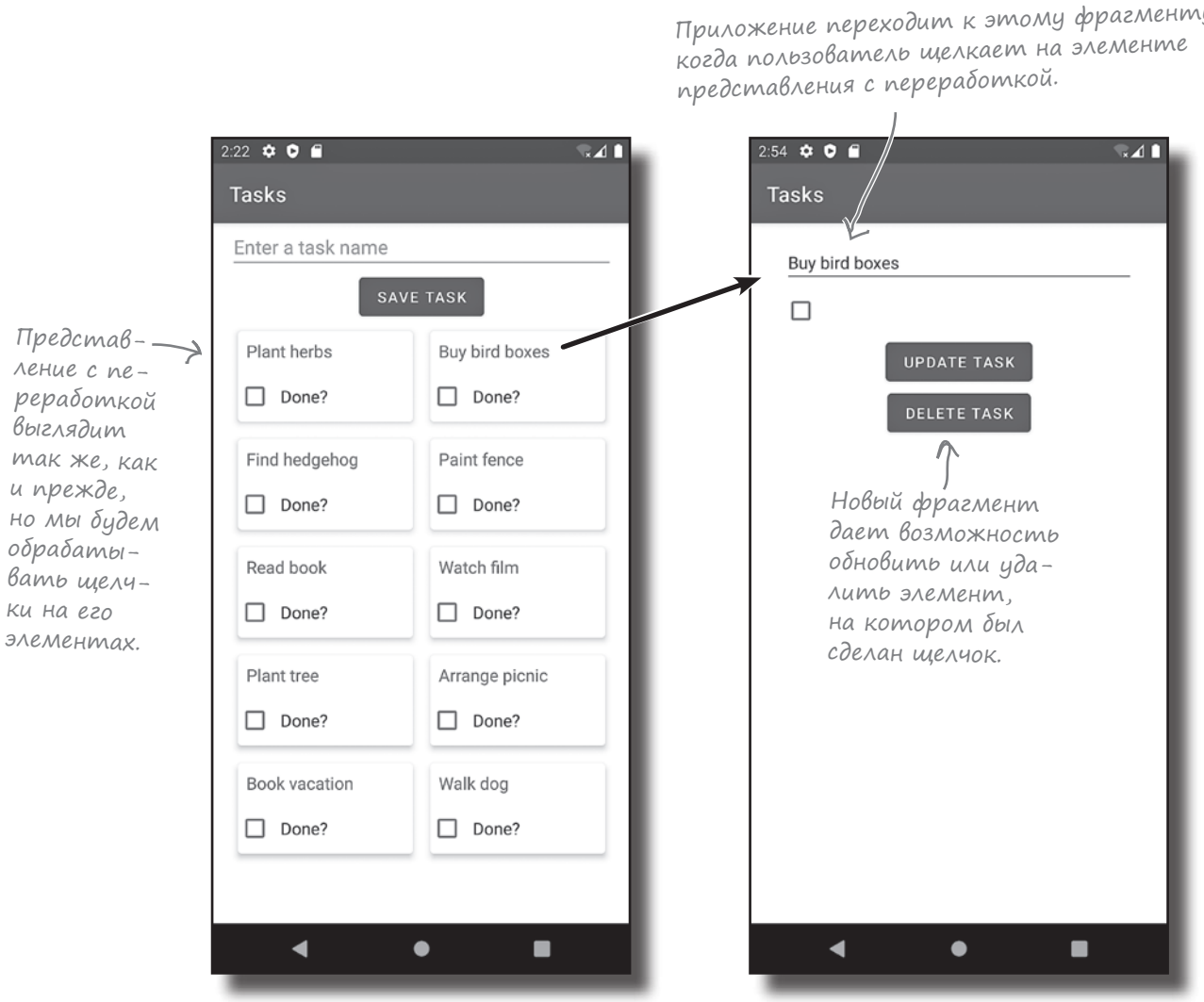
Некоторые приложения требуют, чтобы пользователь выбрал элемент из списка. В этой главе вы узнаете, **как сделать представления с переработкой центральной частью структуры вашего приложения**, для чего следует организовать **обработку щелчков на их элементах**. Вы увидите, как **реализовать навигацию в представлениях с переработкой**, чтобы приложение переходило к новому экрану каждый раз, когда пользователь щелкает на записи. Мы покажем, **как вывести дополнительную информацию** о выбранной записи и обновить ее в базе данных. К концу этой главы у вас будут все средства, необходимые для того, чтобы **преобразовать ваши великолепные замыслы в приложение вашей мечты**.

Навигация в представлениях с переработкой

В двух предыдущих главах вы узнали, как построить представление с переработкой, которое выводит список данных с возможностью прокрутки, а также научились использовать `DiffUtil` для повышения его эффективности. Тем не менее это еще не все.

Представления с переработкой становятся важнейшим компонентом многих приложений Android, потому что помимо отображения списков данных они могут использоваться для навигации в приложениях. Когда пользователь щелкает на элементе представления с переработкой, приложение может переходить к новому фрагменту с расширенной информацией об этой записи.

Чтобы понять, как работает навигация, мы изменим приложение Tasks, чтобы при щелчке на одной из задач в представлении с переработкой происходил переход к новому фрагменту. Этот фрагмент выводит информацию выбранной записи и предоставляет пользователю возможность обновить или удалить ее:



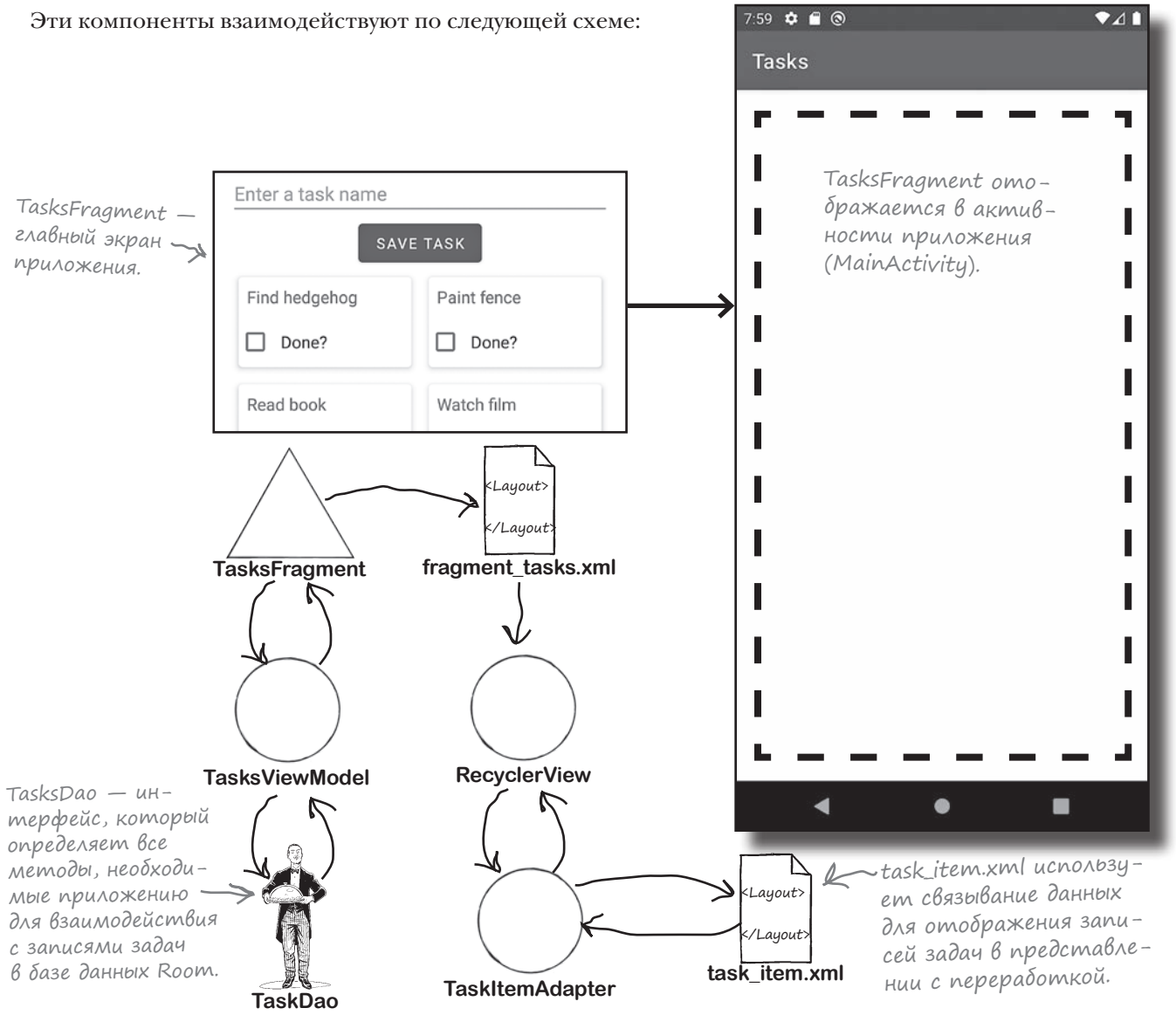
Структура приложения Tasks в текущей версии

Прежде чем разбираться в том, как нужно изменить приложение Tasks, вспомним его текущую структуру.

Приложение состоит из одной активности (`MainActivity`), которая отображает фрагмент с именем `TasksFragment`. Этот фрагмент представляет главный экран приложения, а его макет включает представление с переработкой, которое отображает сетку задач. Представление с переработкой использует адаптер с именем `TaskItemAdapter`, а размещение его элементов определяется файлом макета.

`TasksFragment` использует модель представления с именем `TasksViewModel`. Модель представления отвечает за бизнес-логику фрагмента и получает свои данные из базы данных Room через интерфейс `TaskDao`.

Эти компоненты взаимодействуют по следующей схеме:

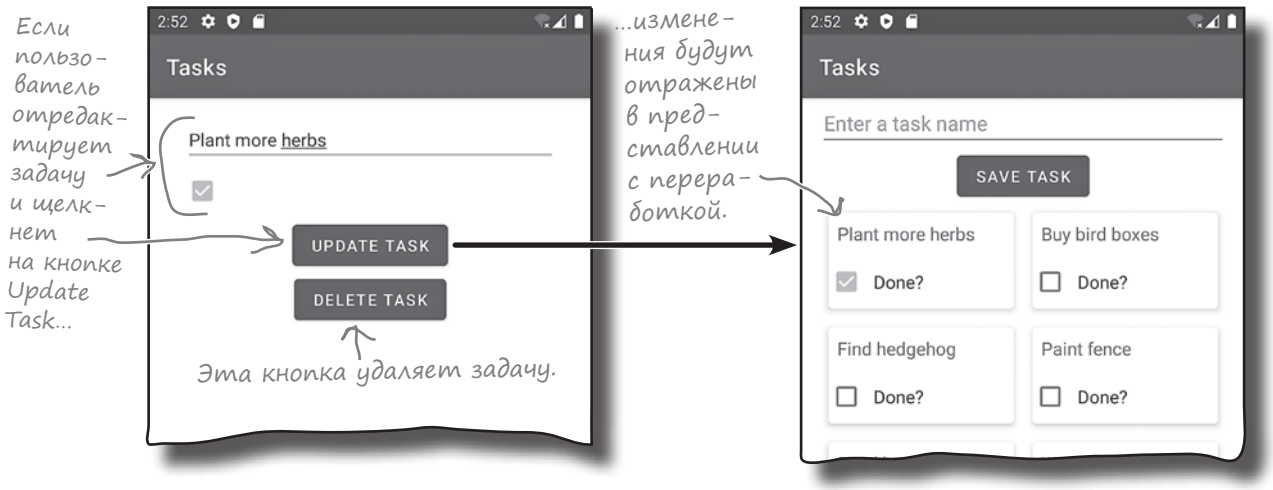


Использование представления с переработкой для перехода к новому фрагменту

Мы обновим приложение Tasks, чтобы по щелчку на задаче в представлении с переработкой отображался фрагмент с именем `EditTaskFragment`. Новый фрагмент будет выглядеть так:

Таким образом, `EditTaskFragment` включает текстовое поле и флажок, при помощи которых пользователь может редактировать задачу. В текстовом поле выводится имя задачи, а флажок содержит признак завершения.

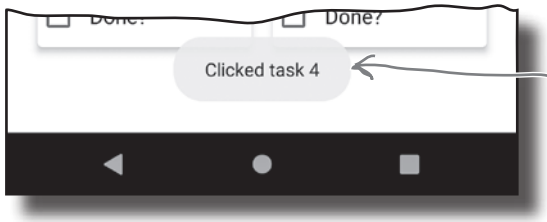
Фрагмент также включает кнопку Update Task, по которой обновляется запись в базе данных, и кнопку Delete Task для удаления записи. По щелчку на любой из этих кнопок приложение возвращается к фрагменту `TasksFragment`, который отображает обновленный список задач в своем представлении с переработкой:



Что мы собираемся сделать

Построение новой версии приложения состоит из трех фаз:

1. **Программирование реакции на щелчки.**
Мы обновим приложение, чтобы по щелчку на задаче в представлении с переработкой идентификатор этой задачи отображался на панели Toast.

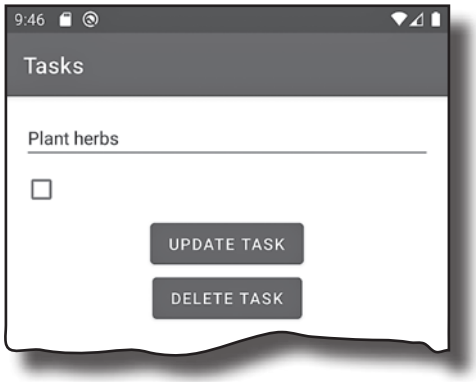


Если щелкнуть на задаче с идентификатором 4, появляется такое сообщение.

2. **Переход к EditTaskFragment по щелчку на задаче.**
Мы создадим `EditTaskFragment` и воспользуемся компонентом Navigation для перехода к этому фрагменту, когда пользователь щелкает на записи. Открывается новый фрагмент с идентификатором задачи.

Идентификатор задачи отображается в EditTaskFragment, чтобы пользователь видел, что идентификатор был успешно получен новым фрагментом.

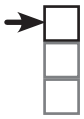
3. **Запись с описанием задачи отображается в EditTaskFragment, чтобы пользователь мог обновить или удалить запись.**
Для `EditTaskFragment` будет создана модель представления, которая использует интерфейс `TaskDao` для взаимодействия с базой данных.



Не забудьте!

Мы собираемся изменить приложение Tasks. Откройте проект этого приложения.

Начнем с программирования реакции представления на щелчки.



Реакция на щелчки
Навигация
Вывод записи

Реакция на щелчки

Первое изменение, которое будет внесено в приложение Tasks, — вывод панели Toast по щелчку на одном из элементов представления с переработкой.

Чтобы каждый элемент мог реагировать на щелчки, мы добавим `OnClickListener` к корневому представлению каждого элемента. Для этого метод `setOnClickListener()` каждого элемента будет вызываться сразу же после добавления данных каждого элемента в его макет.

Слушателей `OnClickListener` лучше всего добавлять в методе `bind()` объекта `TaskItemViewHolder`, так как именно здесь переменной связывания данных макета присваивается объект `Task`. Напомним, что метод `bind()` вызывается методом `onBindViewHolder()` объекта `TaskItemAdapter`, который вызывается каждый раз, когда представлению с переработкой потребуется отобразить данные элемента.

Ниже приведен код добавления `OnClickListener` в корневое представление макета каждого элемента; мы добавим его в *TaskItemAdapter.kt* через несколько страниц:

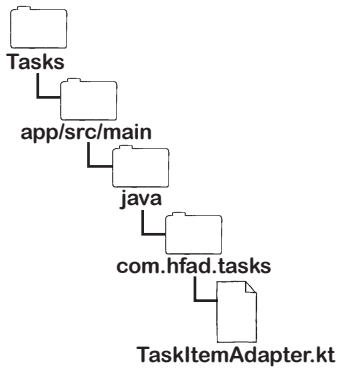
```
class TaskItemAdapter : ListAdapter<...>(TaskDiffItemCallback()) {
    ...
    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
    class TaskItemViewHolder(val binding: TaskItemBinding)
                            : RecyclerView.ViewHolder(binding.root) {
        ...
        fun bind(item: Task) {
            binding.task = item
            binding.root.setOnClickListener {
                //Код, выполняемый по щелчку на элементе
            }
        }
    }
}
```

Метод onBindViewHolder() вызывается каждый раз, когда представление с переработкой должно вывести данные элемента.

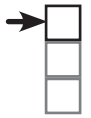
onBindViewHolder() вызывает метод bind() объекта TaskItemViewHolder и передает ему item.

Назначает переменную связывания данных для элемента.

Обеспечивает реакцию элемента на щелчки.



Итак, теперь вы знаете, как добавить слушатель `OnClickListener` для каждого элемента. Теперь нужно сделать так, чтобы по щелчку на элементе отображалась панель Toast.



Реакция на щелчки
Навигация
Вывод записи

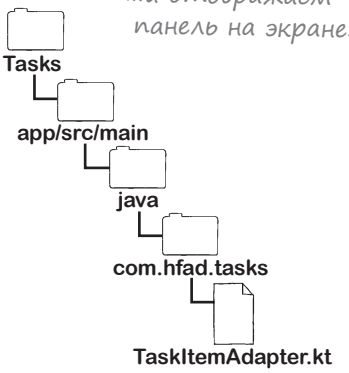
Где создать Toast?

Чтобы панель Toast отображалась каждый раз, когда пользователь щелкает на элементе, *можно* добавить следующий код (выделенный жирным шрифтом) в метод `setOnClickListener()` держателя представлений:

```
class TaskItemViewHolder(val binding: TaskItemBinding)
                        : RecyclerView.ViewHolder(binding.root) {
    ...
    fun bind(item: Task) {
        binding.task = item
        binding.root.setOnClickListener {
            Toast.makeText(binding.root.context, "Clicked task ${item.taskId}",
                                                 Toast.LENGTH_SHORT).show()
        }
    }
}
```

Задаем текст Toast...

...и отображаем панель на экране.



Однако такое решение будет означать, что код, описывающий поведение приложения, будет помещен в код держателя приложения. Последний отвечает за связывание данных с макетом каждого элемента, так что такое размещение вряд ли можно считать естественным.

Включение кода Toast в держатель представления также снижает гибкость кода держателя представления. Это будет означать, что каждый раз, когда пользователь щелкает на элементе, код будет только отображать панель Toast и использовать его в других местах не удастся.

Нет ли альтернативного решения?

Код Toast будет передаваться TasksFragment в лямбда-выражении

Альтернативное решение — определить код, необходимый каждому элементу для выполнения в `TasksFragment`, и передать его `TaskItemViewHolder` — через `TaskItemAdapter` — в лямбда-выражении. Такой подход означает, что тем, что происходит по щелчку, будет управлять *фрагмент*, а не держатель представления.

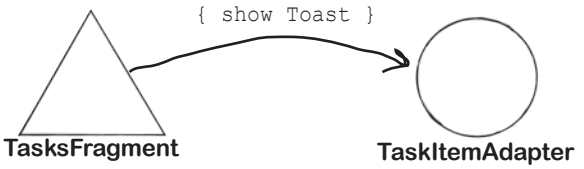
Прежде чем рассматривать код, разберемся в том, как он работает.

Реакция на щелчки
Навигация
Вывод записи

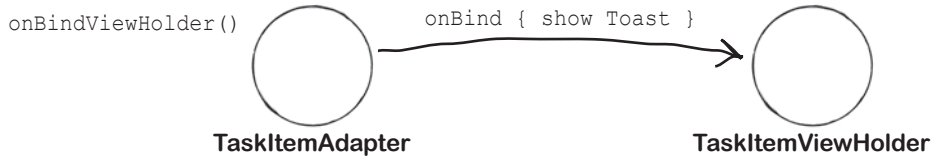
Как работает код

При выполнении кода происходят следующие события:

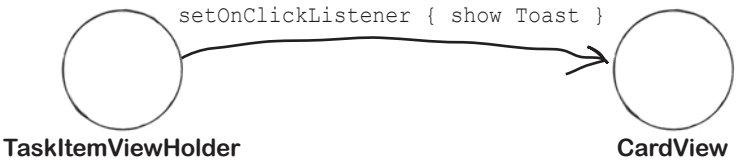
1. **TasksFragment передает лямбда-выражение конструктору TaskItemAdapter.**
   Лямбда-выражение включает код для отображения панели Toast.



2. **При вызове метода onBindViewHolder() объекта TaskItemAdapter он вызывает метод bind() объекта TaskItemViewHolder и передает ему лямбда-выражение.**



3. **TaskItemViewHolder добавляет лямбда-выражение в слушатель OnClickListener каждого элемента.**
   Когда пользователь щелкает на каждом элементе (`CardView`), выполняется лямбда-выражение и отображается панель Toast.



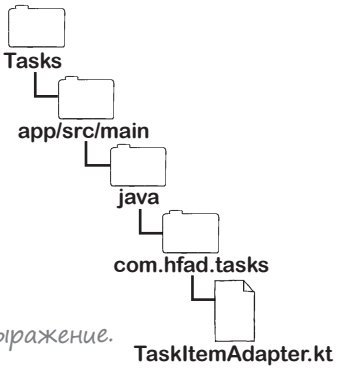
Чтобы реализовать эту схему, необходимо обновить код `TasksFragment`, `TaskItemAdapter` и `TaskItemViewHolder`. Начнем с обновления `TaskItemAdapter` и `TaskItemViewHolder`, чтобы адаптер мог получить лямбда и передать его держателю представления.

Код для решения этой задачи приведен на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

Полный код TaskItemAdapter.kt

Ниже приведен код `TaskItemAdapter` и `TaskItemViewHolder`; обновите файл *TaskItemAdapter.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.tasks

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.hfad.tasks.databinding.TaskItemBinding

class TaskItemAdapter(val clickListener: (taskId: Long) -> Unit)
        : ListAdapter<Task, TaskItemAdapter.TaskItemViewHolder>(TaskDiffItemCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                            : TaskItemViewHolder = TaskItemViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: TaskItemViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item, clickListener)
    }

    class TaskItemViewHolder(val binding: TaskItemBinding)
                            : RecyclerView.ViewHolder(binding.root) {

        companion object {
            fun inflateFrom(parent: ViewGroup): TaskItemViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val binding = TaskItemBinding.inflate(layoutInflater, parent, false)
                return TaskItemViewHolder(binding)
            }
        }

        fun bind(item: Task, clickListener: (taskId: Long) -> Unit) {
            binding.task = item
            binding.root.setOnClickListener { clickListener(item.taskId) }
        }
    }
}
```

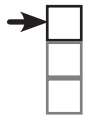
TaskItemAdapter получает лямбда-выражение.

Лямбда-выражение передается методу bind() объекта TaskItemViewHolder.

Метод bind() получает лямбда-выражение.

Элемент должен реагировать на щелчки.

По щелчку на элементе выполняется лямбда-выражение.



Передача лямбда-выражения TaskItemAdapter

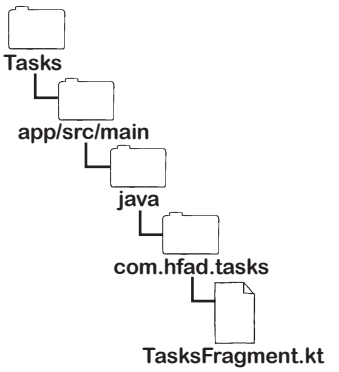
Итак, когда конструктор `TaskItemAdapter` получает параметр с лямбда-выражением, мы должны передать его в коде `TasksFragment` при создании фрагмента.

Как вы помните, `TasksFragment` создает объект `TaskItemAdapter` в методе `onCreateView()`, который затем назначается представлению с переработкой следующим образом:

```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    ...
    val adapter = TaskItemAdapter()
    binding.tasksList.adapter = adapter
    ...
}
```

Создаем адаптер...

...и назначаем его представлению с переработкой.



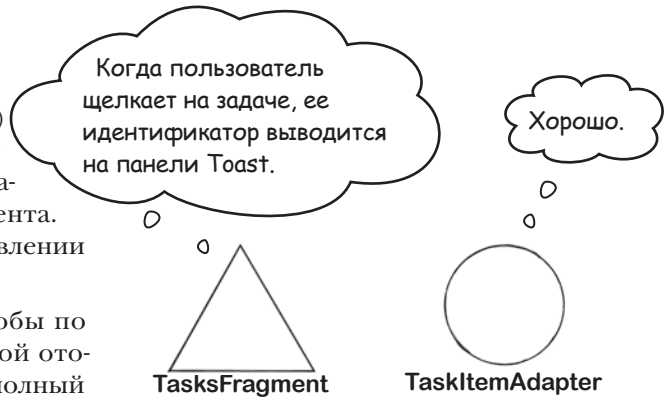
Так как при каждом щелчке на представлении с переработкой должна отображаться панель Toast, код можно обновить, чтобы конструктору `TaskItemAdapter` передавалось следующее лямбда-выражение:

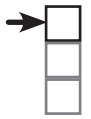
```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    ...
    val adapter = TaskItemAdapter { taskId ->
        Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
    }
    binding.tasksList.adapter = adapter
    ...
}
```

Лямбда-выражение передается объекту TaskItemAdapter, который по щелчку отображает панель Toast.

Адаптер передает лямбда-выражение методу `bind()` объекта `TaskItemViewHolder`, который использует лямбда-выражение в коде `OnClickListener`, назначаемом в корневом представлении каждого элемента. Когда пользователь щелкает на элементе в представлении с переработкой, выполняется лямбда-выражение.

Теперь вы знаете все необходимое для того, чтобы по щелчку на элементе представления с переработкой отображалась панель Toast. Посмотрим, как выглядит полный код `TasksFragment`.

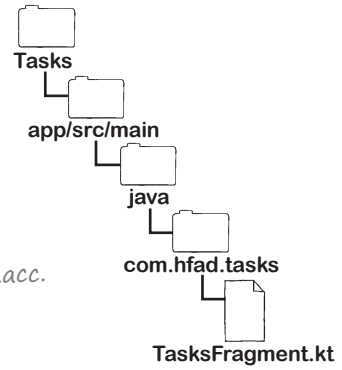




Реакция на щелчки
Навигация
Вывод записи

Полный код TasksFragment.kt

Ниже приведен обновленный код `TasksFragment`; убедитесь в том, что код *TasksFragment.kt* включает все изменения, выделенные жирным шрифтом:




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
```

Импортируйте этот класс.

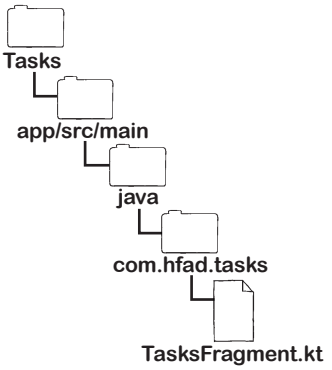
Продолжение на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

Код TasksFragment.kt (продолжение)

```
        val adapter = TaskItemAdapter() { taskId ->
            Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
        }
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.submitList(it)
            }
        })

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Лямбда-выражение передается конструктору TaskItemAdapter.



Посмотрим, что происходит при выполнении этого кода.

Часто Задаваемые Вопросы

В: В коде `TaskItemAdapter` мы определили для аргумента `clickListener` тип `(taskId: Long) -> Unit`. Напомните, что это означает?

О: Это означает, что в аргументе может передаваться блок кода, называемый лямбда-выражением. Код получает аргумент типа `Long` с именем `taskId` и возвращает `Unit`.

В: Понятно. Выходит, это способ передачи кода методу?

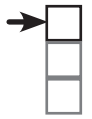
О: Да. Это означает, что `TasksFragment` может указать, какой код должен выполняться, вместо того чтобы жестко фиксировать его в коде `TaskItemAdapter`.

В: Включение его в код `TaskItemAdapter` настолько нежелательно?

О: Это снизит гибкость кода адаптера, а у компонентов приложения не будет настолько четкого разделения обязанностей.

В: Кажется, я видел и другие способы реагирования на щелчки в представлениях с переработкой. Все они неправильные?

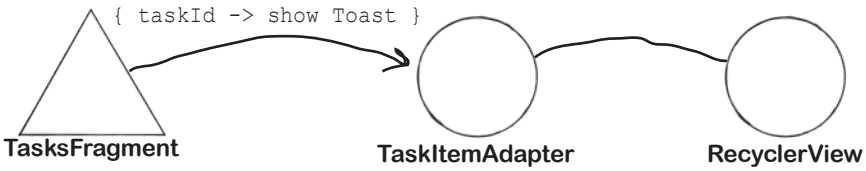
О: Есть много разных способов реагирования на щелчки со стороны представления с переработкой. Мы выбрали этот подход, потому что он чрезвычайно гибок и проще других в изучении. Мы рекомендуем как следует освоить способ, использованный в этой главе, прежде чем браться за исследование других способов.



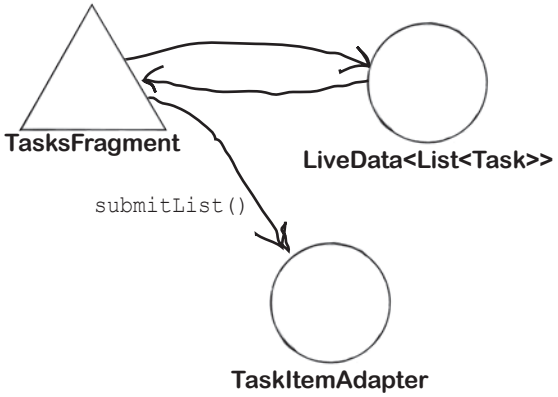
Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

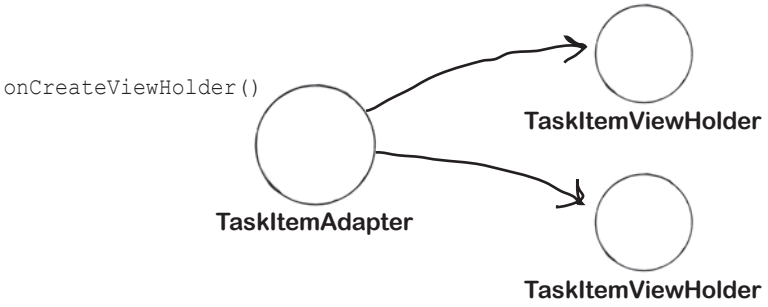
1. **TasksFragment создает объект TaskItemAdapter и назначает его адаптером для представления с переработкой.**
Фрагмент передает лямбда-выражение (с именем `clickListener`) адаптеру, чтобы тот отображал панель Toast при выполнении.

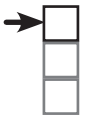


2. **TasksFragment передает TaskItemAdapter список List<Task>.**
`List<Task>` содержит актуальный список записей из базы данных.



3. **Метод onCreateViewHolder() объекта TaskItemAdapter вызывается для каждого элемента, который должен отображаться в представлении с переработкой.**
В результате создается набор объектов `TaskItemViewHolder`.

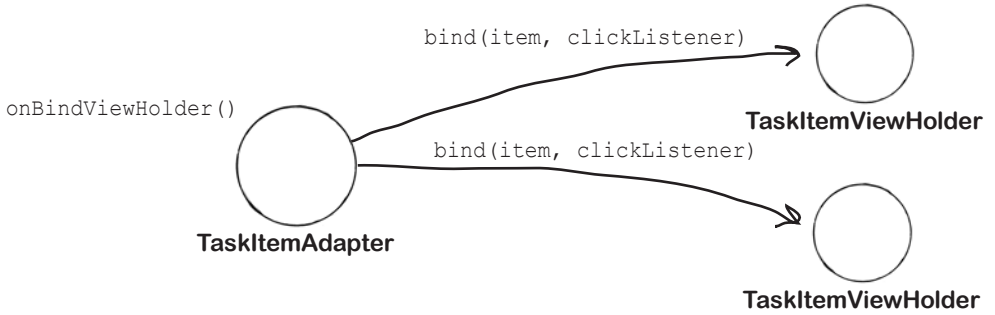




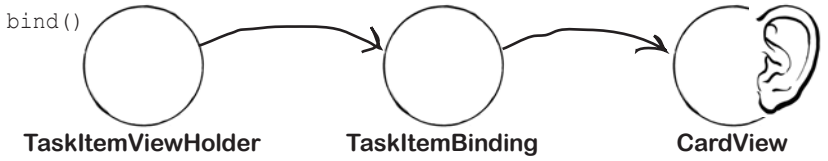
Реакция на щелчки
Навигация
Вывод записи

История продолжается

4 **Метод onBindViewHolder() объекта TaskItemAdapter вызывается для каждого TaskItemViewHolder.**
В результате вызывается метод `bind()` объекта `TaskItemViewHolder`, которому передается элемент, на котором щелкнул пользователь, и лямбда-выражение `clickListener`.



5 **Метод bind() объекта TaskItemViewHolder добавляет слушателя OnClickListener к корневому представлению макета каждого держателя представления.**
В данном примере корневым представлением является `CardView`.



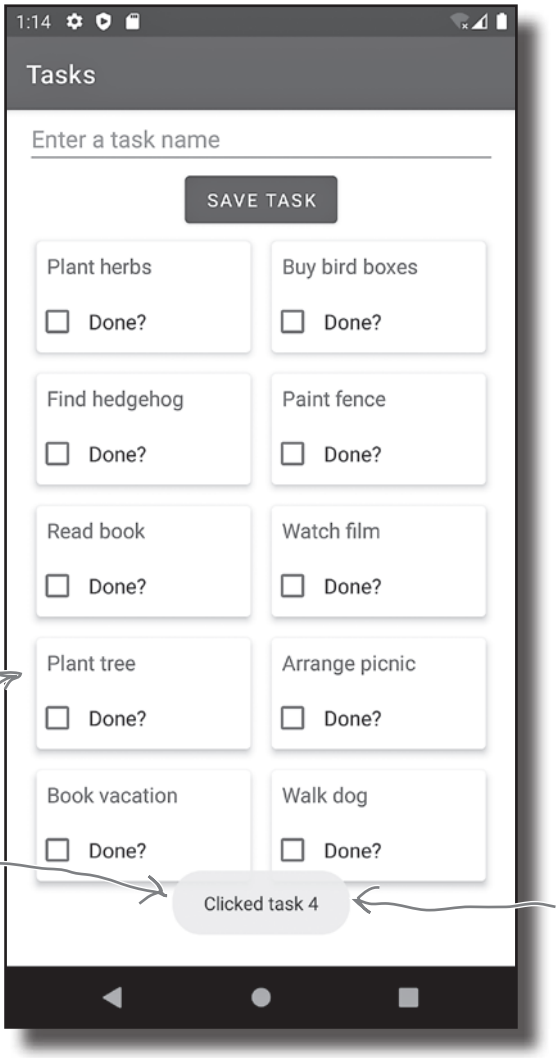
6 **Когда пользователь щелкает на элементе в представлении с переработкой, слушатель OnClickListener регистрирует щелчок.**
Он выполняет лямбда-выражение `clickListener`, которое отображает панель Toast.

Проведем тест-драйв приложения.

Тест-драйв

- [x] **Реакция на щелчки**
- [ ] **Навигация**
- [ ] **Вывод записей**

При запуске приложения `TasksFragment` отображает в представлении с переработкой сетку карточек, как и прежде. Если щелкнуть на одной из задач, представление выводит ее идентификатор на панели Toast.

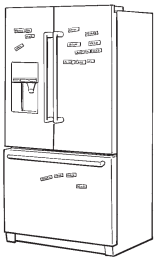


Если щелкнуть на задаче («Plant tree»), появляется панель Toast.

В тексте Toast содержится идентификатор задачи. В данном случае это значение 4 — идентификатор записи в базе данных.

Вы узнали, как сделать так, чтобы элементы представления с переработкой реагировали на события щелчков. Мы воспользуемся новыми знаниями, чтобы приложение переходило к новому фрагменту по щелчку на элементе.

Но сначала проверьте свои силы на следующем упражнении.



Развлечения с магнитами

Приложение Bits and Pizzas включает представление с переработкой, которое использует макет *pizza_item.xml* для отображения объектов `Pizza`. Класс данных `Pizza` выглядит примерно так:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

Представление с переработкой использует адаптер с именем `PizzaAdapter`, код которого приведен ниже. Удастся ли вам завершить код адаптера, чтобы по щелчку на одном из его элементов выполнялось лямбда-выражение, переданное адаптеру в конструкторе?

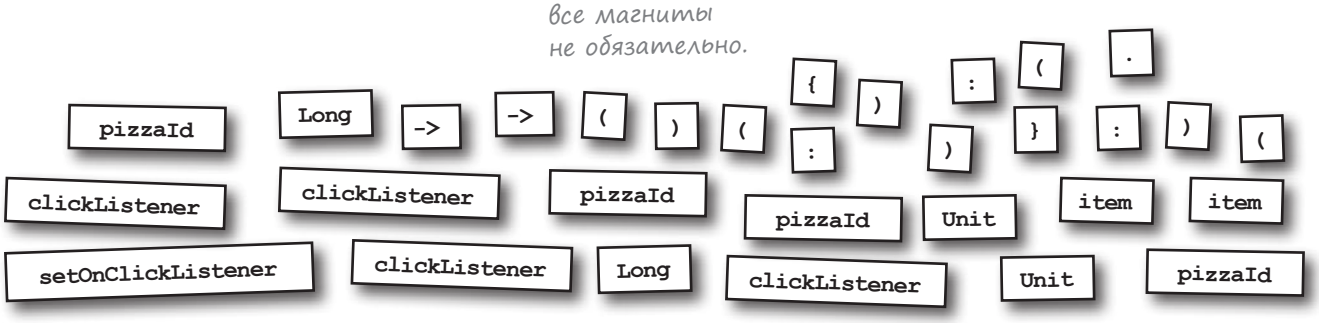
Подсказка: лямбда-выражение должно получать аргумент `pizzaId` и возвращать `Unit`.

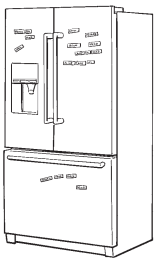
```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.hfad.bitsandpizzas.databinding.PizzaItemBinding

class PizzaAdapter(val clickListener: ..... )
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)

        holder.bind(item, ..... )
    }
```

Продолжение на следующей странице.

```
class PizzaViewHolder(val binding: PizzaItemBinding)
                                    : RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)

            val binding = PizzaItemBinding.inflate(layoutInflater, parent, false)
            return PizzaViewHolder(binding)
        }
    }

    fun bind(item: Pizza,

        ..........) {

        binding.pizza = item

        binding.root. ..........

            ..........

        ..........
    }
  }
}
```

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Приложение Bits and Pizzas включает представление с переработкой, которое использует макет *pizza_item.xml* для отображения объектов `Pizza`. Класс данных `Pizza` выглядит примерно так:

```
package com.hfad.bitsandpizzas

data class Pizza(
    var pizzaId: Long = 0L,
    var pizzaName: String = "",
    var pizzaDescription: String = "",
    var pizzaImageId: Int = 0
)
```

Представление с переработкой использует адаптер с именем `PizzaAdapter`, код которого приведен ниже. Удастся ли вам завершить код адаптера, чтобы по щелчку на одном из его элементов выполнялось лямбда-выражение, переданное адаптеру в конструкторе?

Подсказка: лямбда-выражение должно получать аргумент `pizzaId` и возвращать `Unit`.

```
package com.hfad.bitsandpizzas

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.hfad.bitsandpizzas.databinding.PizzaItemBinding

class PizzaAdapter(val clickListener: ( pizzaId : Long ) -> Unit)
    : ListAdapter<Pizza, PizzaAdapter.PizzaViewHolder>(PizzaDiffItemCallback) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
                        : PizzaViewHolder = PizzaViewHolder.inflateFrom(parent)

    override fun onBindViewHolder(holder: PizzaViewHolder, position: Int) {
        val item = getItem(position)

        holder.bind(item, clickListener)
    }
```

Указывает, что PizzaAdapter получает в конструкторе лямбда-выражение. Лямбда-выражение получает Long и возвращает Unit.

Методу bind() объекта PizzaViewHolder передается лямбда-выражение.

Продолжение на следующей странице.

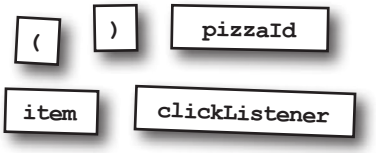
```
class PizzaViewHolder(val binding: PizzaItemBinding)
                                    : RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun inflateFrom(parent: ViewGroup): PizzaViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)

            val binding = PizzaItemBinding.inflate(layoutInflater, parent, false)
            return PizzaViewHolder(binding)
        }
    }

    fun bind(item: Pizza,
             clickListener : ( pizzaId : Long ) -> Unit ) {

        binding.pizza = item

        binding.root. setOnClickListener {
            clickListener ( item . pizzaId )
        }
    }
}
```

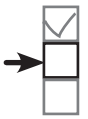
Метод bind() получает в аргументе лямбда-выражение.

Обеспечивает реакцию на щелчки.

Лямбда-выражение выполняется по щелчку на элементе.

Эти магниты остались неиспользованными.

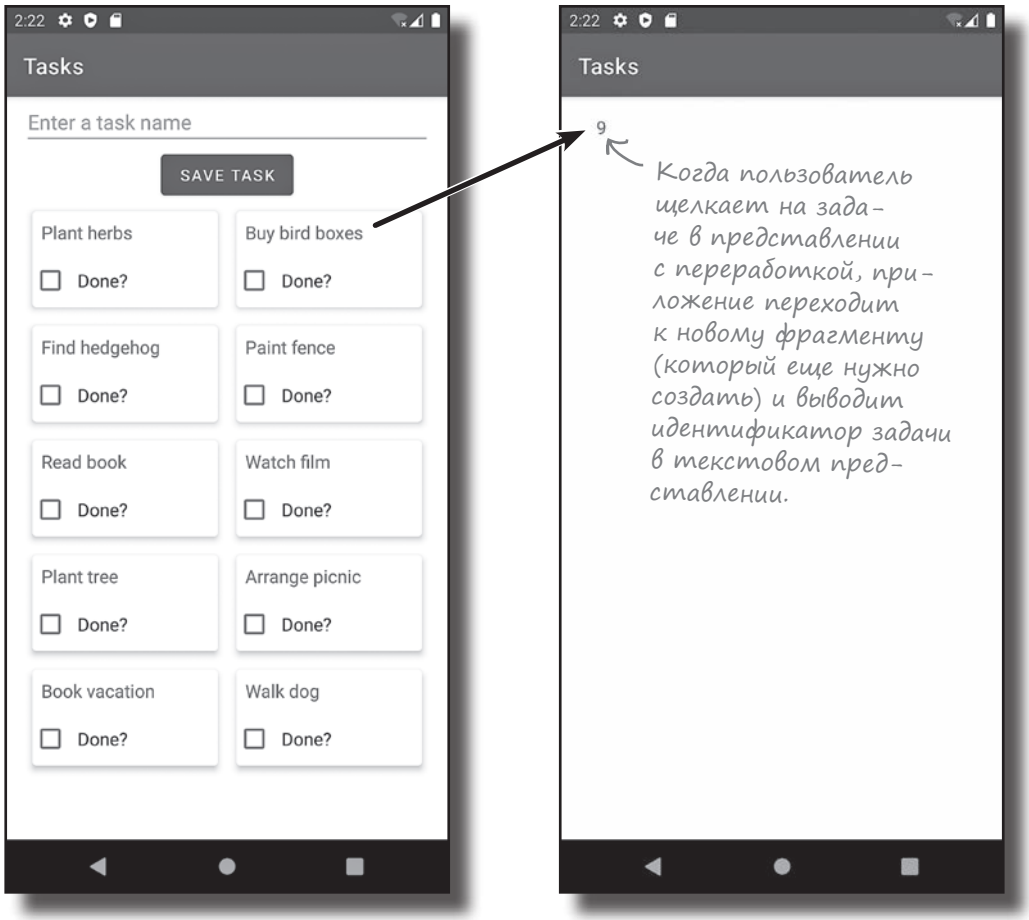




Представление с переработкой должно использоваться для перехода к новому фрагменту

К настоящему моменту вы узнали, как заставить представление с переработкой реагировать на события щелчков. Например, когда пользователь щелкает на задаче в представлении с переработкой приложения Tasks, на экране появляется сообщение.

На следующем этапе мы изменим это поведение, чтобы по щелчку на элементе приложение переходило к новому фрагменту (который мы создадим) и выводило идентификатор задачи. Новая версия приложения должна выглядеть так:



Чтобы эта схема заработала, мы воспользуемся компонентом Navigation для перехода к новому фрагменту и плагином Safe Args для передачи фрагменту идентификатора задачи. Это означает, что для включения этих компонентов придется обновить проект и файлы *build.gradle* приложения.

Реакция на щелчки
Навигация
Вывод записи

Сначала обновляется файл build.gradle проекта...

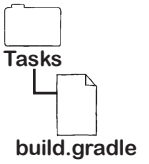
Начнем с обновления файла *build.gradle* проекта. В нем должна содержаться информация о том, какая версия компонента Navigation используется в приложении, а также путь к классам для плагина Safe Args.

Для этого откройте файл *Tasks/build.gradle* и добавьте следующие строки (выделенные жирным шрифтом) в соответствующие разделы:

```
buildscript {
    ext.nav_version = "2.3.5"
    ...
    dependencies {
        ...
        classpath "androidx.navigation:navigation-safe-args-gradle-plug-in:$nav_version"
    }
}
```

Версия компонента Navigation сохраняется в переменной, чтобы мы могли быть уверены в том, что используется одна версия компонента Navigation и плагина Safe Args.

Добавление пути к классам для Safe Args.



...а затем файл build.gradle приложения

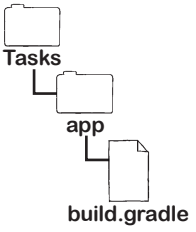
Также необходимо добавить плагин Safe Args в файл *build.gradle* приложения вместе с зависимостью для компонента Navigation.

Откройте файл *Tasks/app/build.gradle* и добавьте следующие строки (выделенные жирным шрифтом) в соответствующие разделы:

```
plugins {
    ...
    id 'androidx.navigation.safeargs.kotlin'
}
...
dependencies {
    ...
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    ...
}
```

Включаем плагин Safe Args.

Добавляем компонент Navigation.



После внесения изменений щелкните на ссылке Sync Now, чтобы синхронизировать их с остальными частями проекта.

После добавления компонента Navigation и плагина Safe Args необходимо создать новый фрагмент, к которому будет переходить приложение.

Реакция на щелчки
Навигация
Вывод записи

Создание EditTaskFragment...

Мы создадим новый фрагмент с именем `EditTaskFragment`, к которому будет переходить приложение по щелчку на элементе представления с переработкой.

Выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите команду File→New→Fragment→Fragment (Blank). Введите имя фрагмента «EditTaskFragment», затем имя его макета «fragment_edit_task» и убедитесь в том, что выбран язык Kotlin.

Мы займемся обновлением кода `EditTaskFragment` и его макета через несколько страниц. А пока создадим граф навигации, который сообщает, как должны происходить переходы между фрагментами приложения.

Это фрагмент EditTaskFragment. В первом воплощении он будет просто сообщать, на какой задаче щелкнул пользователь.

...и создание графа навигации

Граф навигации добавляется в проект по той же схеме, как в других, созданных ранее приложениях.

Выделите папку *Tasks/app/src/main/res* на панели проекта, затем выберите команду File→New→Android Resource File. Введите имя файла «nav_graph», выберите тип ресурса «Navigation» и щелкните на кнопке OK. IDE создает граф навигации с именем *nav_graph.xml*.

Navigation Graph

Граф навигации должен описывать, как пользователь переходит между `TasksFragment` и `EditTaskFragment`. Схема навигации в нашем приложении выглядит так:

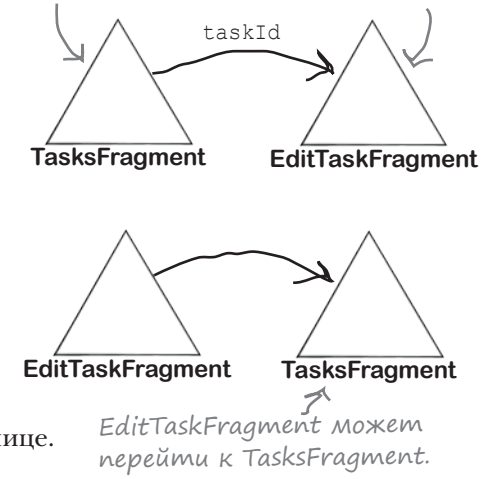
1. **Приложение отображает TasksFragment.**
   Это первый фрагмент, который видит пользователь, поэтому он становится стартовой целью в графе навигации.

2. **Когда пользователь щелкает на элементе в представлении с переработкой TasksFragment, приложение переходит к EditTaskFragment.**
   `TasksFragment` передает `EditTaskFragment` параметр типа `Long`, содержащий идентификатор задачи для элемента, на котором щелкнул пользователь.

3. **Когда пользователь щелкает на кнопке в EditTaskFragment (мы добавим ее во фрагмент позднее в этой главе), приложение возвращается к TasksFragment.**

Навигация начинается с TasksFragment.

TasksFragment может перейти к EditTaskFragment.



EditTaskFragment может перейти к TasksFragment.

Полная реализация этой схемы приведена на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

Обновление графа навигации

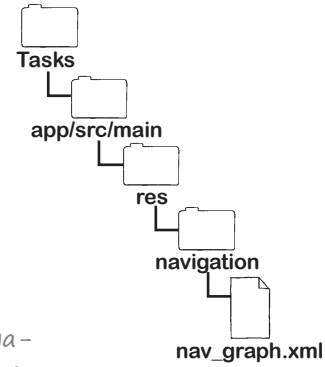
Ниже приведен полный код графа навигации; обновите файл *nav_graph.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/tasksFragment">
    <fragment
        android:id="@+id/tasksFragment"
        android:name="com.hfad.tasks.TasksFragment"
        android:label="fragment_tasks"
        tools:layout="@layout/fragment_tasks" >
        <action
            android:id="@+id/action_tasksFragment_to_editTaskFragment"
            app:destination="@id/editTaskFragment" />
    </fragment>
    <fragment
        android:id="@+id/editTaskFragment"
        android:name="com.hfad.tasks.EditTaskFragment"
        android:label="fragment_edit_task"
        tools:layout="@layout/fragment_edit_task" >
        <argument
            android:name="taskId"
            app:argType="long" />
        <action
            android:id="@+id/action_editTaskFragment_to_tasksFragment"
            app:destination="@id/tasksFragment"
            app:popUpTo="@id/tasksFragment"
            app:popUpToInclusive="true" />
    </fragment>
</navigation>
```

Файл обязательно должен содержать эту строку.

Приложение начинает с TasksFragment.

Это действие позволяет TasksFragment перейти к EditTaskFragment.



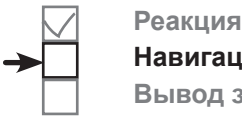
EditTaskFragment получает аргумент типа Long.

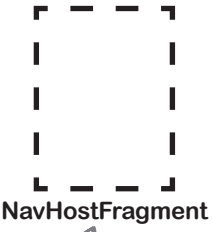
Действие позволяет EditTaskFragment перейти к TasksFragment.

Фрагменты извлекаются из стека до TasksFragment включительно.

На следующем шаге нужно связать граф навигации с `MainActivity`, чтобы активность отображала каждый фрагмент при переходе к нему.

Добавление NavHostFragment в макет MainActivity





Хост навигации NavHostFragment будет добавлен в представление FragmentContainerView макета activity_main.xml. Затем мы прикажем ему использовать только что созданный граф навигации.

Чтобы связать только что созданный граф навигации с `MainActivity`, необходимо добавить хост навигации в макет и отдать ему команду использовать *nav_graph.xml* в качестве графа навигации. Это позволит `MainActivity` отображать правильные фрагменты при перемещении пользователя по приложению.

Хост навигации добавляется в макет тем же способом, который применялся в предыдущих главах: добавлением `NavHostFragment` в представление `FragmentContainerView` файла *activity_main.xml*. Обновите файл *activity_main.xml* (изменения выделены жирным шрифтом):

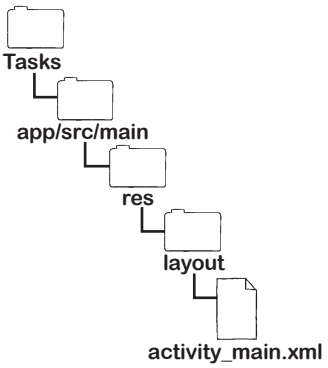
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    ~~android:id="@+id/fragment_container_view"~~
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    ~~android:name="com.hfad.tasks.TasksFragment"~~
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true"
    tools:context=".MainActivity" />
```

Добавляем дополнительное пространство имен.

Изменяем идентификатор.

Обновляем атрибут name, чтобы он ссылался на NavHost Fragment.

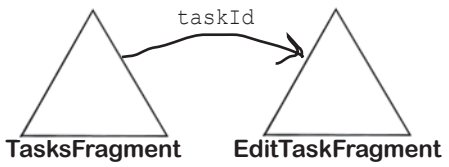
Добавляем эти атрибуты.



И это весь код, который необходимо изменить в макете `MainActivity`. Теперь нужно сделать так, чтобы фрагмент `TasksFragment` переходил к `EditTaskFragment`, когда пользователь щелкает на одном из элементов в его представлении с переработкой.

Переход от TasksFragment к EditTaskFragment

Каждый раз, когда пользователь щелкает на элементе представления с переработкой фрагмента `TasksFragment`, приложение должно перейти к `EditTaskFragment` и передать идентификатор задачи, на которой щелкнул пользователь.

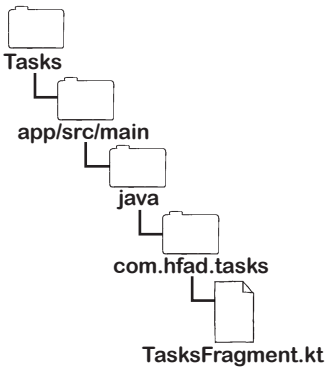


Одно из возможных решений — обновление лямбда-выражения, которое `TasksFragment` передает своему адаптеру `TaskItemAdapter`, чтобы оно включало весь необходимый код навигации:

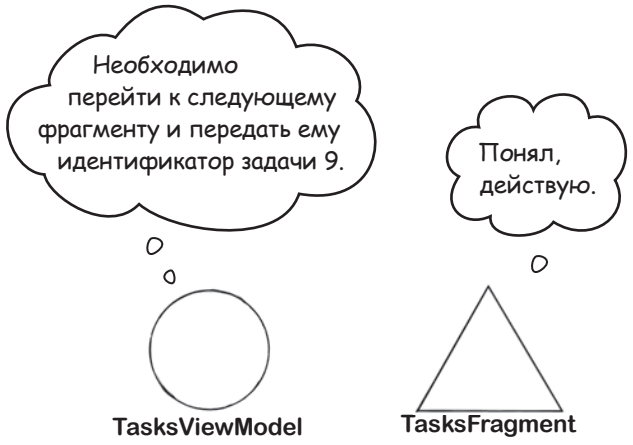
```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    ...
    val adapter = TaskItemAdapter { taskId ->
        val action = TasksFragmentDirections
                        .actionTasksFragmentToEditTaskFragment(taskId)
        this.findNavController().navigate(action)
    }
    binding.tasksList.adapter = adapter
    ...
}
```

При выполнении это лямбда-выражение передает идентификатор задачи EditTaskFragment и переходит к этому фрагменту.

Как видите, класс `TasksFragmentDirections` (сгенерированный плагином Safe Args) используется для передачи идентификатора задачи элемента фрагменту `EditTaskFragment` и перехода к этому фрагменту.



Однако такой подход означает, что бизнес-логика (принятие решения о том, когда фрагмент `TasksFragment` должен перейти к `EditTaskFragment`) размещается в коде фрагмента — вместо добавления ее к `TasksViewModel`. Как вы узнали в главе 13, подобные решения должны приниматься в коде модели представления, а не в коде фрагмента.



Для решения этой проблемы мы будем использовать подход, аналогичный тому, который использовался в приложении Guessing Game главы 13. Мы добавим в `TasksViewModel` новое свойство Live Data. В нем будет храниться идентификатор задачи, на которой щелкнул пользователь. Когда значение этого свойства изменяется, `TasksFragment` реагирует переходом к `EditTaskFragment` с передачей идентификатора.

Реакция на щелчки
Навигация
Вывод записи

Добавление нового свойства в TasksViewModel

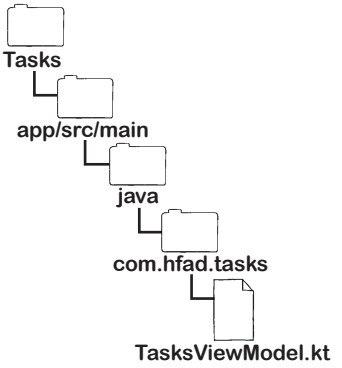
Начнем с добавления в `TasksViewModel` нового свойства Live Data для идентификатора задачи, который должен передаваться из `TasksFragment` фрагменту `EditTaskFragment`. Назовем это свойство `navigateToTask`; оно будет определяться следующим кодом (он будет добавлен в *TasksViewModel.kt* на следующей странице):

```
class TasksViewModel(val dao: TaskDao) : ViewModel() {
    ...
    private val _navigateToTask = MutableLiveData<Long?>()
    val navigateToTask: LiveData<Long?>
        get() = _navigateToTask
    ...
}
```

Добавляем приватное резервное свойство с именем _navigateToTask.

navigateToTask предоставляет другим классам доступ только для чтения к значению своего резервного свойства.

Как видно из листинга, свойство `navigateToTask` использует изменяемое резервное свойство с модификатором `private`; это означает, что значение этого свойства может задать только `TasksViewModel`. Тем самым свойство защищается от нежелательных обновлений со стороны других классов.



Добавление методов для обновления нового свойства

Каждый раз, когда пользователь щелкает на задаче в представлении с переработкой, фрагмент `TasksFragment` должен перейти к фрагменту `EditTaskFragment` и передать ему идентификатор задачи.

Для этого мы добавим в `TasksViewModel` два метода — `onTaskClicked()` и `onTaskNavigated()`, — которые будут использоваться для присваивания значения резервного свойства `navigateToTask`. Метод `onTaskClicked()` присваивает свойству идентификатор задачи, а метод `onTaskNavigated()` возвращает ему значение `null`.

Код этих двух методов:

```
fun onTaskClicked(taskId: Long) {
    _navigateToTask.value = taskId
}

fun onTaskNavigated() {
    _navigateToTask.value = null
}
```

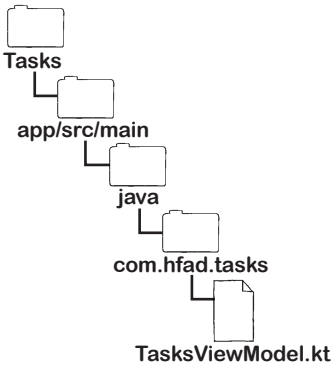
Присваивает _navigateToTask идентификатор задачи, который должен быть передан TasksFragment фрагменту EditTaskFragment.

И это все изменения, которые необходимо внести в `TasksViewModel`. Полный код будет приведен на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

Полный код TasksViewModel.kt

Ниже приведен обновленный код `TasksViewModel`; убедитесь в том, что код *TasksViewModel.kt* содержит все изменения, выделенные жирным шрифтом:




```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class TasksViewModel(val dao: TaskDao) : ViewModel() {
    var newTaskName = ""
    val tasks = dao.getAll()
    private val _navigateToTask = MutableLiveData<Long?>()
    val navigateToTask: LiveData<Long?>
        get() = _navigateToTask

    fun addTask() {
        viewModelScope.launch {
            val task = Task()
            task.taskName = newTaskName
            dao.insert(task)
        }
    }

    fun onTaskClicked(taskId: Long) {
        _navigateToTask.value = taskId
    }

    fun onTaskNavigated() {
        _navigateToTask.value = null
    }
}
```

Импортируйте эти классы.

Добавляем свойство navigateToTask вместе с его резервным свойством _navigateToTask.

Добавляем эти два метода.

Код `TasksViewModel` обновлен. Давайте посмотрим, какие изменения необходимо внести в код `TasksFragment`.

Переход от TasksFragment к EditTaskFragment

Необходимо обновить код `TasksFragment`, чтобы по щелчку на задаче происходил переход к `EditTaskFragment` с передачей идентификатора задачи.

Для этого мы вызовем метод `onTaskClicked()` объекта `TasksViewModel`, когда пользователь щелкает на задаче, и выполним переход к `EditTaskFragment`, когда свойство `navigateToTask` обновляется новым идентификатором задачи.

Вызов onTaskClicked() по щелчку на задаче

Чтобы вызвать метод `onTaskClicked()`, мы передадим приведенное ниже лямбда-выражение (выделенное жирным шрифтом) конструктору `TaskItemAdapter`:

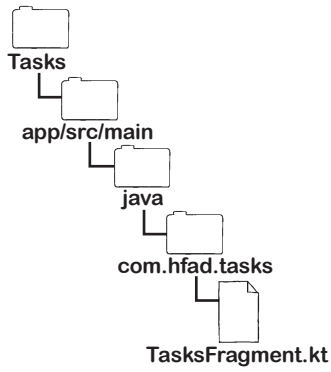
```
val adapter = TaskItemAdapter { taskId ->
    viewModel.onTaskClicked(taskId)
}
binding.tasksList.adapter = adapter
```

Выполняется, когда пользователь щелкает на задаче.

Каждый раз, когда пользователь щелкает на задаче, выполняется лямбда-выражение; оно вызывает метод `onTaskClicked()` объекта `TasksViewModel`, присваивая `navigateToTask` идентификатор задачи.

Переход к EditTaskFragment при обновлении navigateToTask

Чтобы фрагмент `TasksFragment` переходил к `EditTaskFragment`, мы отдадим ему команду наблюдать за свойством `navigateToTask` объекта `TaskViewModel`. Когда этому свойству присваивается идентификатор задачи (`Long`), фрагмент переходит к `EditTaskFragment` и передает ему идентификатор. После этого свойству `navigateToTask` снова возвращается значение null вызовом метода `onTaskNavigated()` модели представления. Для этого используется следующий код:



Наблюдение за свойством navigateToTask.

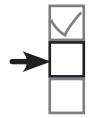
```
viewModel.navigateToTask.observe(viewLifecycleOwner, Observer { taskId ->
    taskId?.let {
        val action = TasksFragmentDirections
                .actionTasksFragmentToEditTaskFragment(taskId)
        this.findNavController().navigate(action)
        viewModel.onTaskNavigated()
    }
})
```

Блок let будет выполнять-ся только в том случае, если значение taskId не равно null.

Переходим к EditTaskFragment с передачей taskId.

Вызывается метод onTaskNavigated(), чтобы свойству navigateToTask было присвоено значение null.

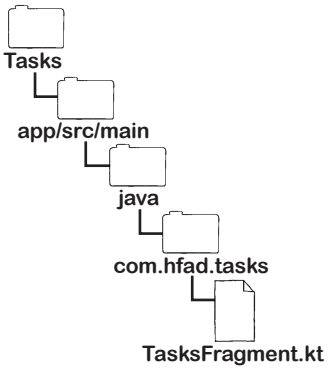
Обновим код `TasksFragment`.



Реакция на щелчки
Навигация
Вывод записи

Полный код TasksFragment.kt

Ниже приведен обновленный код `TasksFragment`; убедитесь в том, что он включает все изменения (выделенные жирным шрифтом):



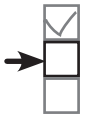

```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.fragment.findNavController
import com.hfad.tasks.databinding.FragmentTasksBinding

class TasksFragment : Fragment() {
    private var _binding: FragmentTasksBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentTasksBinding.inflate(inflater, container, false)
        val view = binding.root

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao

        val viewModelFactory = TasksViewModelFactory(dao)
        val viewModel = ViewModelProvider(
                this, viewModelFactory).get(TasksViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
```

Это импортирование стало ненужным, удалите его.

Импортируйте этот класс.

Продолжение на следующей странице.

Код TasksFragment.kt (продолжение)



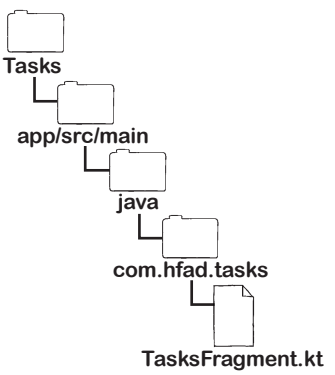
Реакция на щелчки
Навигация
Вывод записи

```
        val adapter = TaskItemAdapter { taskId ->
            Toast.makeText(context, "Clicked task $taskId", Toast.LENGTH_SHORT).show()
            viewModel.onTaskClicked(taskId)
        }
        binding.tasksList.adapter = adapter

        viewModel.tasks.observe(viewLifecycleOwner, Observer {
            it?.let {
                adapter.submitList(it)
            }
        })

        viewModel.navigateToTask.observe(viewLifecycleOwner, Observer { taskId ->
            taskId?.let {
                val action = TasksFragmentDirections
                        .actionTasksFragmentToEditTaskFragment(taskId)
                this.findNavController().navigate(action)
                viewModel.onTaskNavigated()
            }
        })

        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Вызываем этот метод вместо отображения панели Toast.

Наблюдение за свойством navigateToTask.

Этот код должен выполняться тогда, когда navigateToTask присваивается новое значение taskId, отличное от null.



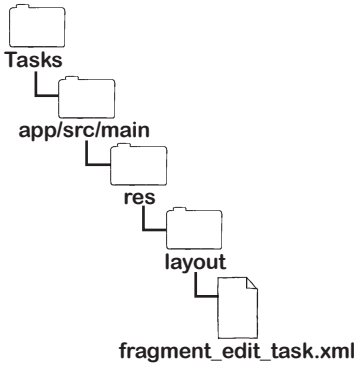
Мы обновили код `TasksViewModel` и `TasksFragment`, чтобы по щелчку на задаче в представлении с переработкой фрагмент `TasksFragment` переходил к `EditTaskFragment` и передавал ему идентификатор задачи.

Следующее, что необходимо сделать, — вывести идентификатор задачи в макете `EditTaskFragment`.

Вывод идентификатора задачи в EditTaskFragment

Чтобы в `EditTaskFragment` выводился идентификатор задачи, необходимо обновить макет фрагмента и код Kotlin. Мы добавим в макет текстовое представление, а затем воспользуемся кодом Kotlin для получения идентификатора задачи и включения его в текстовое представление.

Начнем с включения текстового представления в макет фрагмента; обновите файл *fragment_edit_task.xml* и приведите его к следующему виду:




```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".EditTaskFragment">

    <TextView
        android:id="@+id/task_id"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16sp" />
</LinearLayout>
```

Макет содержит текстовое представление, которое будет использоваться для вывода идентификатора задачи.

Также необходимо обновить EditTaskFragment.kt

После того как текстовое представление будет добавлено в макет `EditTaskFragment`, необходимо заполнить его идентификатором задачи. Для этого в метод `onCreateView()` добавляется следующий код:

Получение ссылки на текстовое представление.

Получаем значение taskId.

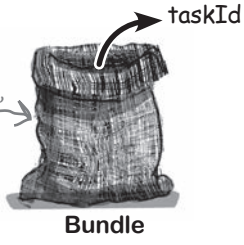
```
val textView = view.findViewById<TextView>(R.id.task_id)
val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
textView.text = taskId.toString()
```

Задаем свойство text текстового представления.

Как видите, в этом решении используется класс `EditTaskFragmentArgs` (сгенерированный плагином Safe Args) для получения значения аргумента `taskId`, переданный `EditTaskFragment`. Затем полученное значение используется для задания текста текстового представления.

Посмотрим, как выглядит полный код `EditTaskFragment`.

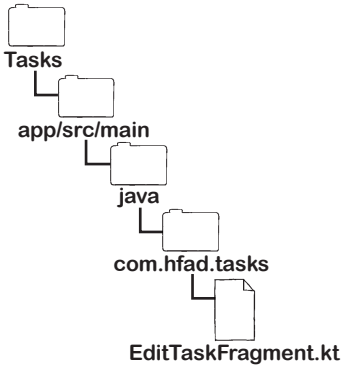
Как вы узнали в главе 7, метод fromBundle() используется для получения любых аргументов, переданных фрагменту.



Реакция на щелчки
Навигация
Вывод записи

Полный код EditTaskFragment.kt

Ниже приведен полный код `EditTaskFragment`; внесите изменения в файл *EditTaskFragment.kt* и приведите его к следующему виду:




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView

class EditTaskFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_edit_task, container, false)
        val textView = view.findViewById<TextView>(R.id.task_id)
        val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
        textView.text = taskId.toString()
        return view
    }
}
```

Идентификатор задачи выводится в текстовом представлении задачи.

Разберемся, что происходит при выполнении этого кода.

Часто Задаваемые Вопросы

В: В коде `EditTaskFragment` для получения ссылки на текстовое представление макета используется метод `findViewById()`. Почему не связывание представления или связывание данных?

О: В данном случае используется `findViewById()`, потому что это самый быстрый способ убедиться в том, что идентификатор задачи был успешно передан фрагменту `EditTaskFragment`. Когда вы убедитесь в том, что эта версия кода работает, можно будет обновить код для использования связывания данных.

В: Когда я попытался обновить код Kotlin, среда Android Studio сообщила, что не может найти классы `TasksFragmentDirections` или `EditTaskFragmentArgs`. Почему это произошло?

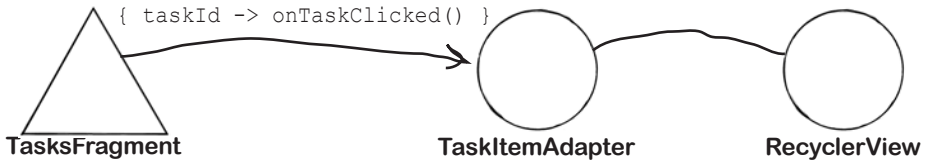
О: Эти классы генерируются плагином Safe Args, поэтому обязательно убедитесь в том, что файлы *build.gradle* были обновлены для его включения. Затем лишний раз проверьте граф навигации и убедитесь в том, что `EditTaskFragment` получает аргумент.

Реакция на щелчки
Навигация
Вывод записи

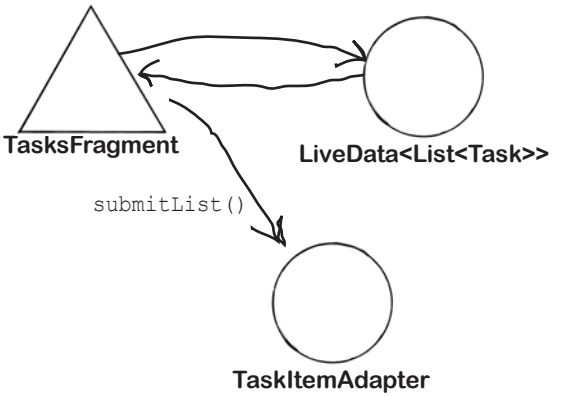
Что происходит при выполнении кода

При выполнении кода происходят следующие события:

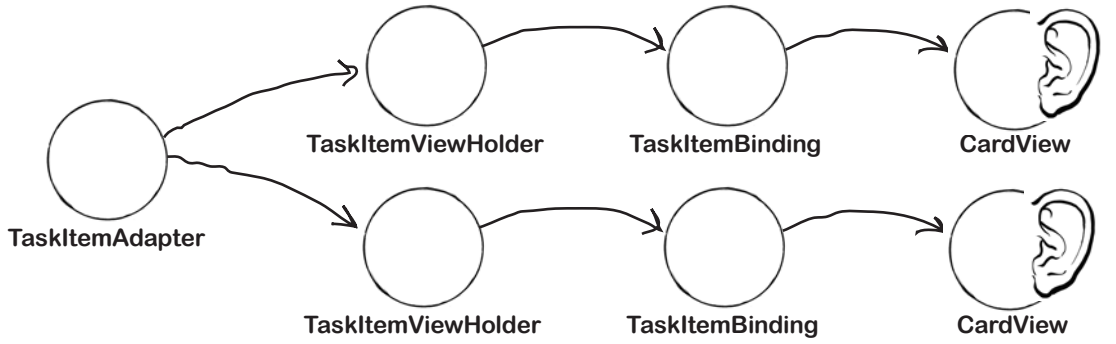
1. **TasksFragment создает объект TaskItemAdapter и назначает его адаптером представления с переработкой.**
Фрагмент передает адаптеру лямбда-выражение `clickListener`, которое вызывает метод `onTaskClicked()` объекта `TasksViewModel` при его выполнении.

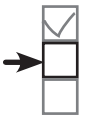


2. **TasksFragment передает List<Task> объекту TaskItemAdapter.**
`List<Task>` содержит актуальный список записей из базы данных.



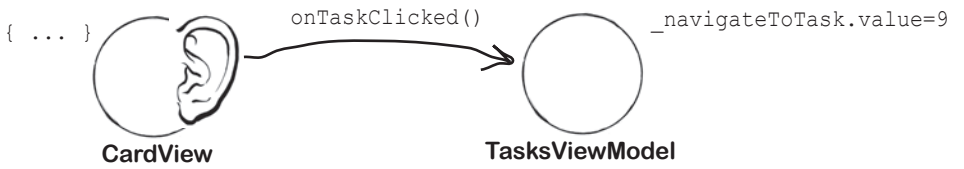
3. **TaskItemAdapter создает набор объектов TaskItemViewHolder, и для корневого представления каждого держателя представления назначается OnClickListener.**
В данном примере корневым представлением является `CardView`.



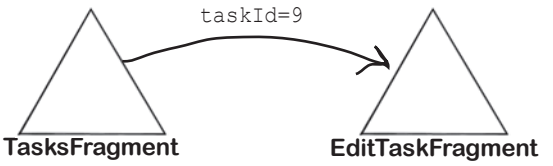


История продолжается

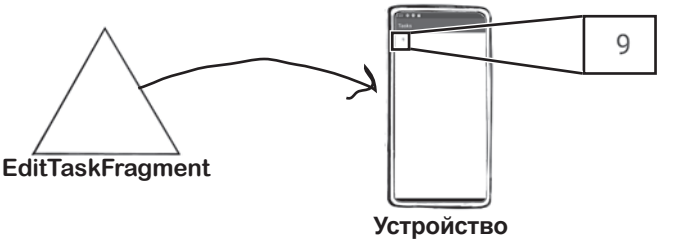
4 **Когда пользователь щелкает на задаче в представлении с переработкой, OnClickListener регистрирует щелчок и выполняет лямбда-выражение.**
Он вызывает метод `onTaskClicked()` объекта `TasksViewModel`, который присваивает свойству `_navigateToTask` идентификатор задачи, на которой был сделан щелчок.



5 **TasksFragment оповещается об обновлении свойства navigateToTask объекта TasksViewModel, которое использует _navigateToTask в качестве резервного свойства.**
Фрагмент переходит к `EditTaskFragment` и передает ему значение свойства `navigateToTask`.



6 **Фрагмент EditTaskFragment получает переданное ему значение taskId и выводит его в макете.**

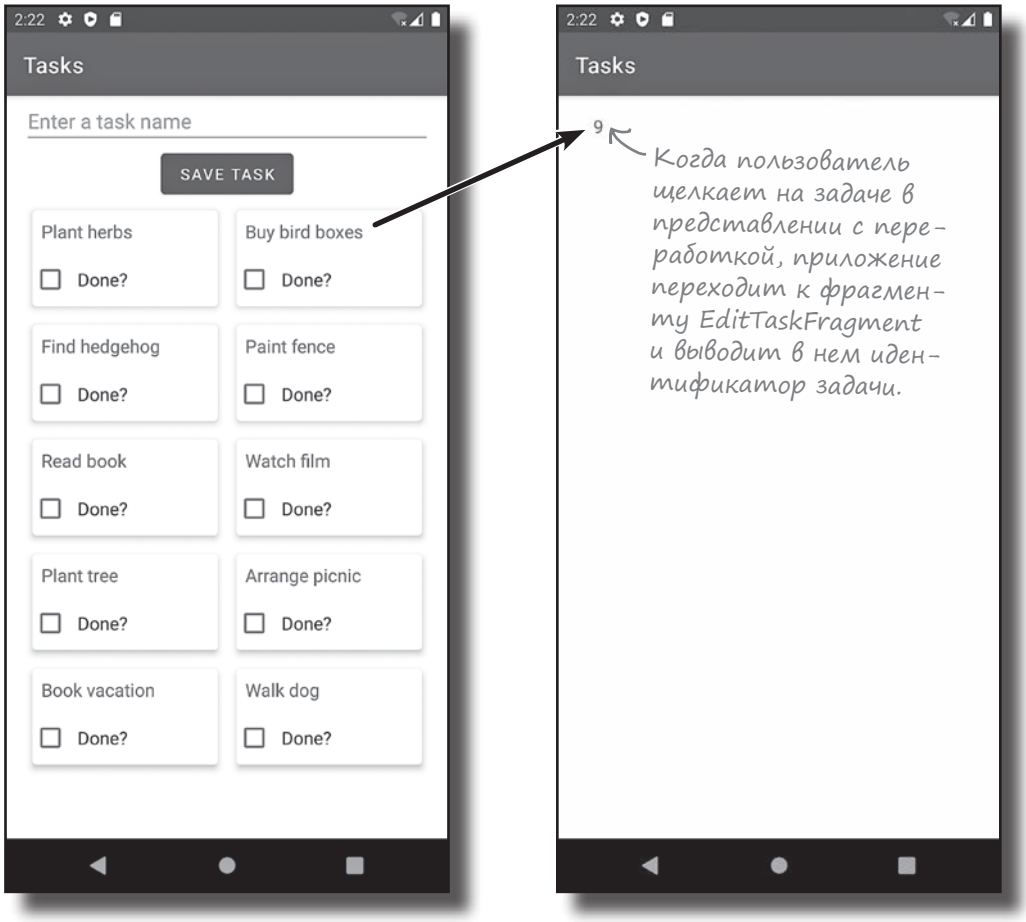


Проведем тест-драйв приложения.

Тест-драйв

При запуске приложения `TasksFragment` отображает сетку карточек в представлении с переработкой, как и в предыдущей версии.

Если щелкнуть на одной из задач, приложение переходит к фрагменту `EditTaskFragment`, который отображает идентификатор задачи.



Вы научились использовать представление с переработкой для перехода к новому фрагменту и передачи идентификатора задачи, на которой был сделан щелчок.

Далее мы обновим приложение Tasks, чтобы по щелчку на элементе фрагмент `EditTaskFragment` отображал полную информацию о задаче и предоставлял возможность обновления или удаления записи из базы данных.

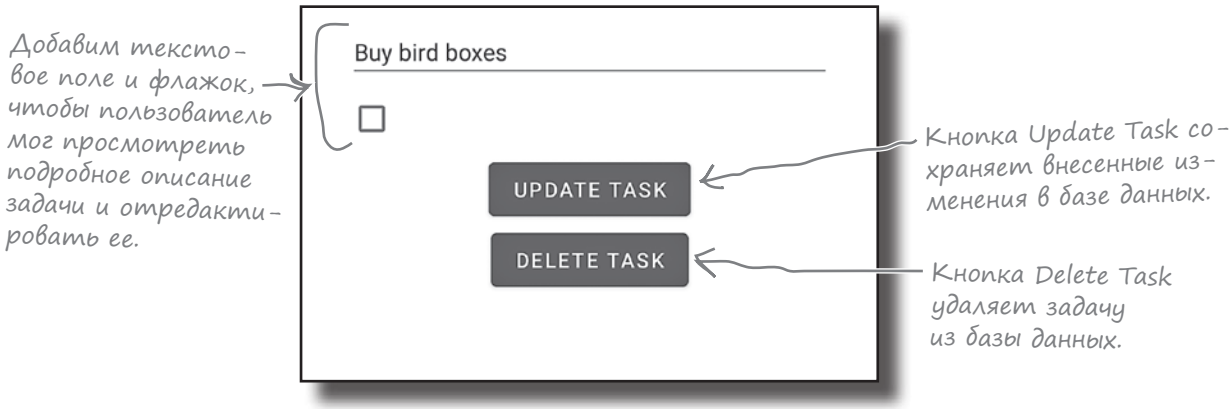
Реакция на щелчки
Навигация
Вывод записи

Использование EditTaskFragment для обновления записей

Мы обновили приложение Tasks, чтобы когда пользователь щелкает на задаче в представлении с переработкой, оно переходило к фрагменту `EditTaskFragment`, в котором выводится идентификатор задачи.

Но *на самом деле* в `EditTaskFragment` должна выводиться полная запись задачи, а пользователь должен иметь возможность обновления или удаления задачи из базы данных.

Для этого мы обновим код `EditTaskFragment`, который должен выглядеть так:



Новая версия фрагмента работает по следующей схеме:

1. **Когда пользователь щелкает на задаче в представлении с переработкой, EditTaskFragment выводит расширенную информацию о ней.**
   Для этого фрагмент читает запись задачи из базы данных, выводит имя задачи и признак ее завершения.

2. **Когда пользователь обновляет описание задачи и щелкает на кнопке Update Task, изменения сохраняются в базе.**
   Информация обновляется в базе данных, после чего происходит возврат к `TasksFragment`.

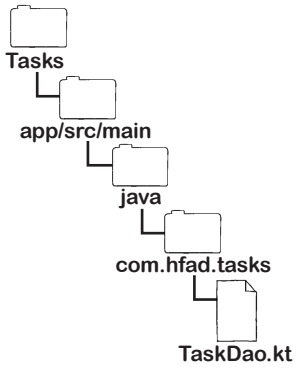
3. **Если пользователь щелкнет на кнопке Delete Task, задача удаляется.**
   Запись задачи удаляется из базы данных, после чего происходит переход к `TasksFragment`.

Для каждого из этих действий фрагмент должен взаимодействовать с базой данных Room; это означает, что он должен использовать интерфейс `TaskDao`, определенный в главе 14. Но прежде чем обновлять приложение, стоит припомнить, что делает интерфейс `TaskDao`.

Использование TaskDao для взаимодействия с записями базы данных

Как вы узнали в главе 14, взаимодействие с записями, хранящимися в базе данных Room, осуществляется через интерфейс DAO. Например, приложение Tasks включает объект DAO с именем `TaskDao`, обеспечивающий взаимодействие с записями задач.

Напомним, как выглядит код `TaskDao`:




```
...
@Dao
interface TaskDao {
    @Insert
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    @Delete
    suspend fun delete(task: Task)

    @Query("SELECT * FROM task_table WHERE taskId = :key")
    fun get(key: Long): LiveData<Task>

    @Query("SELECT * FROM task_table ORDER BY taskId DESC")
    fun getAll(): LiveData<List<Task>>
}
```

Запись задачи вставляется в базу данных.

Эти методы имеют пометку suspend; это означает, что они являются сопрограммами.

Обновление записи задачи.

Удаление задачи.

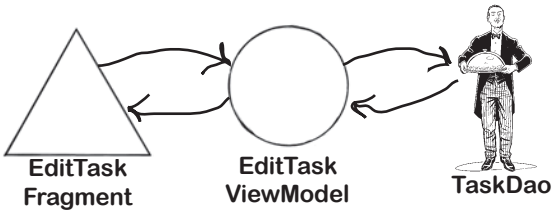
Получает запись задачи с соответствующим идентификатором и возвращает его версию в форме Live Data.

Возвращает список всех задач в форме Live Data.

Как видите, интерфейс включает методы для загрузки одной или нескольких записей из базы данных и сопрограммы для вставки, обновления и удаления записей в фоновом потоке.

Создание модели представления для обращения к методам TaskDao

Мы воспользуемся методами `TaskDao`, чтобы фрагмент `EditTaskFragment` мог прочитать запись задачи из базы данных, обновить ее описание или удалить ее. Вместо того чтобы добавлять этот код в `EditTaskFragment`, мы создадим новую модель представления (с именем `EditTaskViewModel`), которая управляет бизнес-логикой и данными фрагмента. `EditTaskViewModel` будет вызывать методы `TaskDao` и передавать результаты `EditTaskFragment`.



Перейдем к созданию `EditTaskViewModel`.

Создание EditTaskViewModel

Чтобы создать класс `EditTaskViewModel`, выделите пакет *com.hfad.tasks* в папке *app/src/main/java*, затем выберите команду File→New→Kotlin Class/File. Введите имя файла «EditTaskViewModel» и выберите вариант Class.

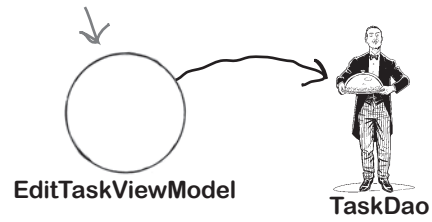
Модель представления должна получить запись задачи...

Прежде всего объект `EditTaskViewModel` должен прочитать запись задачи из базы данных приложения, чтобы отобразить ее в макете `EditTaskFragment`. Для этого конструктору модели представления будут передаваться два значения: идентификатор задачи, который указывает, какую задачу нужно получить, и объект `TaskDao`, который будет использоваться для взаимодействия с базой данных.

Также в модель представления будет добавлено свойство `LiveData<Task>` (с именем `task`), значение которого будет задаваться методом `get()` объекта `TaskDao`. Свойству присваивается запись задачи, которую хочет просмотреть пользователь.

Ниже приведен код, который делает все это; полный код `EditTaskViewModel` будет приведен через пару страниц:

EditTaskViewModel будет использовать TaskDao для взаимодействия с базой данных.



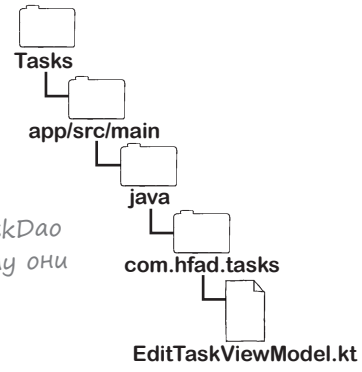
EditTaskViewModel получает идентификатор задачи и объект TaskDao.

```
class EditTaskViewModel(taskId: Long, val dao: TaskDao) : ViewModel() {
    val task = dao.get(taskId)
}
```

Свойству task присваивается LiveData<Task>.

...и содержать методы для обновления и удаления задачи

Мы также добавим в `EditTaskViewModel` методы `updateTask()` и `deleteTask()`, которые будут использоваться `EditTaskFragment` для обновления или удаления записей задач. Эти методы будут вызывать сопрограммы `update()` и `delete()` объекта `TaskDao`:




```
fun updateTask() {
    viewModelScope.launch {
        dao.update(task.value!!)
    }
}

fun deleteTask() {
    viewModelScope.launch {
        dao.delete(task.value!!)
    }
}
```

Методы update() и delete() объекта TaskDao реализованы как сопрограммы, поэтому они должны вызываться в блоке launch.

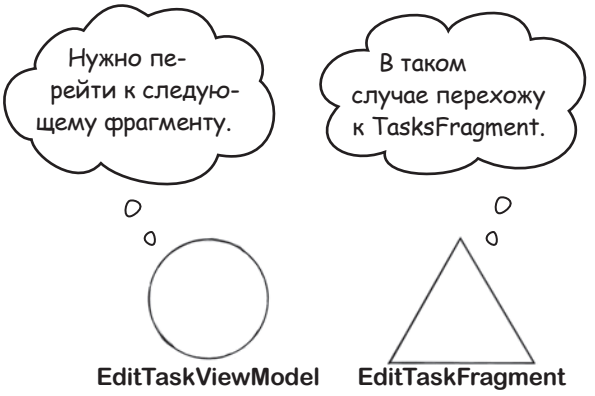
Если значение Task равно null, выдается исключение null-указателя.

Прежде чем добавлять этот код в `EditTaskViewModel`, посмотрим, что еще должен делать код модели представления.

Реакция на щелчки
Навигация
Вывод записи

EditTaskViewModel сообщает EditTaskFragment, когда выполнить переход

Последнее, что должен сделать объект `EditTaskViewModel`, — сообщить фрагменту `EditTaskFragment`, когда ему следует вернуться к `TasksFragment`. Для этого мы добавим в модель представления новое свойство `LiveData<Boolean>` (с именем `navigateToList`) наряду с резервным свойством с именем `_navigateToList`. `EditTaskFragment` будет наблюдать за свойством `navigateToList`, и когда свойство принимает значение *true*, происходит переход к `TasksFragment`.

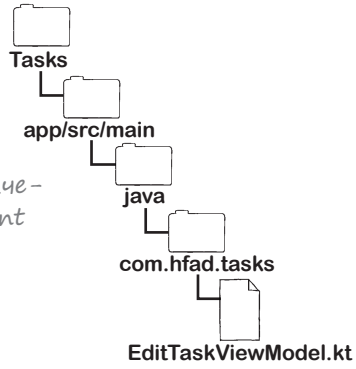


Значение *true* будет присваиваться `_navigateToList` в методах `updateTask()` и `deleteTask()` модели представления. Это означает, что сразу же после обновления или удаления записи задачи приложение переходит к `TasksFragment`.

Ниже приведен обновленный код этих методов:

```
fun updateTask() {
    viewModelScope.launch {
        dao.update(task.value!!)
        _navigateToList.value = true
    }
}

fun deleteTask() {
    viewModelScope.launch {
        dao.delete(task.value!!)
        _navigateToList.value = true
    }
}
```

Когда это свойство принимает значение true, фрагмент EditTaskFragment должен перейти к TasksFragment.



Также в `EditTaskViewModel` будет добавлен метод с именем `onNavigatedToList()`, который возвращает `_navigateToList` значение *false*.

Код этого метода выглядит так:

```
fun onNavigatedToList() {
    _navigateToList.value = false
}
```

EditTaskFragment вызывает этот метод при завершении перехода.

Полный код `EditTaskViewModel` приведен на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

Полный код EditTaskViewModel.kt

Ниже приведен полный код `EditTaskViewModel`; убедитесь в том, что код *EditTaskViewModel.kt* содержит все изменения (выделенные жирным шрифтом):

```
package com.hfad.tasks

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class EditTaskViewModel(taskId: Long, val dao: TaskDao) : ViewModel() {
    val task = dao.get(taskId)
    private val _navigateToList = MutableLiveData<Boolean>(false)
    val navigateToList: LiveData<Boolean>
        get() = _navigateToList

    fun updateTask() {
        viewModelScope.launch {
            dao.update(task.value!!)
            _navigateToList.value = true
        }
    }

    fun deleteTask() {
        viewModelScope.launch {
            dao.delete(task.value!!)
            _navigateToList.value = true
        }
    }

    fun onNavigatedToList() {
        _navigateToList.value = false
    }
}
```

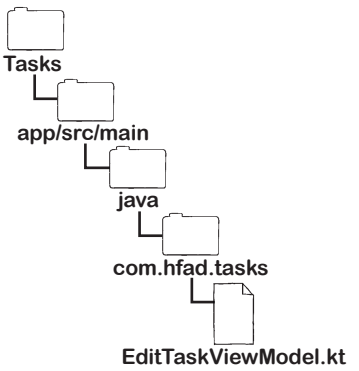
Импортируйте эти классы.

Модель представления получает эти аргументы.

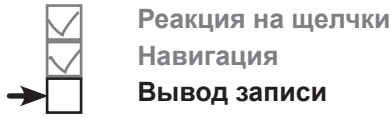
Добавьте эти свойства.

Добавьте методы для обновления и удаления задачи.

Добавьте метод, возвращающий _navigateToList значение false.

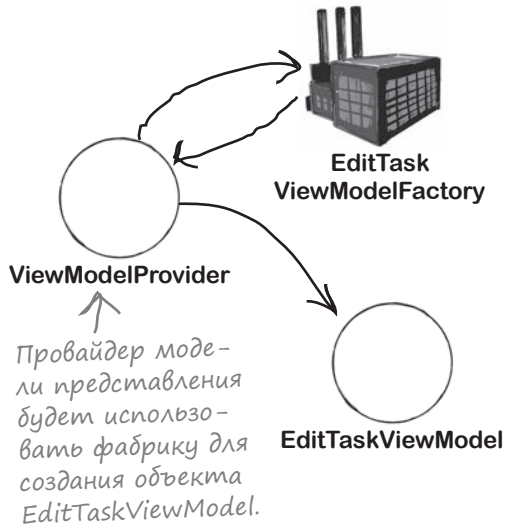


Это весь код, необходимый для `EditTaskViewModel`.
Что же дальше?



EditTaskViewModel необходима фабрика модели представления

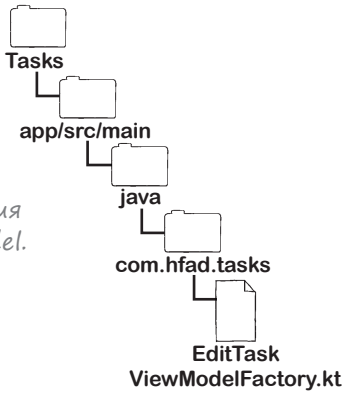
Следующее, что необходимо сделать, — определить фабрику модели представления с именем `EditTaskViewModelFactory`, которая используется `EditTaskFragment` для создания экземпляра `EditTaskViewModel`. Как вы узнали в главе 11, фабрика модели представления необходима для всех моделей представлений, которые, как и `EditTaskViewModel`, не содержат конструктор без аргументов.



Создание EditTaskViewModelFactory

Чтобы создать фабрику, выделите пакет *com.hfad.tasks* в папке *app/src/main/java* и выберите комаду File→New→Kotlin Class/File. Введите имя файла «EditTaskViewModelFactory» и выберите вариант Class.

Когда файл будет создан, обновите код в файле *EditTaskViewModelFactory.kt*, чтобы он выглядел так:




```
package com.hfad.tasks

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider

class EditTaskViewModelFactory(private val taskId: Long,
                               private val dao: TaskDao)
            : ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(EditTaskViewModel::class.java)) {
            return EditTaskViewModel(taskId, dao) as T
        }
        throw IllegalArgumentException("Unknown ViewModel")
    }
}
```

Импортируйте эти классы.

Эти аргументы необходимы фабрике для создания EditTaskViewModel.

Создание объекта EditTaskViewModel.

Этот код практически идентичен коду всех остальных классов фабрик моделей представлений, которые создавались для других приложений.

После написания кода класса `EditTaskViewModel` и его фабрики нужно обновить код фрагмента `EditTaskFragment` и его макета. Начнем с макета.

В fragment_edit_task.xml должны отображаться данные задачи

Обновим файл *fragment_edit_task.xml*, чтобы он включал текстовое поле и флажок, в котором будут отображаться данные `Task`. Мы воспользуемся механизмом связывания данных для связывания этих представлений со свойствами `taskName` и `taskDone` в задаче `EditTaskViewModel`.

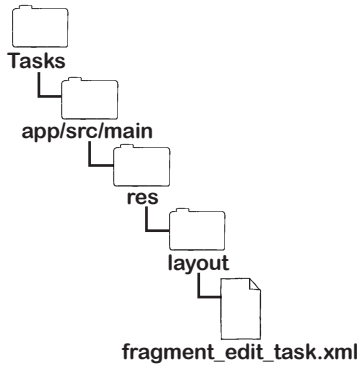
Также в макет будут добавлены две кнопки, которые будут вызывать методы `deleteTask()` и `updateTask()` объекта `EditTaskViewModel` и позволят пользователю обновить или удалить запись задачи.

Ниже приведена новая версия кода макета; обновите файл *fragment_edit_task.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".EditTaskFragment">

    <data>
        <variable
            name="viewModel"
            type="com.hfad.tasks.EditTaskViewModel" />
    </data>

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp"
        tools:context=".EditTaskFragment" >

        <TextView EditText
            android:id="@+id/task_id"
            android:id="@+id/task_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="16sp"
            android:inputType="text"
            android:text="@={viewModel.task.taskName}" />
```

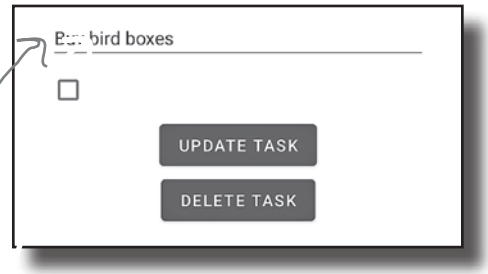
Элемент <layout> добавляется в корневое представление макета.

Определяем переменную связывания данных с именем viewModel.



Переместите эти строки в элемент <layout>.

Замените элемент TextView на EditText и присвойте ему другой идентификатор.



Имя задачи выводится в текстовом поле.

text связывается с именем задачи.

Продолжение на следующей странице.

fragment_edit_task.xml (продолжение)

Добавьте флажок и свяжите его со свойством taskDone задачи.

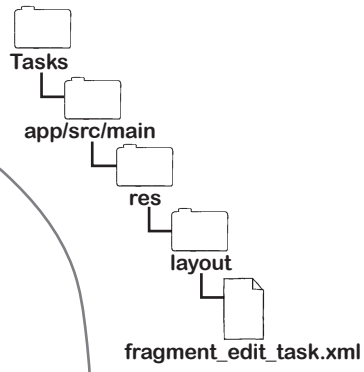
```
<CheckBox
    android:id="@+id/task_done"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:checked="@={viewModel.task.taskDone}" />
```

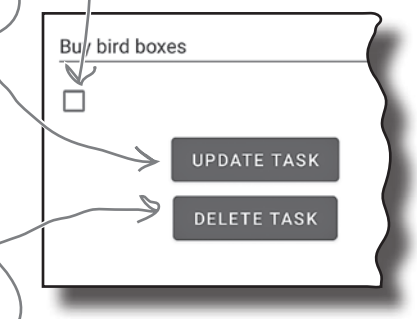
Добавьте кнопку для вызова метода updateTask() модели представления.

```
<Button
    android:id="@+id/update_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Update Task"
    android:onClick="@{() -> viewModel.updateTask()}" />
```

Добавьте вторую кнопку для вызова метода deleteTask() модели представления.

```
<Button
    android:id="@+id/delete_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Delete Task"
    android:onClick="@{() -> viewModel.deleteTask()}" />
</LinearLayout>
</layout>
```





Обновление EditTaskFragment.kt

Осталось внести последнее изменение в приложение Tasks — обновить код Kotlin объекта `EditTaskFragment`. Этот код должен:

1. **Задать значение переменной связывания данных viewModel макета.**
   Ей необходимо присвоить экземпляр `EditTaskViewModel`, который будет создаваться фрагментом.

2. **Назначить владельца жизненного цикла макета.**
   Это необходимо для того, чтобы макет мог взаимодействовать со свойствами Live Data.

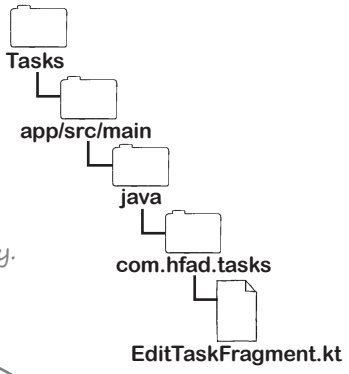
3. **Наблюдать за свойством navigateToList модели представления.**
   Когда свойство принимает значение *true*, фрагмент переходит к `TasksFragment` и вызывает метод `onNavigatedToList()` объекта `EditTaskViewModel`.

Код, который все это делает, вам уже знаком, поэтому мы приведем обновленный код `EditTaskFragment` на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

Полный код EditTaskFragment.kt

Ниже приведен полный код `EditTaskFragment`: убедитесь в том, что файл *EditTaskFragment.kt* содержит все изменения (выделенные жирным шрифтом):




```
package com.hfad.tasks

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
~~import android.widget.TextView~~
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.findNavController
import com.hfad.tasks.databinding.FragmentEditTaskBinding

class EditTaskFragment : Fragment() {
    private var _binding: FragmentEditTaskBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        ~~val view = inflater.inflate(R.layout.fragment_edit_task, container, false)~~
        _binding = FragmentEditTaskBinding.inflate(inflater, container, false)
        val view = binding.root

        val taskId = EditTaskFragmentArgs.fromBundle(requireArguments()).taskId
        ~~val textView = view.findViewById<TextView>(R.id.task_id)~~
        ~~textView.text = taskId.toString()~~

        val application = requireNotNull(this.activity).application
        val dao = TaskDatabase.getInstance(application).taskDao
```

Удалите эту строку.

Импортируйте эти классы.

Теперь фрагмент использует связывание данных, поэтому необходимо добавить эти свойства.

Удалите эту строку.

Добавьте эти строки.

Удалите эти строки.

Добавьте эти строки. Они необходимы для создания EditTaskViewModelFactory.

Продолжение на следующей странице.

Реакция на щелчки
Навигация
Вывод записи

EditTaskFragment (продолжение)

```
        val viewModelFactory = EditTaskViewModelFactory(taskId, dao)
        val viewModel = ViewModelProvider(this, viewModelFactory)
                                        .get(EditTaskViewModel::class.java)
        binding.viewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner

        viewModel.navigateToList.observe(viewLifecycleOwner, Observer { navigate ->
            if (navigate) {
                view.findNavController()
                    .navigate(R.id.action_editTaskFragment_to_tasksFragment)
                viewModel.onNavigatedToList()
            }
        })
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

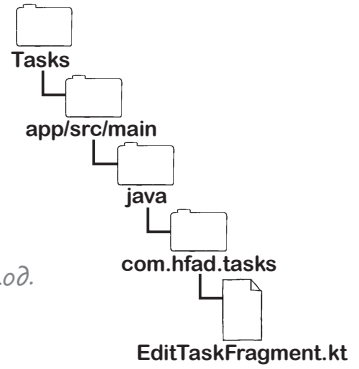
Фабрика модели представления используется для получения ссылки на модель представления.

Присваивается значение переменной связывания данных.

Позволяет макету взаимодействовать со свойствами Live Data.

Переходим к TasksFragment, когда свойству navigateToList присваивается true, и вызываем метод onNavigatedToList() модели представления.

Добавьте этот метод.



И это все, что необходимо для отображения данных `Task` в макете, с возможностью обновления или удаления записи. Давайте разберемся, что происходит в коде во время его выполнения.

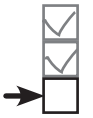
Часто Задаваемые Вопросы

В: Вы сказали, что методы `updateTask()` и `deleteTask()` в `EditTaskViewModel` используют сопрограммы для обновления и удаления записей задач в фоновом потоке. Это так необходимо?

О: Да. Если вы попытаетесь выполнить код базы данных, что может занять много времени, компилятор выдаст сообщение об ошибке.

В: Можно ли добавить кнопку навигации Up в это приложение?

О: Да! Это отличная мысль, так как у пользователя появляется простой способ перехода от `EditTaskFragment` к `TasksFragment` без обновления записей базы данных. Код включения этой кнопки мы не приводим, но предлагаем вам сделать это самостоятельно. Если вам нужно напомнить, как реализуется навигация Up, вам стоит вернуться к главе 8.

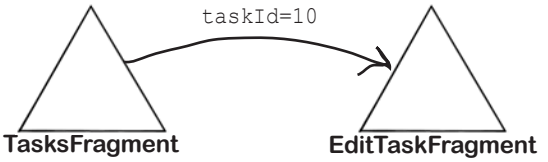


Реакция на щелчки
Навигация
Вывод записи

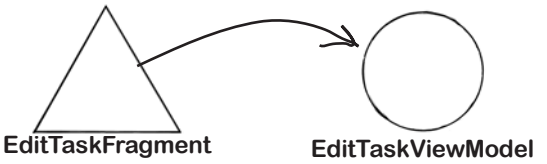
Что происходит при выполнении кода

При выполнении приложения происходят следующие события:

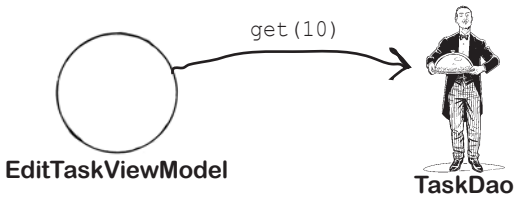
1. **Когда пользователь щелкает на задаче, TasksFragment переходит к EditTaskFragment и передает идентификатор задачи.**



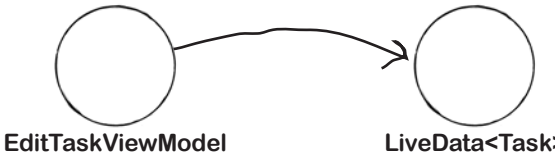
2. **EditTaskFragment получает ссылку на объект EditTaskViewModel.**



3. **EditTaskViewModel вызывает метод get() объекта TaskDao, передавая ему идентификатор задачи.**

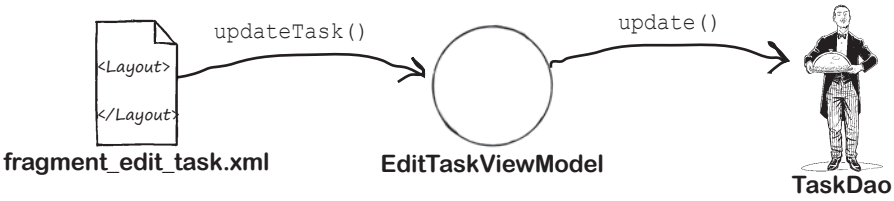


4. **Метод get() объекта TaskDao возвращает значение LiveData<Task>, которое присваивается свойству task объекта EditTaskViewModel.**

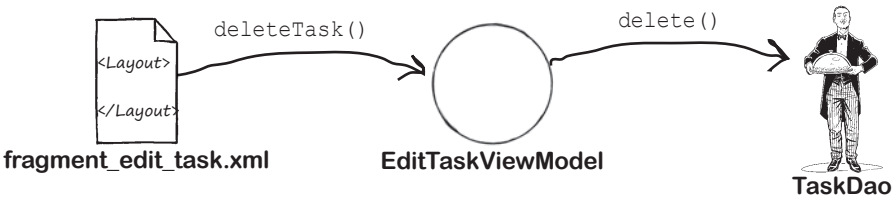


История продолжается

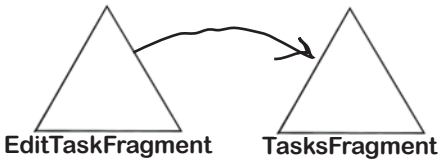
5a **Когда пользователь щелкает на кнопке Update Task, кнопка вызывает метод updateTask() объекта EditTaskViewModel.**
Этот метод использует метод `update()` объекта `TaskDao` для обновления записи в базе данных.



5b **Когда пользователь щелкает на кнопке Delete Task, вызывается метод deleteTask() объекта EditTaskViewModel.**
Этот метод использует метод `delete()` объекта `TaskDao` для удаления записи.



6 **Приложение переходит к TasksFragment.**
Любые изменения, внесенные в запись задачи, отражаются в представлении с переработкой.



Проведем тест-драйв приложения.

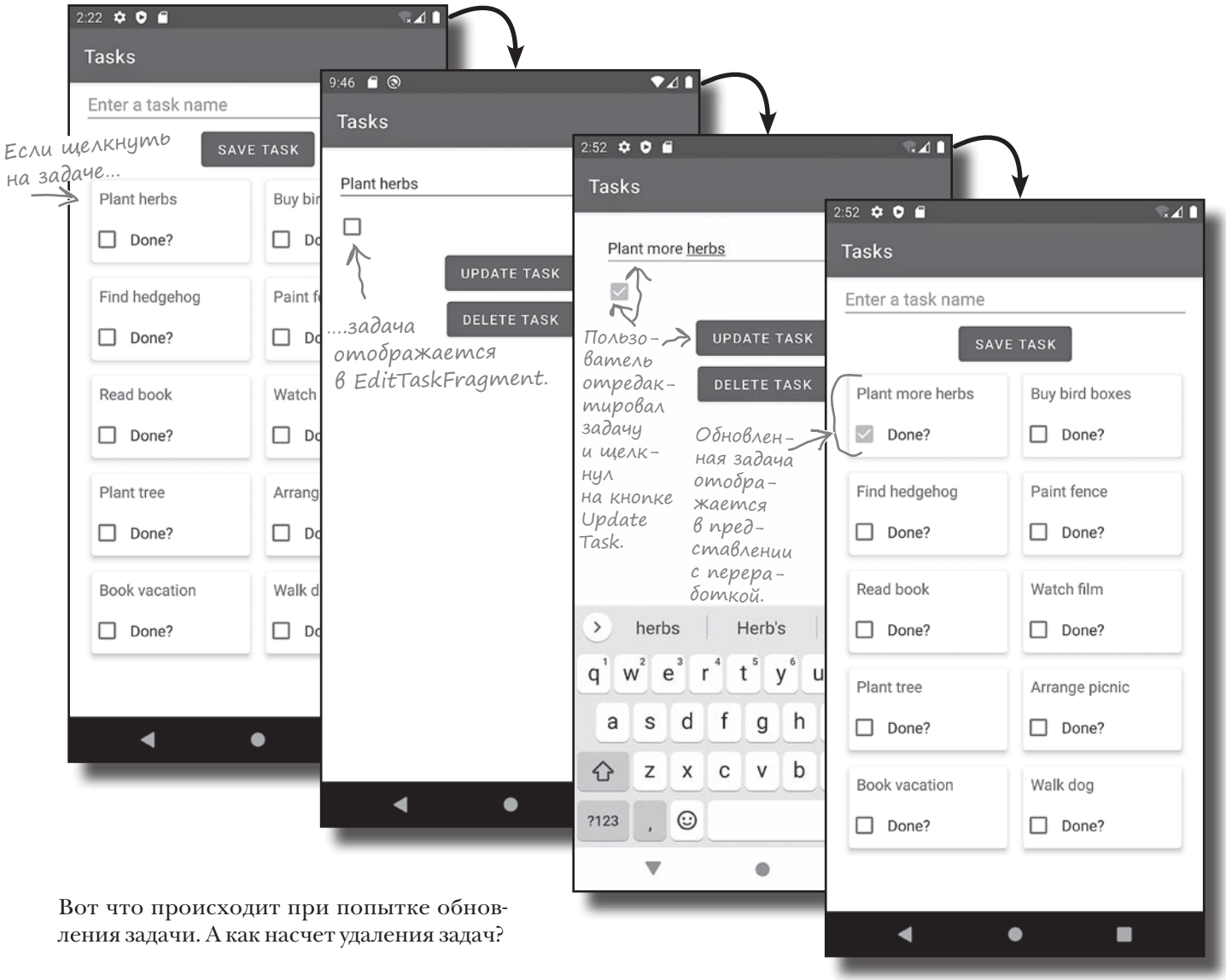
Тест-драйв

Реакция на щелчки
Навигация
Вывод записи

При запуске приложения `TasksFragment` отображает сетку карточек в представлении с переработкой, как и прежде.

Если щелкнуть на одной из задач, приложение переходит к фрагменту `EditTaskFragment`, который отображает данные записи.

Если внести изменения в задачу и щелкнуть на кнопке Update Task, изменения сохраняются в базе данных и отображаются в представлении с переработкой `TasksFragment`.



Вот что происходит при попытке обновления задачи. А как насчет удаления задач?

Тест-драйв (продолжение)

- Реакция на щелчки
- Навигация
- **Вывод записи**

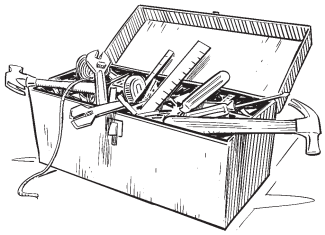
Когда вы щелкаете на задаче, приложение переходит к `EditTaskFragment` и отображает описание задачи, как и прежде.

Если щелкнуть на кнопке Delete Task, запись удаляется из базы данных. Когда приложение переходит к `TaskFragment`, запись исчезает из списка.

Поздравляем! Вы научились строить приложения, использующие представления с переработкой, для перехода между записями, с возможностью обновления и удаления записей. Это позволяет вам предоставить пользователю мощный и гибкий механизм работы с данными в приложении.

ГЛАВА 17

Ваш инструментарий Android



Глава 17 осталась позади, а ваш инструментарий пополнился средствами навигации на базе представлений с переработкой.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Элементы представлений с переработкой могут реагировать на щелчки. В частности, эта возможность может использоваться для перехода к другому фрагменту с расширенной информацией об элементе, на котором был сделан щелчок.
- Чтобы представление с переработкой реагировало на щелчки, следует добавить слушатель `OnClickListener` для корневого представления макета, используемого его элементами.
- Чтобы сообщить слушателю `OnClickListener`, как следует реагировать на щелчки, передайте ему лямбда-выражение. Это лямбда-выражение будет выполняться каждый раз, когда `OnClickListener` регистрирует щелчок.
- Лямбда-выражение может определяться в коде фрагмента и передаваться `OnClickListener` через конструктор адаптера.

В мире Compose

Во всех пользовательских интерфейсах, которые мы строили ранее, использовались представления и файлы макетов. Но благодаря инструментарию **Jetpack Compose это не единственный вариант**. В этой главе перед вами развернется **мир Compose**. Вы научитесь строить пользовательские интерфейсы из компонентов Compose вместо представлений. Вы узнаете, как пользоваться такими компонентами Compose, как **Text**, **Image**, **TextField** и **Button**, размещать их по строкам и столбцам и применять стилевое оформление при помощи **тем**. Вы напишете собственные **компонентные функции** и даже научитесь управлять **состоянием** компонентов Compose с использованием объектов ***MutableState***. Переверните страницу.

UI-компоненты не обязаны быть представлениями

До настоящего момента в этой книге вы учились использовать файлы макетов и представления для построения элегантных интерактивных пользовательских интерфейсов. Но хотя мы сосредоточились на этом подходе, это не единственный вариант.

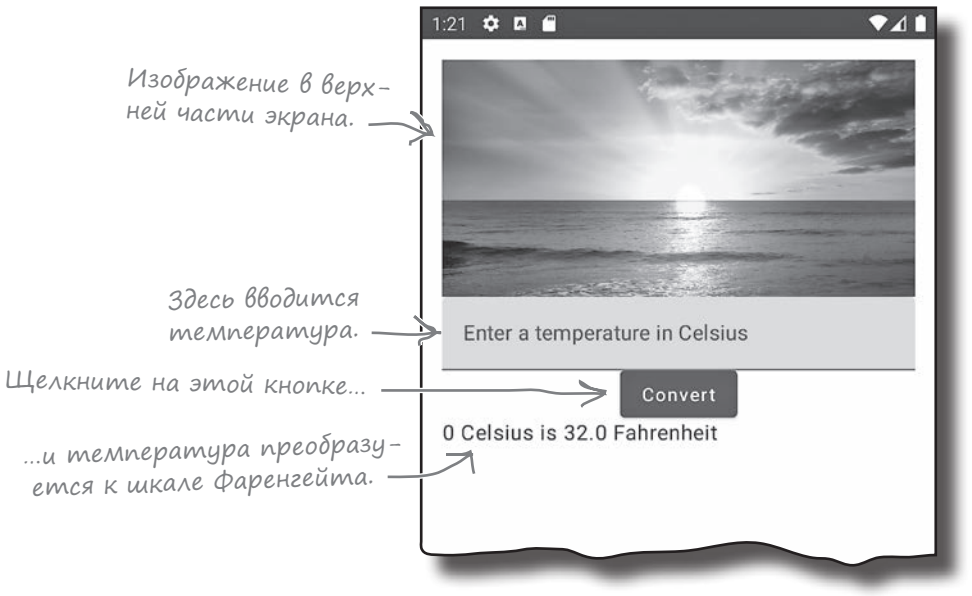
Также пользовательские интерфейсы могут строиться на базе **Jetpack Compose**. Compose является частью Android Jetpack; это целый инструментарий библиотек, вспомогательных программ и API, разработанных для построения платформенных пользовательских интерфейсов в чистом коде Kotlin.

К счастью, при работе с Jetpack Compose можно руководствоваться тем, что вы уже знаете об Android. Например, с Compose можно использовать модели представлений и данные Live Data и даже включать компоненты Compose в существующие пользовательские интерфейсы.

Эта тема более подробно рассматривается в следующей главе.

Мы построим приложение Compose

В этой главе для знакомства с Compose мы построим приложение Temperature Converter, которое преобразует температуру по шкале Цельсия к шкале Фаренгейта. Приложение должно выглядеть так:



Как видите, в приложении используются знакомые компоненты. Тем не менее это приложение написано с использованием Compose.

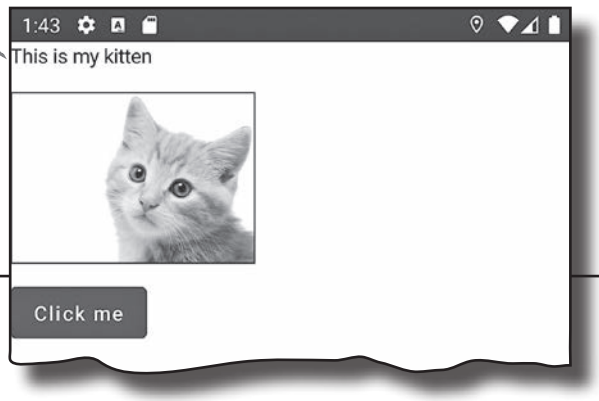
Но прежде чем переходить к созданию приложения, попробуйте разобраться, что делает код Compose, в следующем упражнении.

Возьмите в руку карандаш

Да, мы еще не привели ни одного примера кода Compose. И все же попробуйте разобраться, что делает каждая строка в приведенном ниже коде. Мы даем описания начальных строк, чтобы вам было проще взяться за дело.

```
class MainActivity : ComponentActivity() {  Определяет активность, расширяющую ComponentActivity.
    override fun onCreate(savedInstanceState: Bundle?) {   Переопределяет onCreate().
        super.onCreate(savedInstanceState)   Вызывает onCreate() в суперклассе.
        setContent {
            Column {
                Text("This is my kitten")

                Spacer(Modifier.height(16.dp))

                Image(
                    painter = painterResource(R.drawable.kitten),
                    contentDescription = "Kitten image",
                    modifier = Modifier.border(1.dp, Color.Black)
                )

                Spacer(Modifier.height(16.dp))

                Button(onClick = { }) {
                    Text("Click me")
                }
            }
        }
    }
}
```

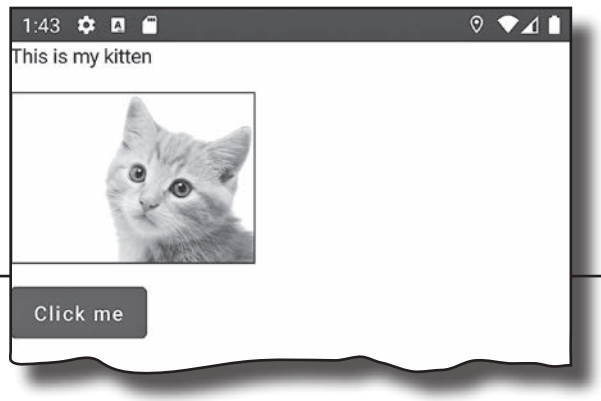
Так должно выглядеть приложение при выполнении кода.



Возьмите в руку карандаш
Решение

Да, мы еще не привели ни одного примера кода Compose. И все же попробуйте разобраться, что делает каждая строка в приведенном ниже коде. Мы даем описания начальных строк, чтобы вам было проще взяться за дело.

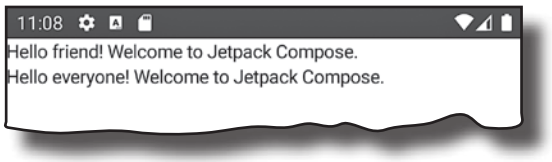
```
class MainActivity : ComponentActivity() {  Определяет активность, расширяющую ComponentActivity.
    override fun onCreate(savedInstanceState: Bundle?) {   Переопределяет onCreate().
        super.onCreate(savedInstanceState)   Вызывает onCreate() в суперклассе.
        setContent {  Задает содержимое активности.
            Column {  Размещает компоненты в столбец.
                Text("This is my kitten")   Выводит текст.

                Spacer(Modifier.height(16.dp))   Добавляет интервал в 16dp.

                Image(  Выводит изображение.
                    painter = painterResource(R.drawable.kitten),  Используемый графический файл.
                    contentDescription = "Kitten image",   Описание изображения.
                    modifier = Modifier.border(1.dp, Color.Black)  Добавляет черную рамку.
                )  Закрывающая скобка для Image.

                Spacer(Modifier.height(16.dp))   Добавляет интервал в 16dp.

                Button(onClick = { }) {  Создает кнопку, по щелчку на которой ничего не происходит.
                    Text("Click me")   Выводит текст на кнопке.
                }  Закрывающая скобка для Button.
            }
        }
    }
}
```



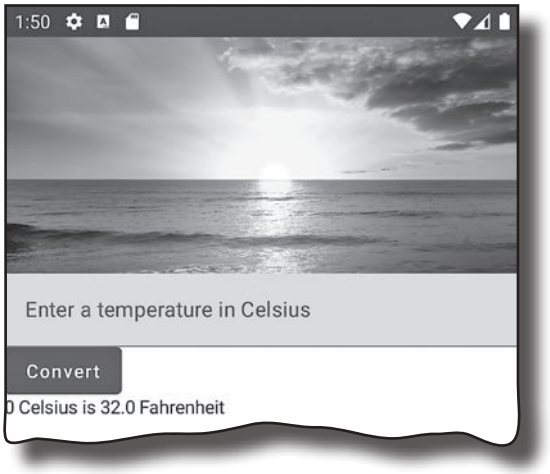
Что мы собираемся сделать

Итак, вы увидели примеры кода и поняли, что он делает. Давайте посмотрим, чем нам предстоит заниматься в этой главе.

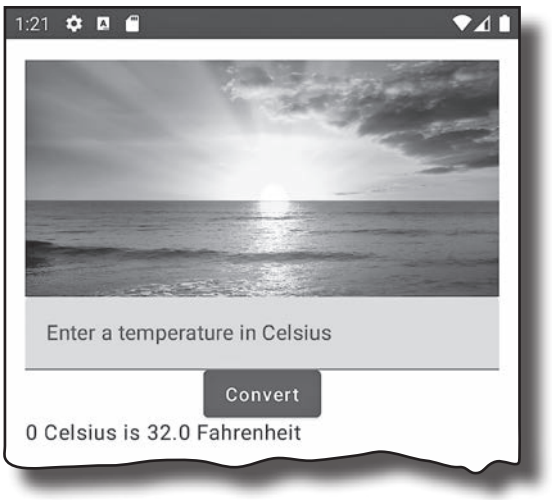
1. **Создание приложения, отображающего два текстовых элемента в столбец.**
Мы создадим новый проект, который использует Compose для вывода фиксированного текста. Затем код будет преобразован в компонентную функцию.



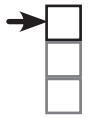
2. **Перевод температуры из шкалы Цельсия в шкалу Фаренгейта.**
На этом шаге мы построим пользовательский интерфейс, который позволяет ввести температуру в градусах по Цельсию. По щелчку на кнопке температура преобразуется в шкалу Фаренгейта.



3. **Изменение внешнего вида приложения.**
Наконец, вы узнаете, как выровнять компоненты по центру, использовать отступы и применять темы.



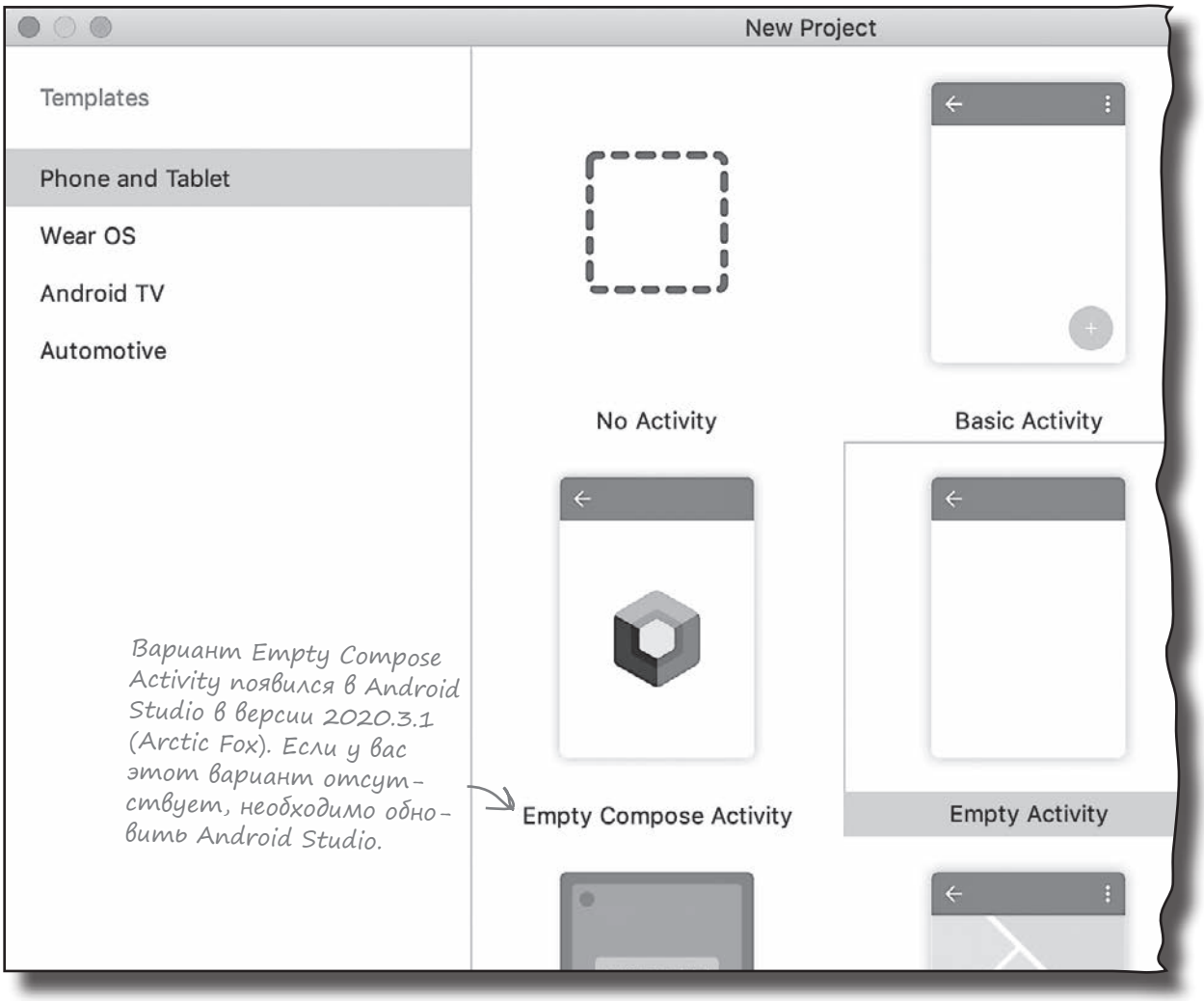
Начнем с создания нового проекта для приложения.



Текст
Преобразователь
Изменение

Создание нового проекта Compose

Мы создадим новый проект Android Studio, который будет использовать Compose для построения своего пользовательского интерфейса. Чтобы создать этот проект, выберите вариант **Empty Compose Activity**:



При выборе этого варианта в проект включаются библиотеки Compose и файлы с кодом. Этот тип проекта лучше всего подходит для построения приложений Android с интерфейсом Compose с нуля.

После выбора варианта Empty Compose Activity щелкните на кнопке Next, чтобы настроить конфигурацию проекта.

Настройка проекта

Следующий экран должен быть вам знаком — он включает те же параметры, которые использовались для настройки проектов в книге.

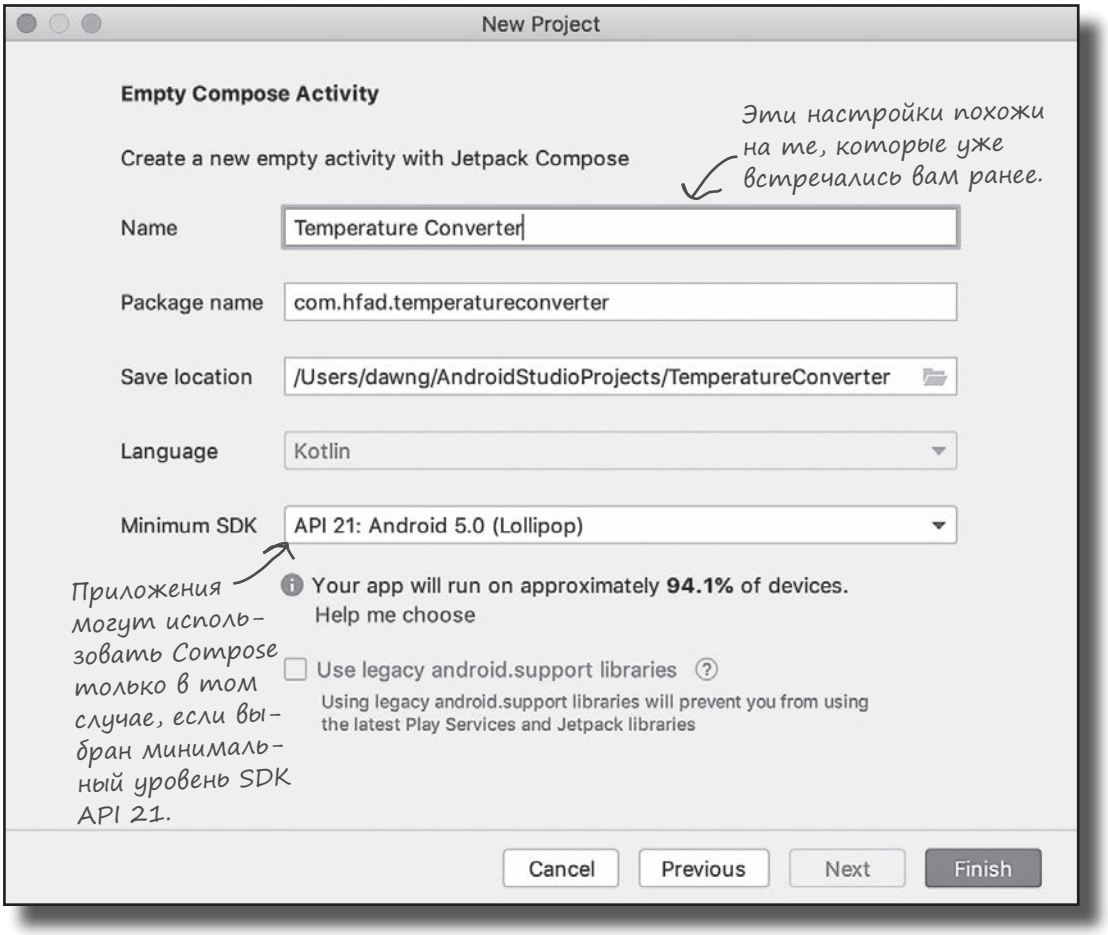
Введите имя «Temperature Converter» и имя пакета «com.hfad.temperatureconverter», подтвердите папку для сохранения по умолчанию.

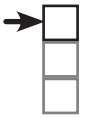
Обратите внимание: язык Kotlin выбирается автоматически, и изменить этот выбор не удастся. Приложения Compose могут строиться *только* на языке Kotlin, так что выбрать другой язык не разрешается.

Выберите минимальный уровень SDK API 21, чтобы приложение работало на большинстве устройств Android. Это самая старая версия SDK, доступная для проектов этого типа, потому что инструментарий Compose совместим только с API 21 и выше.

После выбора всех настроек щелкните на кнопке Finish.

Пользовательские интерфейсы Compose могут создаваться только на языке Kotlin.

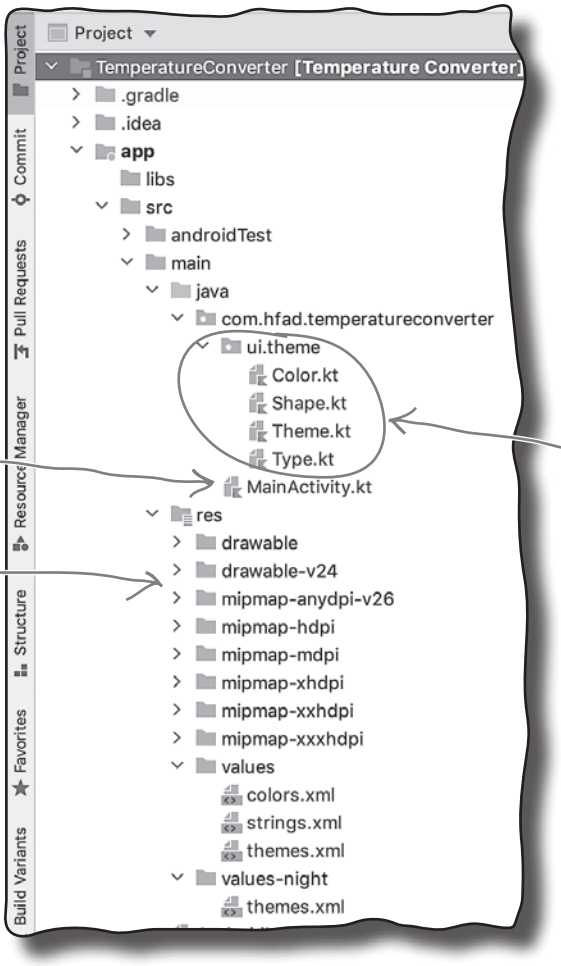




Текст
Преобразователь
Изменение

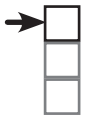
У проектов Compose нет файлов макетов

Когда вы создаете проект в режиме Empty Compose Activity, Android Studio автоматически создает структуру папок и заполняет ее всеми файлами, необходимыми для нового проекта. Структура папок выглядит так:



Многие файлы и папки должны казаться знакомыми — такие же файлы и папки генерируются для проектов, не использующих Compose. Например, в проект включен файл активности с именем *MainActivity.kt* и файл строкового ресурса с именем *strings.xml*.

Главное отличие заключается в том, что **Android Studio не генерирует файлы макетов**. Проекты Compose используют для определения внешнего вида экрана код активности вместо макетов.



Как выглядит код активности Compose

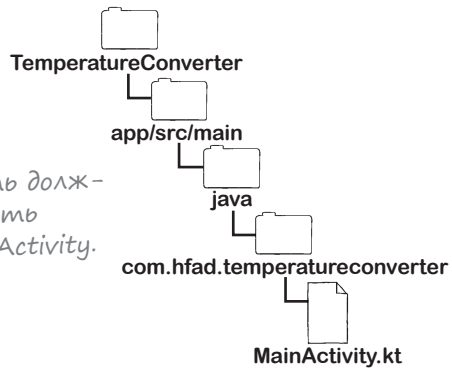
Если вы используете Compose, за поведение и внешний вид приложения отвечает код активности. По этой причине он несколько отличается от кода активности, с которым вы привыкли работать.

Посмотрим, как выглядит базовый код активности Compose. Откройте пакет *com.hfad.temperatureconverter* в папке *app/src/main/java* и откройте файл *MainActivity.kt* (если он не был открыт ранее). Затем **замените код**, сгенерированный Android Studio, следующим кодом:

```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {

        }
    }
}
```

Активность должна расширять ComponentActivity.

Метод onCreate() должен вызвать setContent().



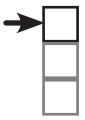
Активности Compose расширяют ComponentActivity

Как видно из кода, активность Compose не расширяет `AppCompatActivity`: вместо этого она использует **`ComponentActivity`**.

`androidx.activity.ComponentActivity` является субклассом `Activity`. Он применяется для определения базовой активности, которая использует для пользовательского интерфейса Compose вместо файла макета.

Как и все остальные активности, которые вам встречались, активность переопределяет метод `onCreate()`. Но вместо того чтобы вызывать `setContentView()` для заполнения макета активности, она использует **`setContent()`**. Это файл расширения, который применяется для включения компонентов Compose в пользовательский интерфейс активности, чтобы они выполнялись при создании активности.

Чтобы вы лучше поняли, как работает этот механизм, воспользуемся Compose для добавления текста в пользовательский интерфейс активности.



Текст
Преобразователь
Изменение

Использование компонента Text для вывода текста

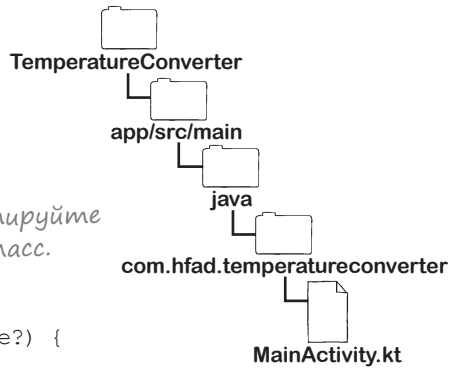
Чтобы в `MainActivity` выводился текст, мы добавим компонент **`Text`** в вызов `setContent()`. Компонент `Text` можно рассматривать как Compose-эквивалент текстового представления. Просто укажите текст, который должен появиться на экране, и он будет выведен в активности.

Ниже приведен код добавления текста в `MainActivity`; обновите файл *MainActivity.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.Text

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello friend! Welcome to Jetpack Compose.")
        }
    }
}
```

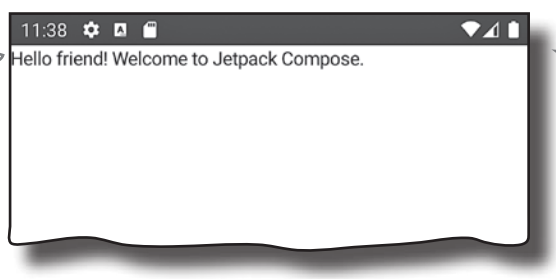
Импортируйте этот класс.

Выводит текст в пользовательском интерфейсе активности.



При выполнении этого кода текст будет выведен в верхней части экрана:

Текст, добавленный в MainActivity.



Панель приложения отсутствует, потому что Android Studio применяет тему без панели приложения.

Итак, теперь вы знаете, как вывести фиксированный текст средствами Compose. Давайте сделаем вывод более гибким и добавим `Text` в **компонентную функцию**.

Использование компонентов Compose в компонентных функциях

Компонентной функцией называется функция, использующая один или несколько компонентов Compose для определения частей пользовательского интерфейса.

Чтобы вы лучше поняли, как работает этот механизм, мы определим компонентную функцию с именем `Hello`, которая получает строковый аргумент с именем пользователя. При выполнении эта функция добавляет строку в компонент `Text`, предназначенный для вывода текста в пользовательском интерфейсе.

Код новой функции `Hello` выглядит так:

```
@Composable
fun Hello(name: String) {
    Text("Hello $name! Welcome to Jetpack Compose.")
}
```

Аннотация превращает эту функцию в компонентную.

Мы добавим эту функцию в MainActivity.kt через пару страниц.

Как видите, функция снабжена аннотацией **`@Composable`**. Эта аннотация необходима для *всех* компонентных функций. Если опустить эту аннотацию, код не будет компилироваться.

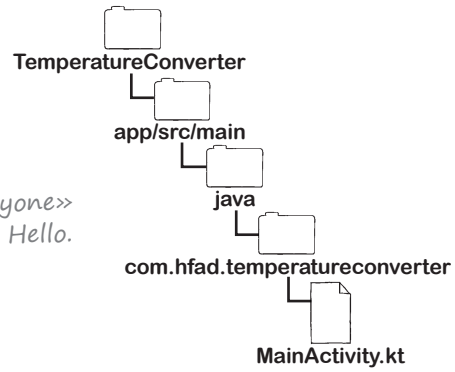
Вызов компонентной функции Hello из setContent()

Пометка функции аннотацией `@Composable` не означает, что функция использует компоненты Compose; **она делает саму функцию компонентом Compose**, который может использоваться в коде, как и любые другие компоненты Compose.

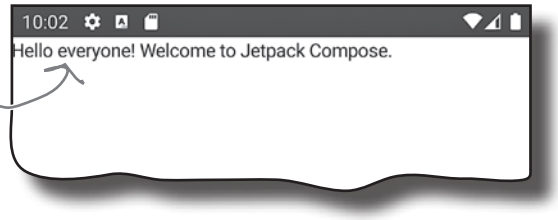
В создаваемом приложении в пользовательском интерфейсе `MainActivity` должен выводиться текст `Hello`. Для этого следует вызвать `Hello` из `setContent()`:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Hello("everyone")
        }
    }
}
```

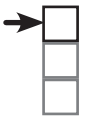
Передает строку «everyone» компонентной функции Hello.



При выполнении этот код выводит текст:



Текст Hello включает переданную строку.



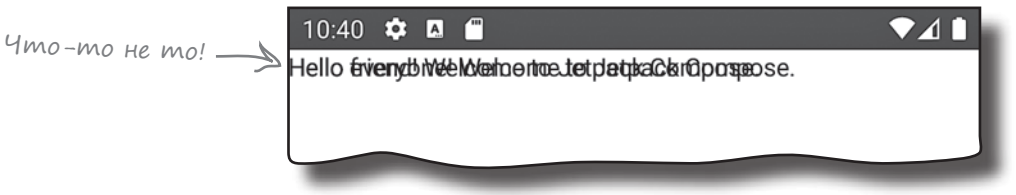
Текст
Преобразователь
Изменение

Многие пользовательские интерфейсы состоят из нескольких компонентов

До настоящего момента вы видели, как использовать один компонент Compose, но на практике пользовательские интерфейсы обычно состоят из нескольких компонентов или одна компонентная функция вызывается в них многократно. Например, если вы хотите, чтобы в приложении выводились два сообщения, компонентную функцию `Hello` можно вызвать дважды с разными аргументами:

```
Hello("friend")
Hello("everyone")
```

Если ваш пользовательский интерфейс включает несколько компонентов, необходимо указать, как они должны быть размещены. Если этого не сделать, Compose наложит компоненты друг на друга:



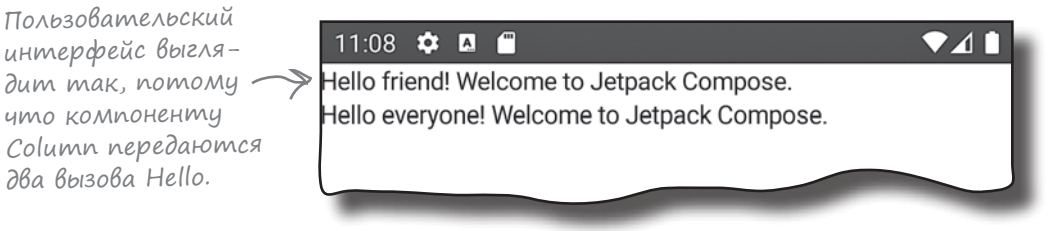
Как же разместить компоненты Compose?

Для размещения компонентов можно использовать компоненты Row и Column

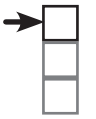
Обычно компоненты размещаются по строкам или по столбцам, и в Compose имеются компоненты **`Row`** и **`Column`** для решения этой задачи. Например, чтобы разместить два компонента `Hello` в столбец, достаточно добавить их в компонент `Column`:

```
Column {
    Hello("friend")
    Hello("everyone")
}
```

При выполнении этого кода компоненты выстраиваются в столбец:



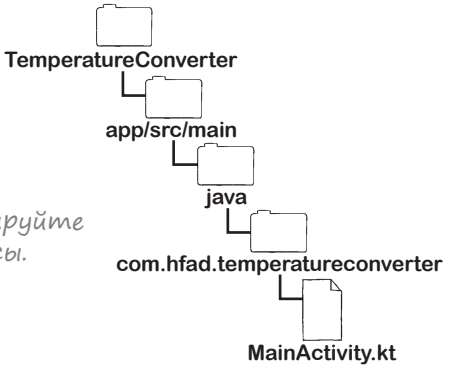
Обновим активность `MainActivity` так, чтобы она строила этот пользовательский интерфейс.



Полный код MainActivity.kt

Мы добавим компонентную функцию `Hello` в *MainActivity.kt*, выполним ее дважды и разместим результаты в столбец.

Ниже приведен полный код новой версии; обновите файл *MainActivity.kt* (изменения выделены жирным шрифтом):




```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.material.Text
import androidx.compose.runtime.Composable

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello friend! Welcome to Jetpack Compose.")
            Column {
                Hello("friend")
                Hello("everyone")
            }
        }
    }
}

@Composable
fun Hello(name: String) {
    Text("Hello $name! Welcome to Jetpack Compose.")
}
```

Импортируйте эти классы.

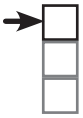
Удалите эту строку.

Два сообщения выводятся в столбец.

Добавьте компонентную функцию Hello. Проследите за тем, чтобы она следовала после определения класса MainActivity.

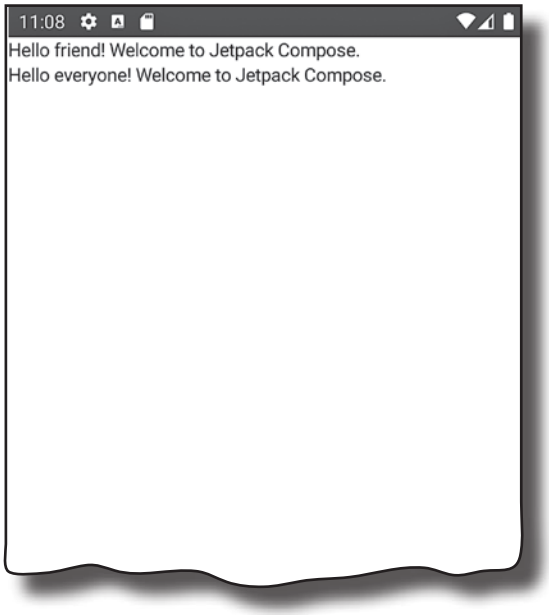
Давайте проведем тест-драйв приложения и посмотрим, как выглядит результат.

Тест-драйв



При выполнении приложения отображается активность `MainActivity`. Она включает два компонента `Hello`, выстроенных в один столбец.

Два компонента располагаются так, как требовалось.



Поздравляем! Вы научились использовать Compose для создания пользовательского интерфейса `MainActivity` вместо добавления представлений в файл макета.

Часто Задаваемые Вопросы

В: Использовать Compose обязательно? Почему не ограничиться представлениями и макетами?

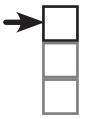
О: Хотя технология Jetpack Compose относительно нова, мы полагаем, что у нее большое будущее. Возможно, когда-нибудь она даже станет стандартным способом построения Android-приложений, и мы определенно рекомендуем вам научиться пользоваться ею.

В: Вы сказали, что каждая активность, использующая компоненты Compose, должна расширять класс `ComponentActivity` или один из его субклассов. А `ComponentActivity` расширяет `AppCompatActivity`?

О: Нет. Вас это может удивить, но `AppCompatActivity` является субклассом `ComponentActivity`, а не наоборот.

В: Понятно. Означает ли это, что я могу добавлять компоненты Compose в существующий код активности?

О: Да. Вы даже можете совмещать компоненты Compose и представления в одном пользовательском интерфейсе. О том, как это делается, будет рассказано в следующей главе.



Предварительный просмотр компонентных функций

У компонентных функций есть еще одна особенность: их результаты можно предварительно просматривать в Android Studio, не загружая приложение на устройство. Это можно делать с любой компонентной функцией, которая не получает аргументов, и даже использовать этот прием для предварительного просмотра целых **композиций** — пользовательских интерфейсов, состоящих из компонентов Compose.

Композицией называется пользовательский интерфейс, состоящий из компонентов Compose.

Чтобы сообщить, что вы хотите выполнить предварительный просмотр компонентной функции, снабдите ее аннотацией `@Preview`. Например, следующий код определяет компонентную функцию с именем `PreviewMainActivity` и позволяет выполнить предварительный просмотр двух компонентов `Hello`, выстроенных в столбец:

```
@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    Column {
        Hello("friend")
        Hello("everyone")
    }
}
```

Добавление аннотации @Preview позволяет выполнить предварительный просмотр функции без ее загрузки на устройство. По умолчанию используется прозрачный фон, но присваивание showBackground значения true переопределяет этот режим.

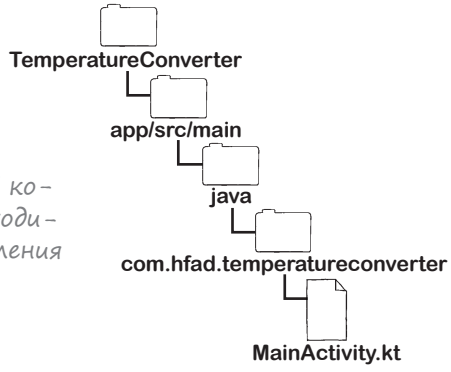
Добавление MainActivityPreview в MainActivity.kt

Чтобы понять, как работает предварительный просмотр, добавим компонентную функцию `MainActivityPreview` в *MainActivity.kt*. Обновите файл и включите в него приведенные ниже изменения:

```
...
import androidx.compose.ui.tooling.preview.Preview

...

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    Column {
        Hello("friend")
        Hello("everyone")
    }
}
```

Импортируйте этот класс.

Добавьте эту функцию в конец файла, чтобы она находилась за пределами определения класса MainActivity.



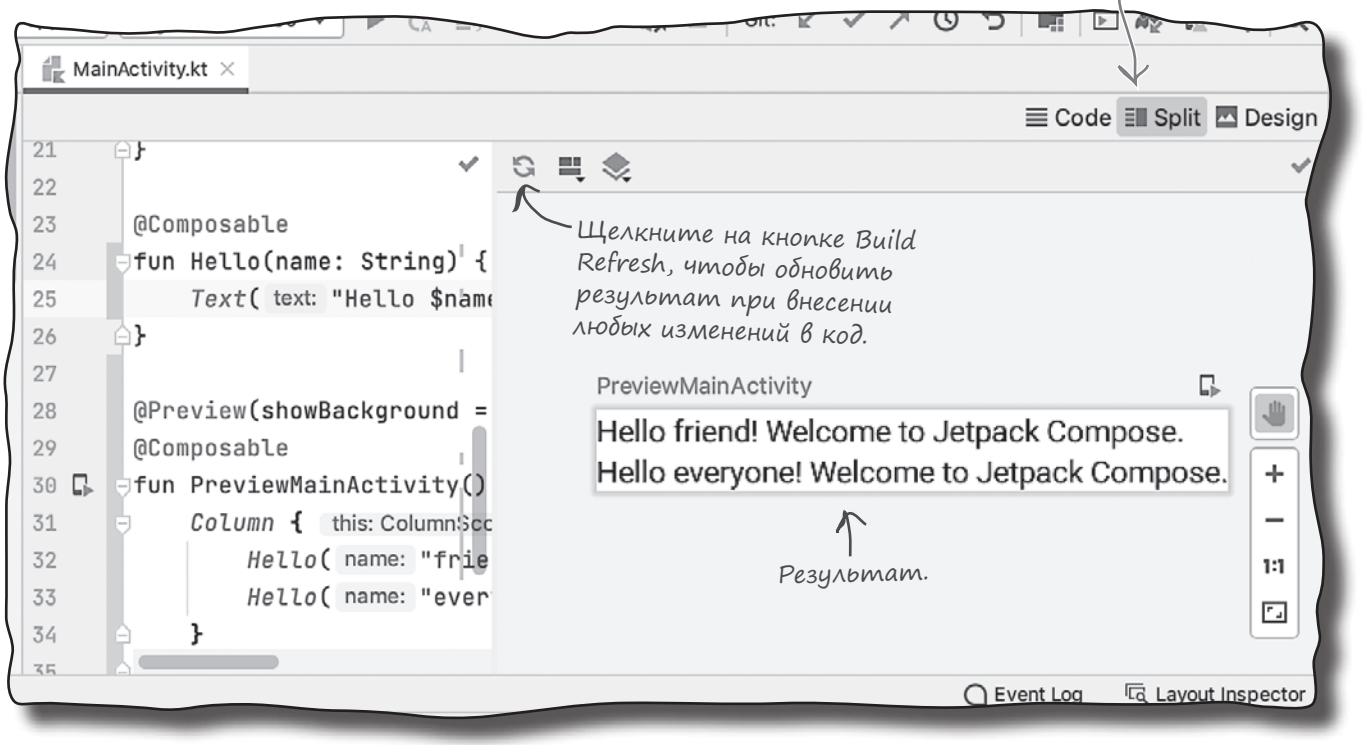
На следующей странице мы покажем, как работает предварительный просмотр.

Текст
Преобразователь
Изменение

Предварительный просмотр компонентов в режимах Design и Split

Любые компонентные функции, помеченные аннотацией `@Preview`, можно просмотреть в режиме Split или Design файла активности. Режим Split позволяет просматривать код одновременно с результатом, а в режиме Design отображается только результат.

Вот как выглядит `MainActivityPreview` при выборе режима Split:

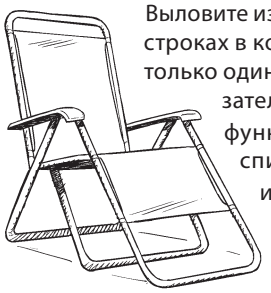


Если вы внесете изменения в компонентную функцию, которая задействована в предварительном просмотре, результат необходимо обновить. Просто щелкните на кнопке Build Refresh в верхнем меню предварительного просмотра, и вы увидите последствия внесенных изменений.

Итак, теперь вы знаете, как просмотреть результат компонентной функции. Проверьте свои силы на следующем упражнении.

Предварительный просмотр возможен для любой компонентной функции, не получающей аргументов.

У бассейна

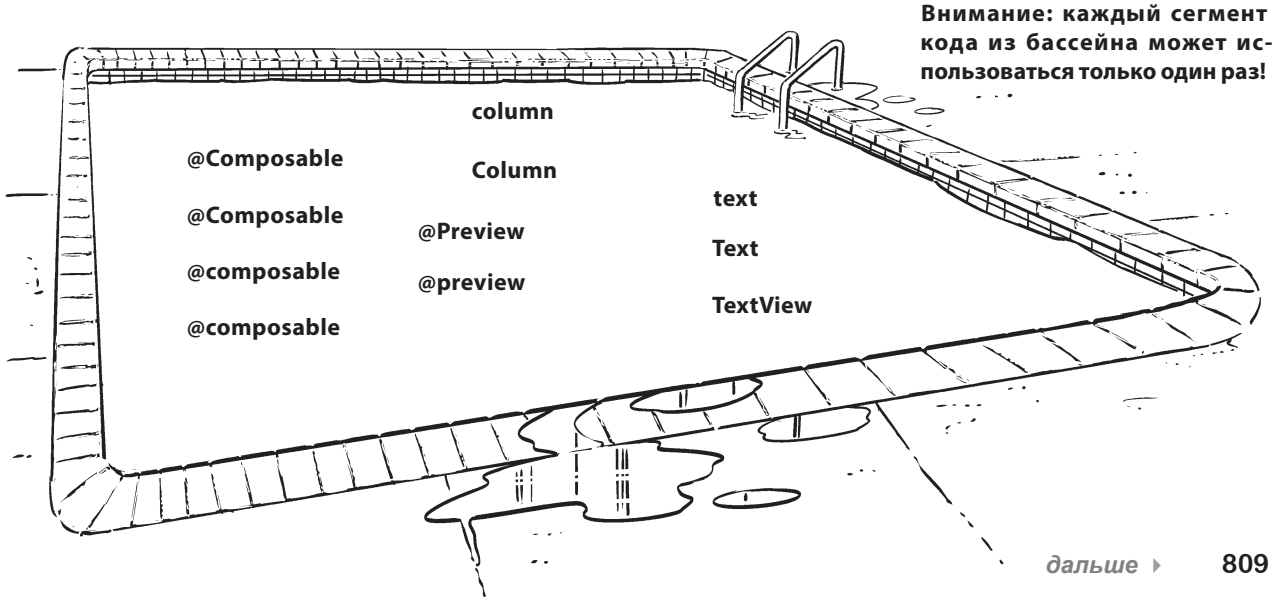


Выловите из бассейна сегменты кода и расставьте их в пустых строках в коде. Каждый сегмент кода может использоваться только один раз, при этом все сегменты использовать необязательно. Ваша **задача** — создать две компонентные функции: первая функция, `TeamHello`, получает список имен и выводит приветствие для каждого имени, а вторая функция, `HelloPreview`, выполняет предпросмотр `TeamHello` и выстраивает приветствия в столбец.

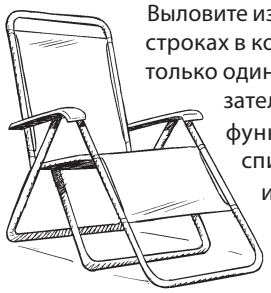
Результат предварительного просмотра HelloPreview должен выглядеть так.

```
.....
fun TeamHello(names: List<String>) {
    for (name in names) {
        .....("Hello $name!")
    }
}


.....(showBackground = true)
.....
fun HelloPreview() {
    .....{
        TeamHello(listOf("Virginia", "Zan", "Katie"))
    }
}
```

Внимание: каждый сегмент кода из бассейна может использоваться только один раз!



У бассейна. Решение



Выловите из бассейна сегменты кода и расставьте их в пустых строках в коде. Каждый сегмент кода может использоваться только один раз, при этом все сегменты использовать необязательно. Ваша **задача** — создать две компонентные функции: первая функция, `TeamHello`, получает список имен и выводит приветствие для каждого имени, а вторая функция, `HelloPreview`, выполняет предпросмотр `TeamHello` и выстраивает приветствия в столбец.

Результат предварительного просмотра HelloPreview должен выглядеть так.

TeamHello помечается как компонентная функция.

```
@Composable
fun TeamHello(names: List<String>) {
    for (name in names) {
        Text("Hello $name!")
    }
}
```

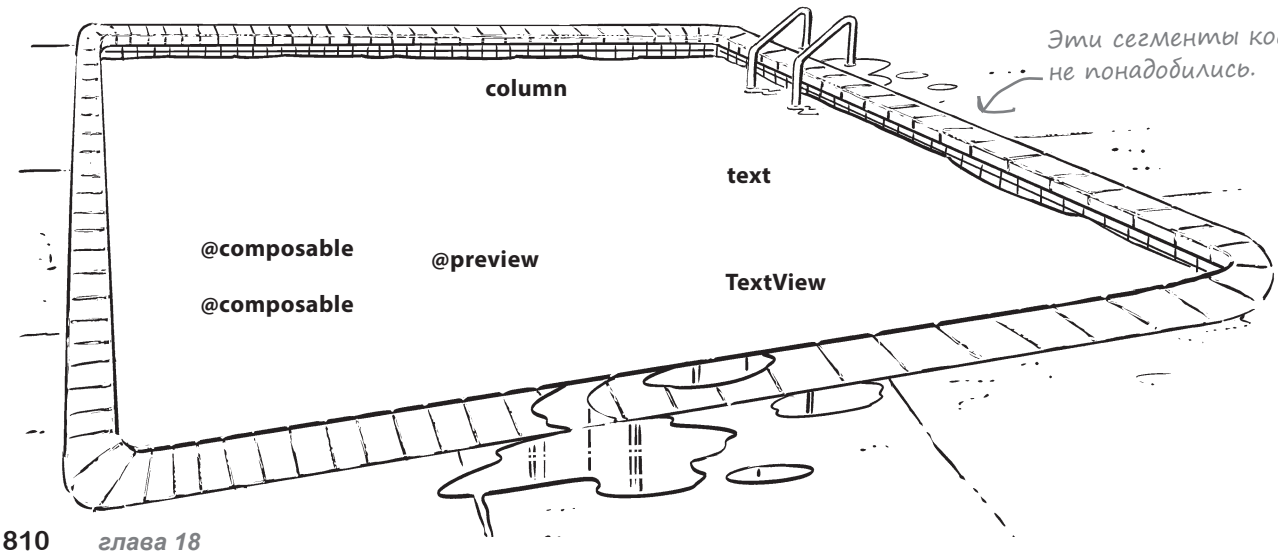
Эта функция должна использоваться для предварительного просмотра.

Функция должна генерировать Text для каждого элемента.

```
@Preview(showBackground = true)
@Composable
fun HelloPreview() {
    Column {
        TeamHello(listOf("Virginia", "Zan", "Katie"))
    }
}
```

HelloPreview также необходимо снабдить аннотацией @Composable.

В предварительном просмотре текстовые элементы должны быть выстроены в столбец.



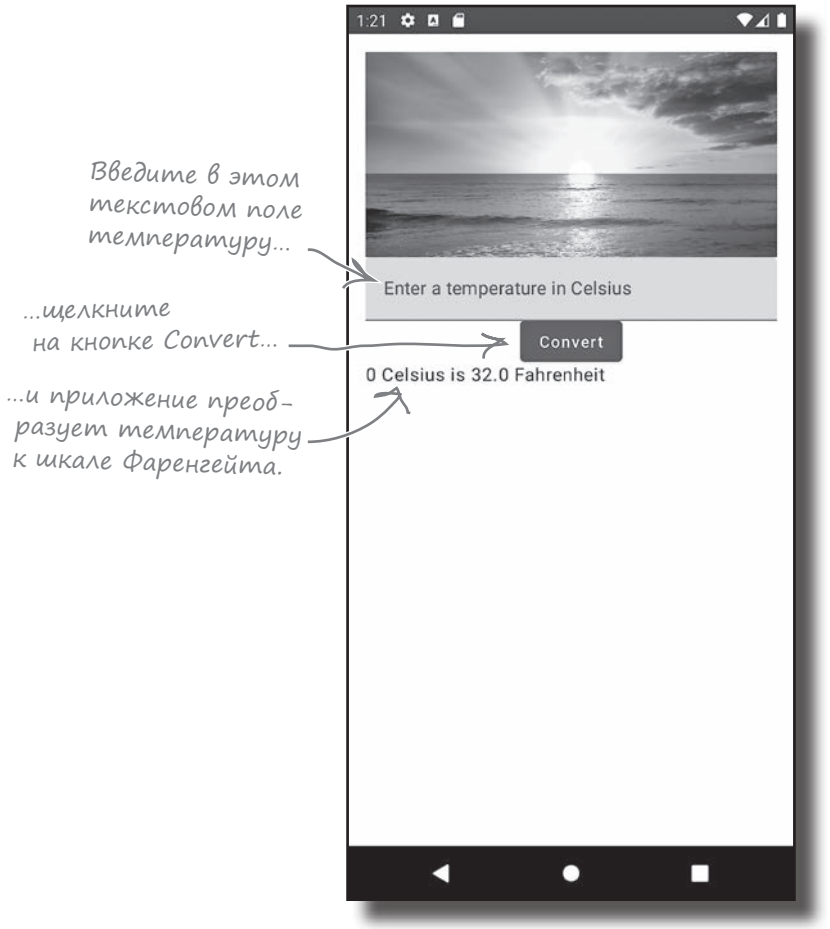
Эти сегменты кода не понадобились.

Преобразование температур в приложении

До настоящего момента мы использовали Compose для вывода текста, написали несколько компонентных функций и научились включать предварительный просмотр. Однако это еще не все.

В оставшейся части этой главы мы погрузимся в Compose и изменим только что построенное приложение, чтобы оно переводило температуру из шкалы Цельсия в шкалу Фаренгейта. Вместо того чтобы приветствовать пользователя, приложение будет запрашивать температуру по Цельсию, а затем по щелчку на кнопке переводить значение к шкале Фаренгейта.

Ниже показано, как должно выглядеть приложение; как видите, оно включает графическое изображение, текстовое поле для ввода данных, кнопку и текст:



А теперь за дело!

Добавление компонентной функции MainActivityContent

Начнем с добавления в *MainActivity.kt* новой компонентной функции с именем `MainActivityContent`, предназначенной для основного содержания активности. В эту функцию будут добавлены все компоненты, необходимые для пользовательского интерфейса `MainActivity`, и она будет вызываться из `setContent()` и `PreviewMainActivity`. В частности, это означает, что композиция активности будет отображаться как при запуске приложения, так и при предварительном просмотре.

Также из кода будет удалена компонентная функция `Hello`, которая стала лишней. Ниже приведена новая версия кода *MainActivity.kt*; обновите файл (изменения выделены жирным шрифтом):

```
...
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainActivityContent()
            Column {
                Hello("friend")
                Hello("everyone")
            }
        }
    }
}

@Composable
fun Hello(name: String) {...}

@Composable
fun MainActivityContent() {
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MainActivityContent()
    Column {
        Hello("friend")
        Hello("everyone")
    }
}
```

Используйте MainActivityContent для пользовательского интерфейса MainActivity.

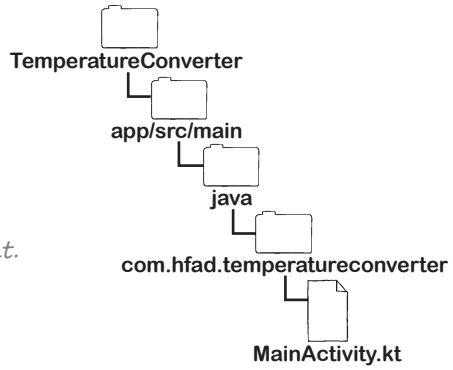
Удалите эти строки.

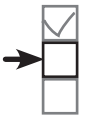
Удалите компонентную функцию Hello.

Добавьте MainActivityContent.

Включите предварительный просмотр для MainActivityContent.

Удалите эти строки.





Текст
Преобразователь
Изменение

Вывод изображения в заголовке...

Первым компонентом, добавленным в `MainActivityContent`, будет изображение, которое отображается в верхней части экрана.

Сначала убедитесь в том, что проект содержит папку *app/src/main/res/drawable* (если папки нет, создайте ее самостоятельно). Затем загрузите файл *sunrise.webp* по адресу *tinyurl.com/hfad3* и добавьте его в папку *drawable*.

В заголовке выводится изображение.

...с использованием компонента Image

Для вывода изображения в Compose используется компонент **`Image`**. Основной код выглядит так:

```
Image(
    painter = painterResource(R.drawable.sunrise),
    contentDescription = "sunrise image"
)
```

Указывает, какое изображение должно использоваться.

Описание изображения.

Компоненту `Image` должны передаваться два аргумента: `painter` и `contentDescription`.

Аргумент `painter` определяет изображение, которое должно выводиться. В данном случае значение `painterResource(R.drawable.sunrise)` выводит графический ресурс *sunrise.webp*.

Аргумент `contentDescription` содержит описание изображения, которое выводится для пользователей с ограниченными возможностями.

Существует много других необязательных аргументов, которые могут использоваться для управления внешним видом изображения и параметрами его отображения. Например, следующий код устанавливает высоту изображения 180dp, заполняет всю доступную ширину и масштабирует изображение:

```
Image(
    painter = painterResource(R.drawable.sunrise),
    contentDescription = "sunrise image",
    modifier = Modifier
        .height(180.dp)
        .fillMaxWidth(),
    contentScale = ContentScale.Crop
)
```

Модификаторы используются для определения дополнительных настроек компонентов Compose, например ширины и высоты.

Этот аргумент масштабирует изображение.

Теперь вы знаете, как добавить изображение в Compose, и мы можем добавить его в `MainActivity`.

Текст
Преобразователь
Изменение

Добавление Image в MainActivity.kt

Чтобы добавить изображение в `MainActivity`, мы определим в *MainActivity.kt* новую компонентную функцию (с именем `Header`), которая создает это изображение. Функция будет выполняться из компонентной функции `MainActivityContent`, чтобы изображение добавлялось в пользовательский интерфейс и в режим предварительного просмотра.

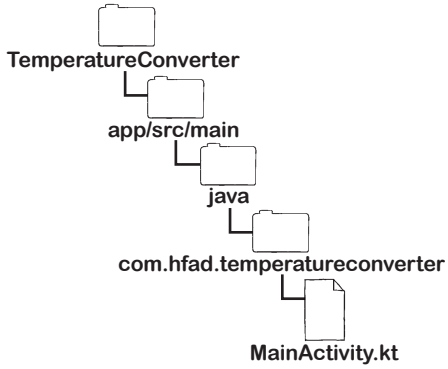
Ниже приведена новая версия *MainActivity.kt*; обновите этот файл (изменения выделены жирным шрифтом):

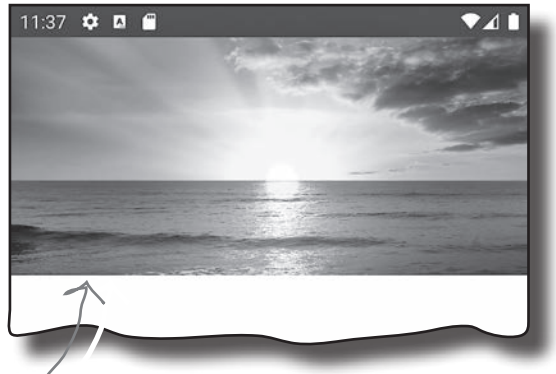
```
...
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp

...

@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}

@Composable
fun MainActivityContent() {
    Header(R.drawable.sunrise, "sunrise image")
}
...
```

Добавьте эти команды импортирования.

Добавьте компонентную функцию Header.

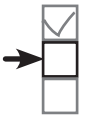
Функция Header выполняется из MainActivityContent.





Так выглядит компонентная функция Header при выполнении кода.

И это все, что необходимо знать для вывода изображения в композиции. Давайте перейдем к следующему компоненту.



Текст
Преобразователь
Изменение

Вывод температуры в виде текста

На следующем шаге мы добавим компонентную функцию (с именем `TemperatureText`), которая преобразует температуру из шкалы Цельсия в шкалу Фаренгейта и выводит результат. Эта функция будет вызываться из `MainActivityContent`, чтобы она включалась как в пользовательский интерфейс, так и в предварительный просмотр.

Код решения этой задачи вам уже знаком, поэтому обновите файл *MainActivity.kt* (изменения выделены жирным шрифтом):

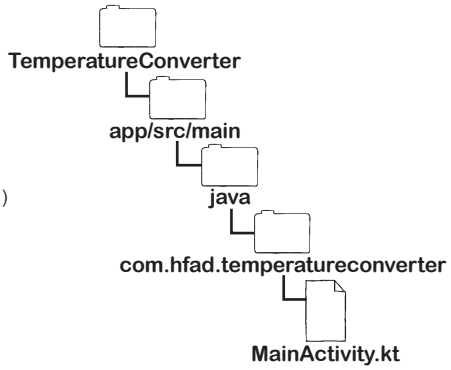
```
...
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}

@Composable
fun MainActivityContent() {
    Column {
        Header(R.drawable.sunrise, "sunrise image")
        TemperatureText(0)
    }
}
...
```

Добавляется компонентная функция TemperatureText.

Формула преобразует температуру из шкалы Цельсия в шкалу Фаренгейта.

Компоненты выстраиваются в столбец.

TemperatureText выполняется из MainActivityContent.

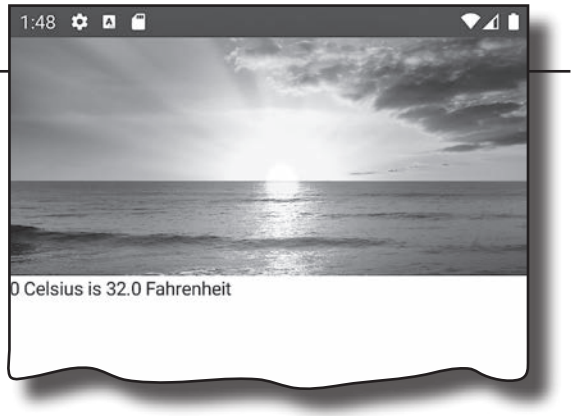


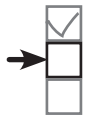
Проведем тест-драйв приложения.

Тест-драйв

При выполнении (или предварительном просмотре) приложение выводит `Image` и `Text` в столбец. Компонент `Text` правильно отображает температуру 0° по Цельсию в шкале Фаренгейта.

Мы добились того, чтобы функция `TemperatureText` работала с жестко фиксированной температурой. Обновим его новой температурой, когда пользователь щелкает на кнопке.





Текст
Преобразователь
Изменение

Использование компонента Button для добавления кнопки

Для добавления кнопок в Compose используется компонент **`Button`**. Код `Button` выглядит примерно так:

```
Button(onClick = {
    //Код, выполняемый по щелчку
}) { Text("Button Text") }
```

В onClick передается лямбда-выражение, которое определяет, что происходит по щелчку на кнопке.

Текст, выводимый на кнопке.

При использовании компонента `Button` необходимо задать два аспекта: поведение по щелчку и то, что должно отображаться на кнопке.

Поведение по щелчку задается аргументом `onClick` компонента `Button`. Он получает лямбда-выражение, которое выполняется при каждом щелчке на кнопке.

Чтобы указать, что должно отображаться на кнопке, вы используете отдельное лямбда-выражение, которое включает компонент Compose. При выполнении кода этот компонент добавляется на кнопку. Например, в приведенном выше коде `Button` передается компонент `Text`, поэтому при выполнении создается кнопка с текстом.

Создание компонентной функции ConvertButton

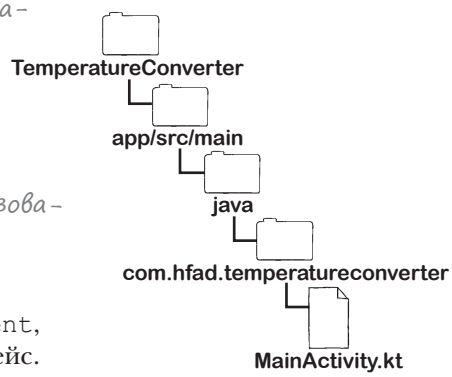
Мы добавим в приложение Temperature Converter компонент `Button`, по щелчку на котором изменяется температура, которую `TemperatureText` преобразует к шкале Фаренгейта. Для этого мы напишем новую компонентную функцию (с именем `ConvertButton`), которая выводит компонент `Button`. Также будет указано, что функция получает аргумент с лямбда-выражением, которое определяет поведение `Button` по щелчку.

Ниже показано, как выглядит код компонентной функции `ConvertButton`; через несколько страниц мы добавим его в *MainActivity.kt*:

```
@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
```

ConvertButton получает лямбда-выражение в аргументе.

Добавляет кнопку в пользовательский интерфейс.



Функция `ConvertButton` готова. Добавим ее в `MainActivityContent`, чтобы компонент `Button` был включен в пользовательский интерфейс.

ConvertButton должно передаваться лямбда-выражение

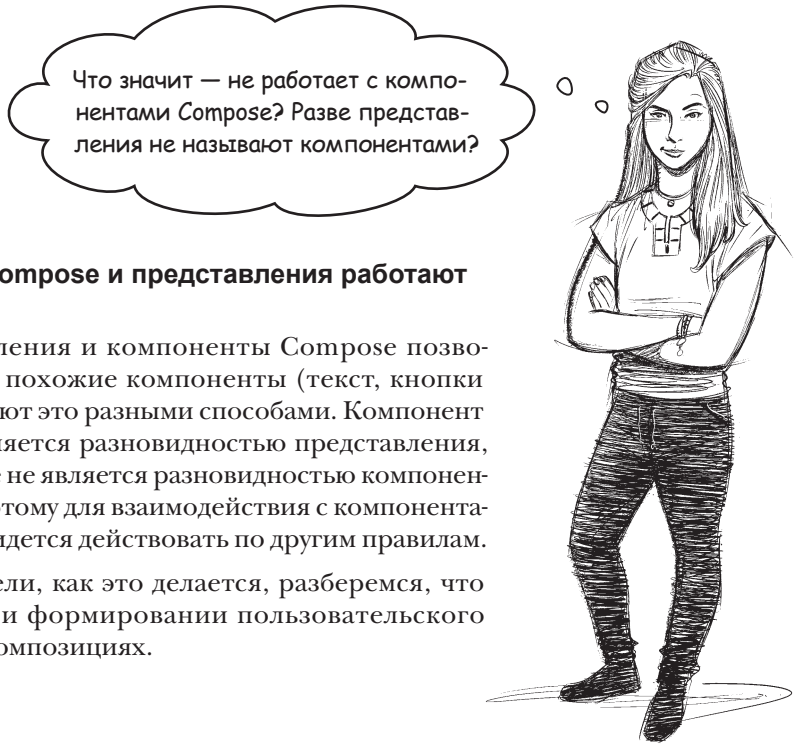
Чтобы выполнить функцию `ConvertButton` из `MainActivityContent`, необходимо передать ей лямбда-выражение, которое определяет, что должно происходить по щелчку на кнопке. Код должен выглядеть примерно так:

```
@Composable
fun MainActivityContent() {
    ...
    ConvertButton {
        //Код, выполняемый по щелчку на кнопке
    }
    ...
}
```

Когда пользователь щелкает на кнопке `ConvertButton`, текст, отображаемый в `TemperatureText`, должен обновляться. Если бы компонент `TemperatureText` был представлением, то текст можно было бы обновить кодом следующего вида:

```
binding.textView.text = "This is the new text"
```

Такой подход работает для представлений, но не для *компонентов Compose*. С компонентами Compose придется действовать иначе.



Компоненты Compose и представления работают по-разному.

Хотя представления и компоненты Compose позволяют выводить похожие компоненты (текст, кнопки и т. д.), они делают это разными способами. Компонент Compose не является разновидностью представления, а представление не является разновидностью компонента Compose, поэтому для взаимодействия с компонентами Compose придется действовать по другим правилам.

Чтобы вы увидели, как это делается, разберемся, что происходит при формировании пользовательского интерфейса в композициях.

Compose под увеличительным стеклом

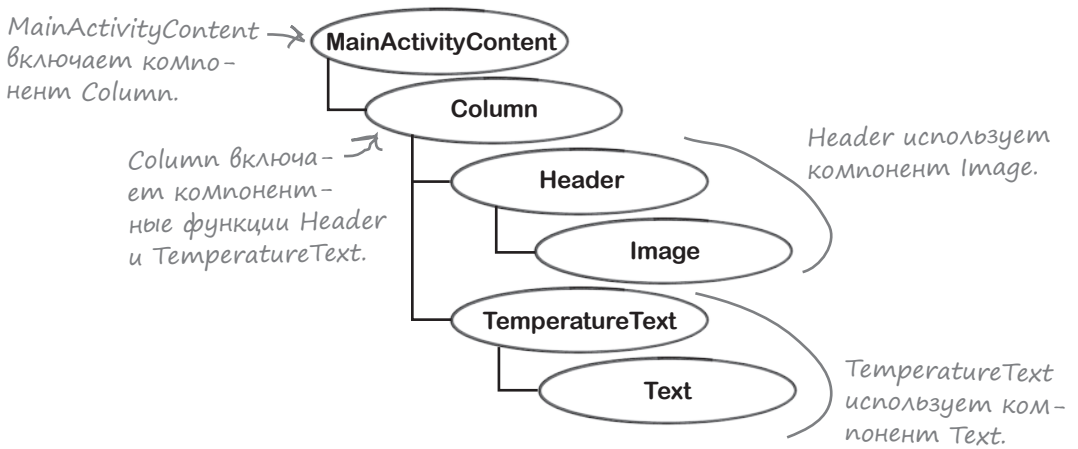
При построении пользовательского интерфейса Compose сначала создает иерархическое дерево всех вызываемых компонентов. Например, при выполнении следующего кода:

```
@Composable
fun MainActivityContent() {
    Column {
        Header(R.drawable.sunrise, "sunrise image")
        TemperatureText(0)
    }
}
```

Header выводит Image.

TemperatureText выводит Text.

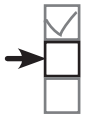
Compose строит дерево компонентов, которое выглядит так:



После того как компонент Compose будет сформирован, он перерисовывается (или **перестраивается**) при изменении его входных данных. Например, `TemperatureText` определяется следующим кодом:

```
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}
```

Так как `TemperatureText` получает один аргумент с именем `celsius`, он будет перестраиваться только при обновлении значения `celsius`. Если вы хотите, чтобы компонент `TemperatureText` был перерисован с новым текстом, необходимо обновить значение его аргумента.



Необходимо изменить значение аргумента TemperatureText

Как было показано ранее, компоненты Compose перестраиваются при изменении значений, от которых они зависят. Это означает, что если вы хотите, чтобы в `TemperatureText` выводился разный текст, когда пользователь щелкает на кнопке `ConvertButton`, передаваемое `ConvertButton` лямбда-выражение должно обновить значение аргумента `TemperatureText`.

Компоненты Compose перестраиваются при изменении любых из их входных значений.

Для этого мы добавим в `MainActivityContent` новую переменную `celsius` и передадим ее значение `TemperatureText`. Когда пользователь щелкает на компоненте `ConvertButton`, значение `celsius` будет обновлено, чтобы компонент `TemperatureText` был перестроен.

Использование remember для сохранения значения в памяти

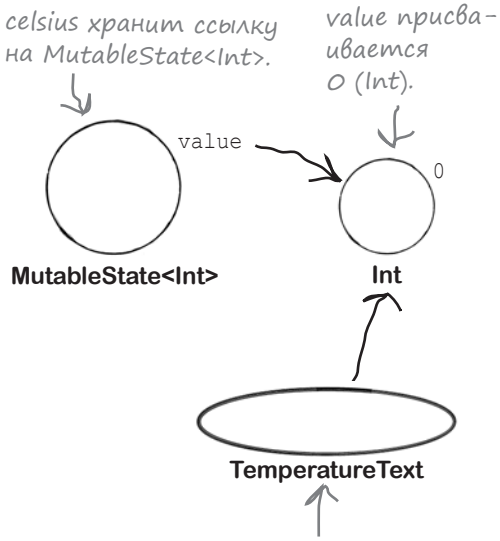
Переменная `celsius` определяется включением следующего кода в `MainActivityContent`:

```
val celsius = remember { mutableStateOf(0) }
```

Сохраняет значение celsius в памяти компонента Compose.

Эта команда создает объект с типом **`MutableState`**, присваивает ему значение `0` и сохраняет в памяти. Работа с `celsius` напоминает работу с объектом Live Data. Каждый раз, когда ему присваивается новое значение, все использующие его компоненты Compose получают оповещения и перестраиваются.

Для сохранения объекта в памяти используется функция **`remember`**. `remember` сохраняет объект в тот момент, когда вызывающая ее компонентная функция — в данном случае `MainActivityContent` — проходит первое построение, и «забывает» его при удалении компонента Compose из пользовательского интерфейса. Это может произойти, когда пользователь поворачивает устройство и активность — вместе с ее пользовательским интерфейсом — уничтожается и создается заново.



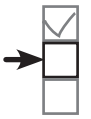
Как и в случае с объектами `MutableLiveData`, значение объекта `MutableState` задается обновлением его свойства **`value`**. Например, чтобы по щелчку на компоненте `ConvertButton` переменной `celsius` присваивалось значение 20, используется следующий код:

```
ConvertButton { celsius.value = 20 }
```

Чтобы компонент `TemperatureText` реагировал на изменение значения, в аргументе следует передать `celsius.value`:

```
TemperatureText(celsius.value)
```

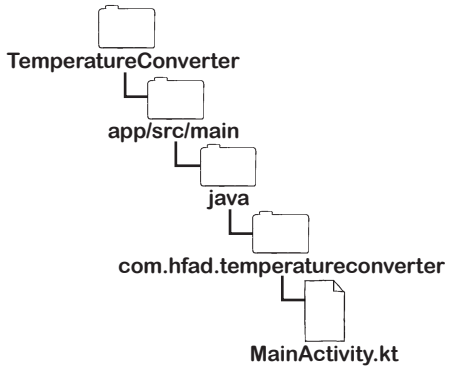
При каждом обновлении `celsius.value` компонент `TemperatureText` перестраивается с новым значением, что приводит к изменению отображаемого текста. Полный код будет приведен через пару страниц.



Текст
Преобразователь
Изменение

Полный код MainActivity.kt

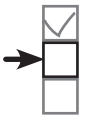
Ниже приведен полный код *MainActivity.kt*; обновите файл (изменения выделены жирным шрифтом):




```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import androidx.compose.material.Button
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainActivityContent()
        }
    }
}

@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}
```

Импортируйте эти дополнительные классы.

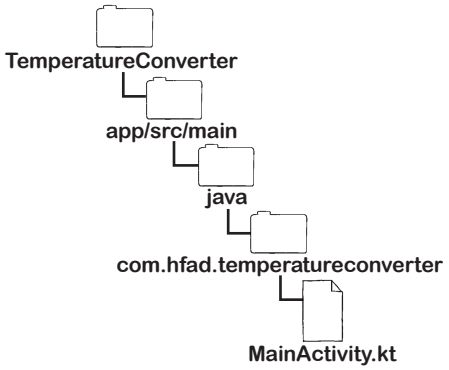
Продолжение на следующей странице.



MainActivity.kt (продолжение)

```
@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
```

Добавьте компонентную функцию ConvertButton.



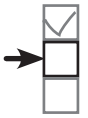

```
@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}

@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }

    Column {
        Header(R.drawable.sunrise, "sunrise image")
        ConvertButton { celsius.value = 20 }
        TemperatureText(celsius.value)
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MainActivityContent()
}
```

Добавьте переменную celsius.

Кнопка ConvertButton должна обновлять celsius по щелчку.

При вызове TemperatureText передается celsius.value, чтобы текст изменялся при обновлении celsius.

Давайте разберемся, что происходит при выполнении кода, и проведем тест-драйв приложения.

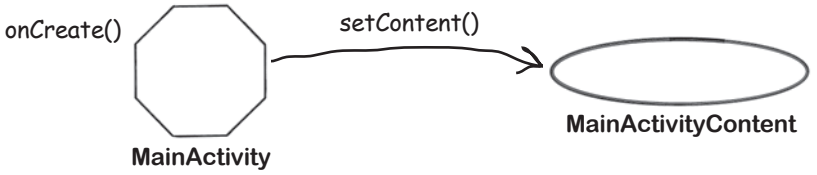


Текст
Преобразователь
Изменение

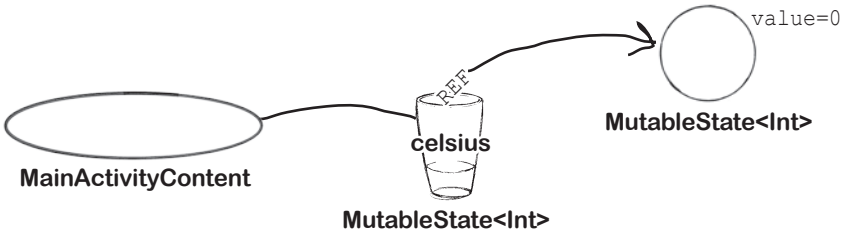
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

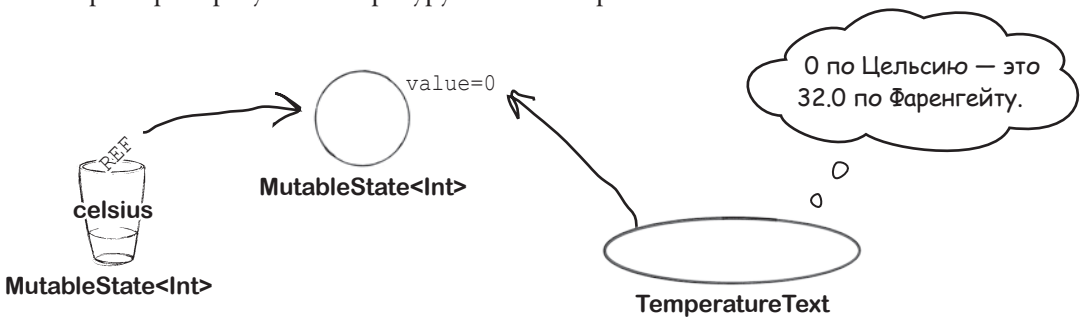
1. **Запускается активность MainActivity, и выполняется ее метод onCreate().** Он вызывает метод `setContent()`, который выполняет компонентную функцию `MainActivityContent`.



2. **MainActivityContent создает переменную MutableState<Int> с именем celsius, присваивает ей значение 0 и сохраняет в памяти.**

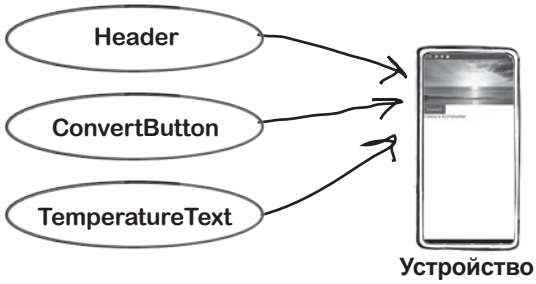


3. **MainActivityContent выполняет компонентные функции Header, ConvertButton и TemperatureText.**
Значение `celsius` передается компонентной функции `TemperatureText`, которая преобразует температуру к шкале Фаренгейта.

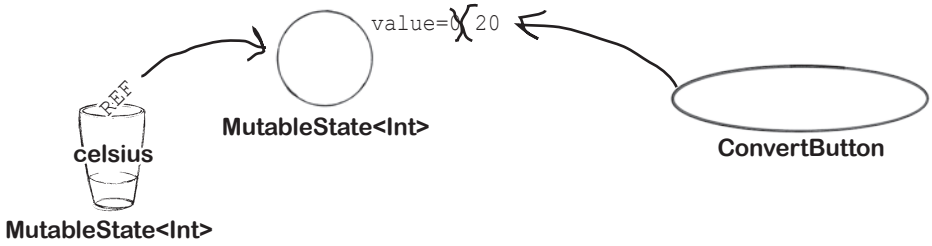


История продолжается

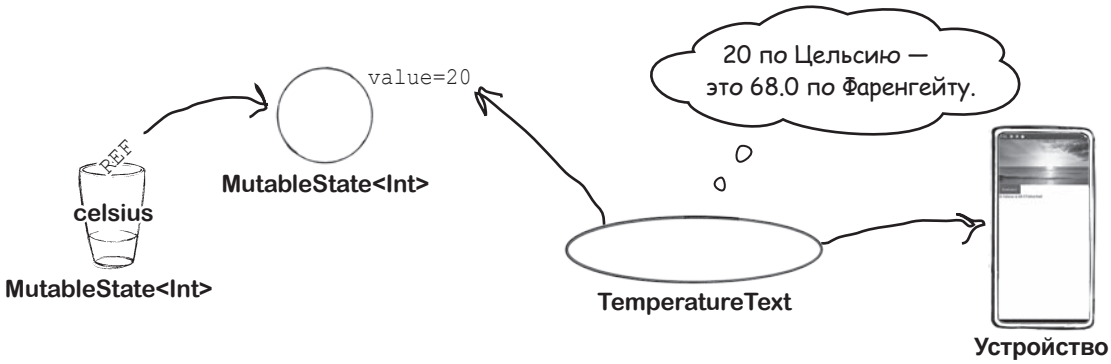
4 **Компонентные функции Header, ConvertButton и TemperatureText добавляют в пользовательский интерфейс изображение, кнопку и текст.**



5 **Пользователь щелкает на компоненте ConvertButton в пользовательском интерфейсе.**
Свойству value объекта `celsius` присваивается значение 20.

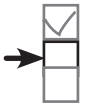


6 **TemperatureText перестраивается.**
Новое значение `celsius` преобразуется к шкале Фаренгейта, а результат преобразования выводится.



Проведем тест-драйв приложения.

Тест-драйв



При запуске приложения отображается активность `MainActivity`. Она включает изображение, кнопку и текст, который представляет значение 0° по Цельсию по шкале Фаренгейта.

Если щелкнуть на кнопке Convert, текст обновляется значением, соответствующим 20° по Цельсию, по шкале Фаренгейта.

Следующее, что необходимо сделать, — предоставить пользователю возможность вводить произвольную температуру. Вскоре мы сделаем это, но сначала проверьте свои силы в упражнении на следующей странице.

Часто Задаваемые Вопросы

В: Так вы говорите, `remember` сохраняет объект в памяти?

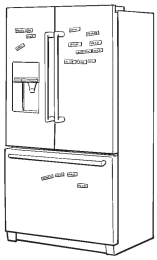
О: Да! Функция `remember` сохраняет объект в памяти в тот момент, когда вызывающий ее компонент Compose строится впервые, и забывает о нем, когда компонент удаляется из пользовательского интерфейса. Это может произойти, когда пользователь поворачивает экран устройства, потому что Android уничтожает активность и создает ее заново вместе с другими компонентами.

В: Можно ли сохранить состояние компонента Compose при повороте экрана?

О: Для простых значений можно воспользоваться функцией `rememberSaveable` вместо `remember`. При этом автоматически сохраняется любое значение, которое может быть сохранено в `Bundle`. Другое возможное решение — воспользоваться моделью представления. В следующей главе мы покажем, как использовать компоненты Compose с моделью представления.

В: Я видел такой синтаксис: `var x by remember { ... }`. Что он делает?

О: То же самое, что `val x = remember { ... }` за исключением того, что `x.value` можно заменить в коде на `x`. Если вы используете это решение, убедитесь в том, что в приложении импортируются `androidx.compose.runtime.getValue` и `androidx.compose.runtime.setValue`; без этого код не будет компилироваться.



Развлечения с магнитами

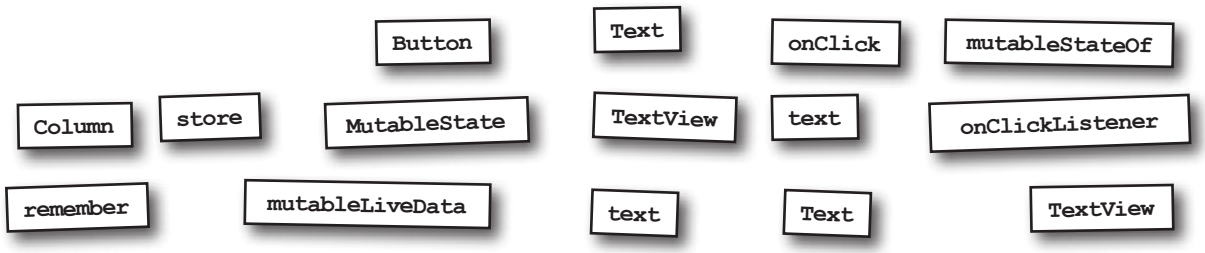
Кто-то выложил на холодильнике код новой компонентной функции с именем `ChangeHello`. Функция выводит текст «Hello friend» и кнопку, по щелчку на которой текст заменяется на «Hello everyone».

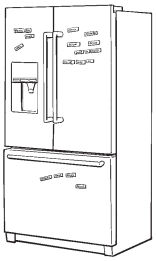
К сожалению, кошка сбросила на пол некоторые магниты. Сможете ли вы восстановить код функции?

При выполнении компонентная функция строит этот пользовательский интерфейс.

```
@Composable
fun ChangeHello() {

    val name = ..... { ..... ("friend") }

  ..... {

      ..... ("Hello ${name.value}")

      ..... (

          ..... = { name.value = "everyone" }

        ) {
          ..... ("Say hello everyone")
        }
    }
}
```

Использовать все магниты не обязательно.





Развлечения с магнитами. Решение

Кто-то выложил на холодильнике код новой компонентной функции с именем `ChangeHello`. Функция выводит текст «Hello friend» и кнопку, по щелчку на которой текст заменяется на «Hello everyone».

К сожалению, кошка сбросила на пол некоторые магниты. Сможете ли вы восстановить код функции?

При выполнении компонентная функция строит этот пользовательский интерфейс.

```
@Composable
fun ChangeHello() {

    val name = remember { mutableStateOf("friend") }

    Column {
        Text("Hello ${name.value}")
        Button(
            onClick = { name.value = "everyone" }
        ) {
            Text("Say hello everyone")
        }
    }
}
```

Сохраняет значение переменной name в памяти.

Компоненты размещаются в столбец.

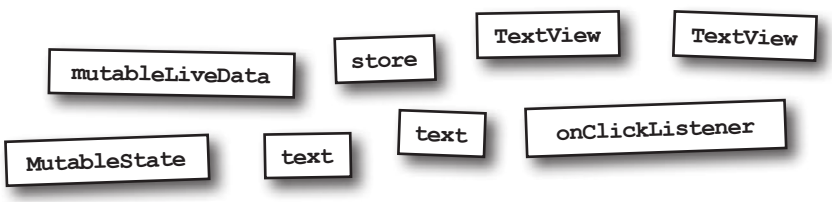
Добавляет текст в пользовательский интерфейс.

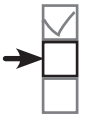
Добавляет кнопку.

Изменяет значение name по щелчку на кнопке.

Этот текст выводится на кнопке.

Эти магниты остались неиспользованными.

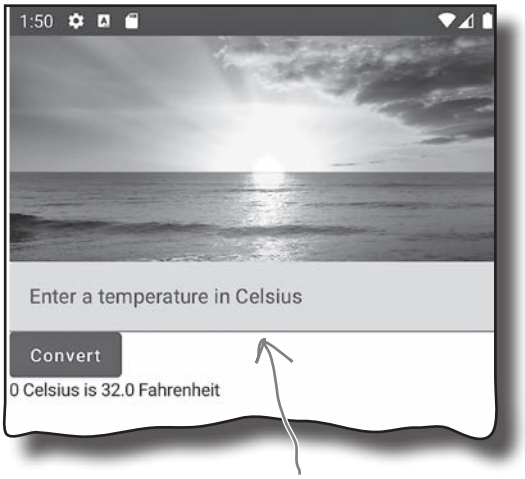




Ввод температуры

До настоящего момента мы строили версию приложения Temperature Converter, которая преобразует жестко фиксированную температуру по Цельсию к шкале Фаренгейта. Когда пользователь щелкает на кнопке, исходная температура преобразуется в другое значение.

Но *на самом деле* нужно, чтобы пользователь мог ввести любую интересующую его температуру, поэтому на следующем шаге в пользовательский интерфейс будет добавлено текстовое поле. Пользователь вводит температуру по Цельсию в текстовом поле; по щелчку на кнопке приложение преобразует введенную температуру к шкале Фаренгейта и выводит результат:



Пользователь вводит температуру по Цельсию в компоненте TextField. Когда он щелкает на кнопке, приложение преобразует значение к шкале Фаренгейта.

Добавление компонента TextField

Чтобы добавить текстовое поле в интерфейс, мы воспользуемся компонентом `TextField`. Его можно рассматривать как аналог `EditText` в мире Compose.

Код добавления `TextField` выглядит примерно так:

```
@Composable
fun ExampleTextField() {
    val text = remember { mutableStateOf("") }
    TextField(
        value = text.value,
        onValueChange = { text.value = it },
        label = { Text("What is your name?") }
    )
}
```

Сохраняет состояние TextField.

Добавляет TextField.

Значение переменной text используется для заполнения TextField.

Когда пользователь обновляет TextField, должно обновляться значение text.value.

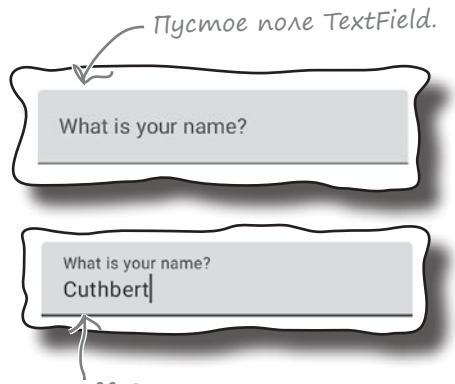
Текст описания поля TextField.

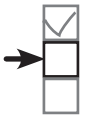
Атрибут `value` используется для получения значения `TextField`. В данном примере для сохранения этого значения в памяти используется переменная `MutableState` с именем `text`.

Атрибут `onValueChange` использует лямбда-выражение для обновления значения переменной `text` при вводе текста пользователем.

Атрибут `label` определяет описание `TextField`. Этот текст отображается в пустом поле `TextField` и исчезает, когда пользователь вводит текст.

Теперь вы знаете, как выглядит код текстового поля, и мы можем добавить компонент в приложение.





Текст
Преобразователь
Изменение

Добавление TextField в компонентную функцию

Чтобы добавить компонент `TextField` в пользовательский интерфейс, мы создадим новую компонентную функцию `EnterTemperature`, которая будет вызываться из `MainActivityContent`.

Код функции `EnterTemperature` выглядит так:

```
@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}
```

Передаются аргументы для значения TextField и лямбда-выражения onValueChange.

Компонент TextField имеет максимально возможную ширину.

Как видите, функция получает два аргумента: строку со значением `TextField` и лямбда-выражение, которое определяет, что должно происходить при вводе нового значения пользователем.

Вызов функции в MainActivityContent

Функция `MainActivityContent` должна передавать оба аргумента `EnterTemperature` при выполнении функции, поэтому мы используем новый объект `MutableState` с именем `newCelsius` для представления значения, которое обновляется при вводе текста пользователем.

Также мы изменим лямбда-выражение, передаваемое `ConvertButton`, чтобы оно обновляло значение `celsius` при вводе пользователем допустимого значения `Int`; это приводит к перестраиванию `TemperatureText` при вводе пользователем новой температуры.

Код приводится ниже; мы займемся обновлением *MainActivity.kt* на следующей странице:

```
@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }
    ...
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        ConvertButton {
            newCelsius.value.toIntOrNull()?.let {
                celsius.value = it
            }
        }
    ...
}
```

Сохраняет температуру, введенную пользователем.

Значение newCelsius.value используется для температуры, введенной пользователем, и обновляется при вводе нового значения.

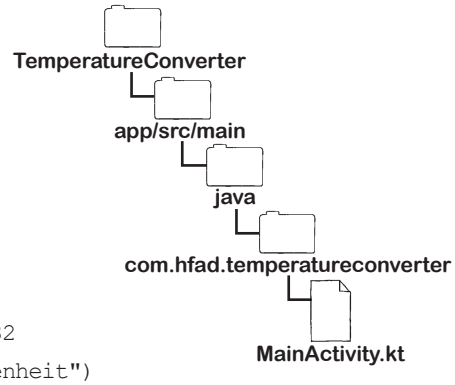
Если пользователь вводит действительное значение Int, обновить celsius.value. Это приводит к перестраиванию компонента TemperatureText.

Текст
Преобразователь
Изменение

Полный код MainActivity.kt

Ниже приведен код *MainActivity.kt*; обновите файл (изменения выделены жирным шрифтом):

```
...
import androidx.compose.material.TextField   ← Импортируйте этот класс.

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MainActivityContent()
        }
    }
}

@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}

@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
```

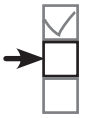


Добавьте компонентную функцию EnterTemperature.

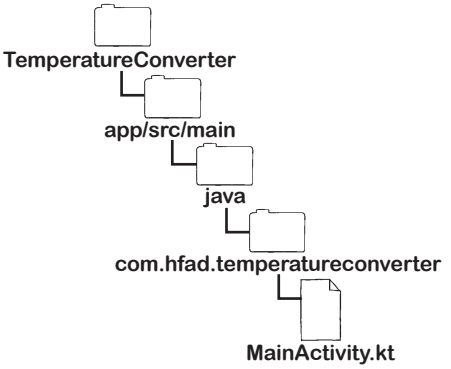

```
@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}
```

Продолжение на следующей странице.

MainActivity.kt (продолжение)




```
@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}

@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }

    Column {
        Header(R.drawable.sunrise, "sunrise image")
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        ConvertButton {
            newCelsius.value.toIntOrNull()?.let {
                celsius.value = 20 it
            }
        }
        TemperatureText(celsius.value)
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MainActivityContent()
}
```



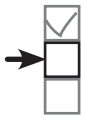
Добавьте переменную newCelsius, которая используется для состояния EnterTemperature.

EnterTemperature добавляется в пользовательский интерфейс.

Эти строки тоже необходимо добавить.

Если newCelsius.value содержит значение типа Int, присвоить его celsius.value.

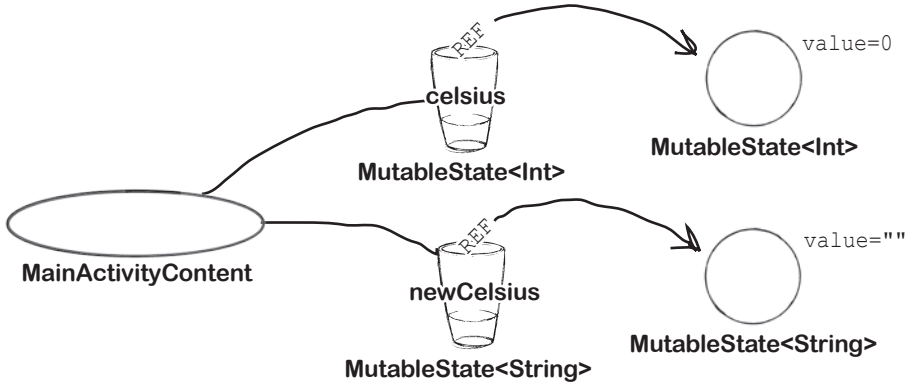
Посмотрим, что происходит при выполнении кода.



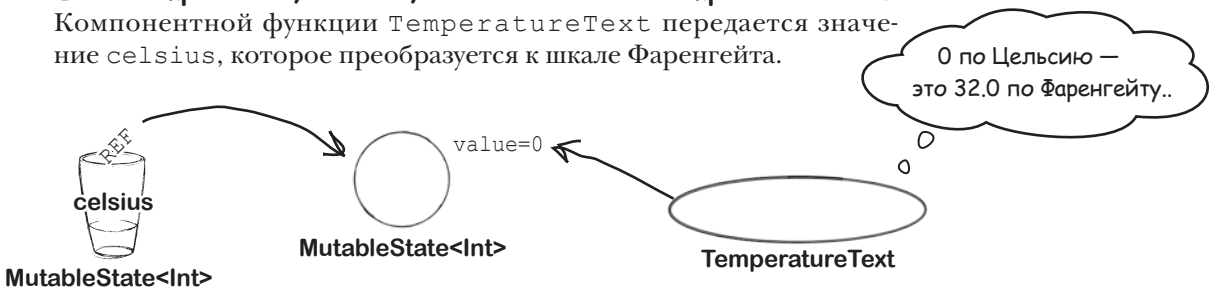
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

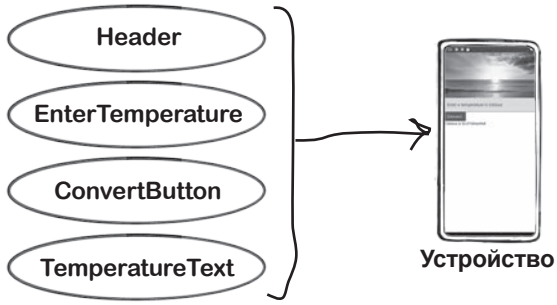
1. **При выполнении активности MainActivityContent создается переменная MutableState<Int> с именем celsius и переменная MutableState<String> с именем newCelsius.**
`celsius` присваивается `0`, `newCelsius` присваивается `""`, и оба значения сохраняются в памяти.



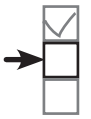
2. **MainActivityContent выполняет компонентные функции EnterTemperature, Header, ConvertButton и TemperatureText.**
Компонентной функции `TemperatureText` передается значение `celsius`, которое преобразуется к шкале Фаренгейта.



3. **Компонентные функции Header, EnterTemperature, ConvertButton и TemperatureText добавляют в пользовательский интерфейс изображение, текстовое поле, кнопку и текст.**

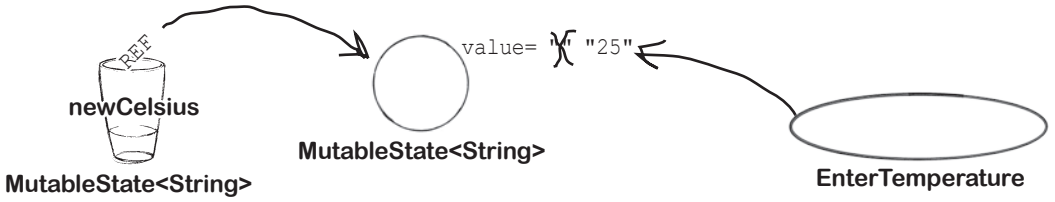


История продолжается

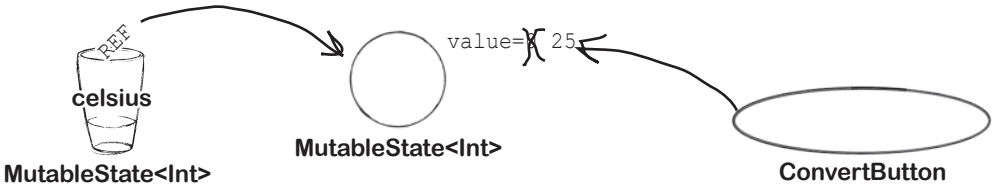


Текст
Преобразователь
Изменение

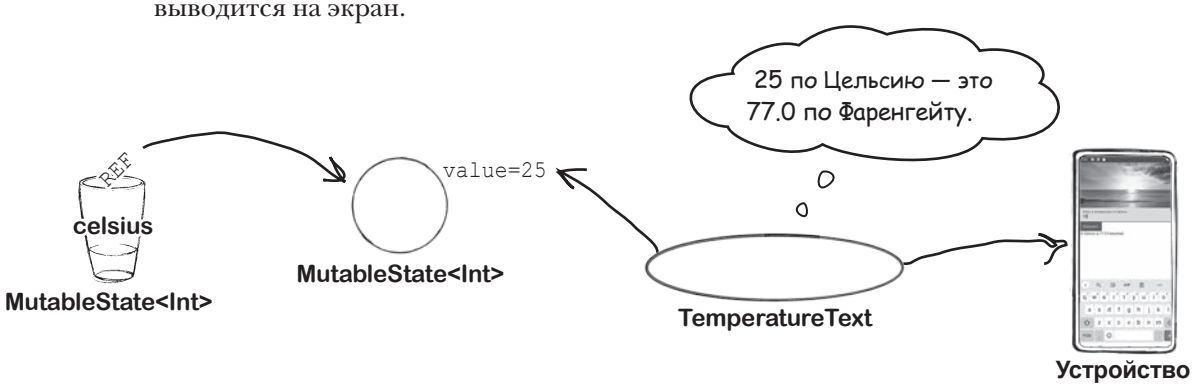
4 **Пользователь вводит новое значение в EnterTemperature (в данном примере "25").**
`EnterTemperature` присваивает это значение `newCelsius`.



5 **Пользователь щелкает на компоненте ConvertButton.**
Значение `newCelsius` преобразуется в `Int` и присваивается `celsius.value`.



6 **TemperatureText перестраивается.**
Новое значение `celsius` преобразуется к шкале Фаренгейта, а результат выводится на экран.

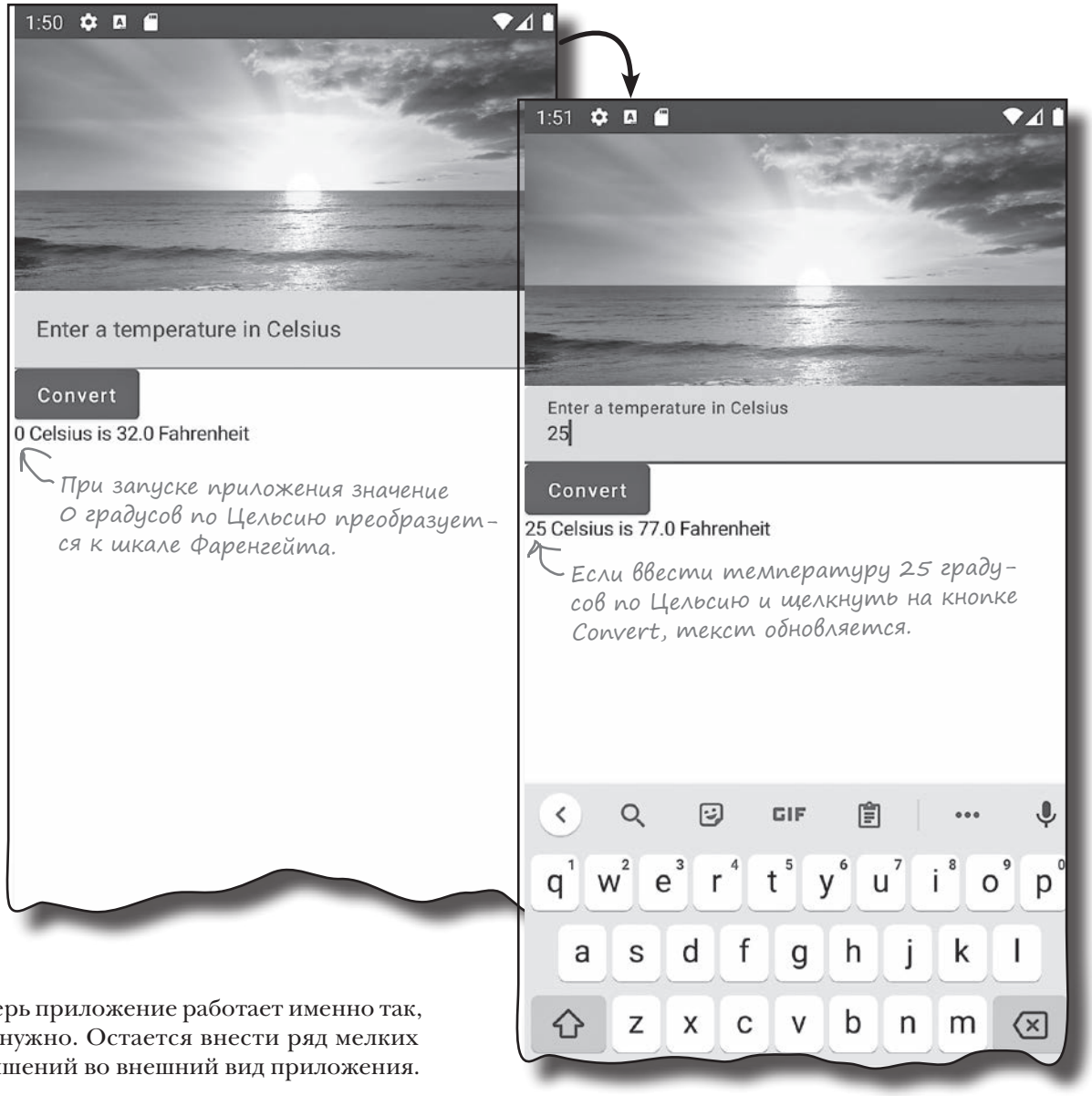


Проведем тест-драйв приложения.

Тест-драйв

При запуске приложения `MainActivity` отображает текстовое поле.

Если ввести температуру и щелкнуть на кнопке Convert, приложение преобразует температуру по Цельсию к шкале Фаренгейта и обновляет текстовое сообщение с температурой.



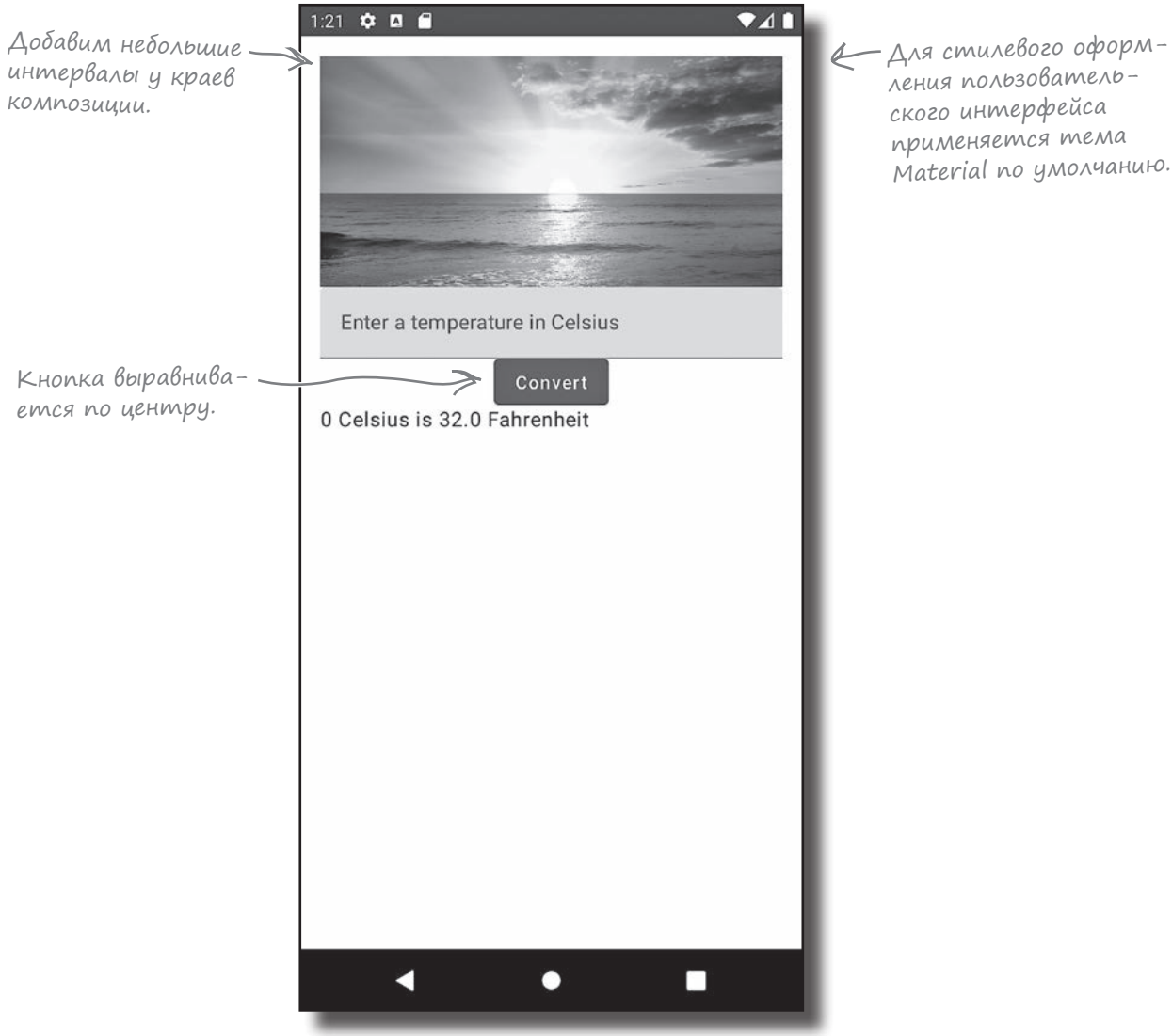
Теперь приложение работает именно так, как нужно. Остается внести ряд мелких улучшений во внешний вид приложения.

Текст
Преобразователь
Изменение

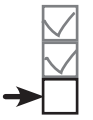
Настройка внешнего вида приложения

Нам предстоит изменить приложение так, чтобы оно отделялось от краев экрана отступами, а UI-компоненты (например, кнопки) были выровнены горизонтально по центру. Мы также применим стилевое оформление к компонентным функциям, чтобы они использовали тему Material по умолчанию.

Новая версия приложения будет выглядеть так:



Эти изменения будут внесены на нескольких ближайших страницах. Начнем с добавления отступов у краев пользовательского интерфейса.

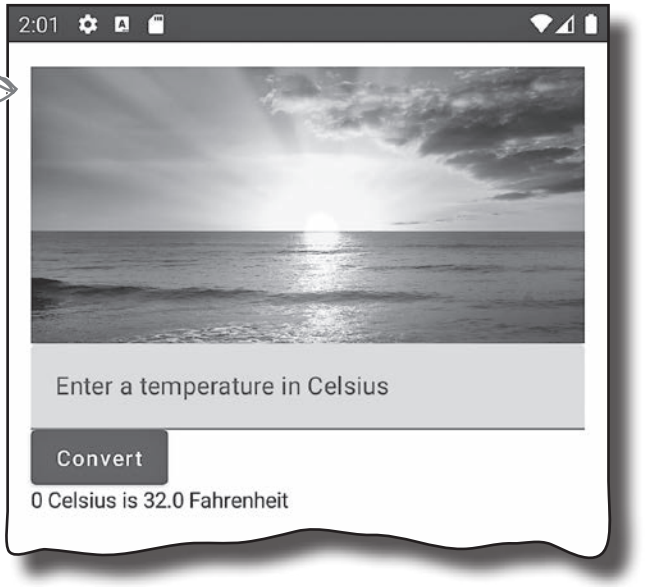


Текст
Преобразователь
Изменение

Добавление отступов к компоненту Column

Мы хотим добавить отступы между краями экрана и компонентами приложения, чтобы оно выглядело так:

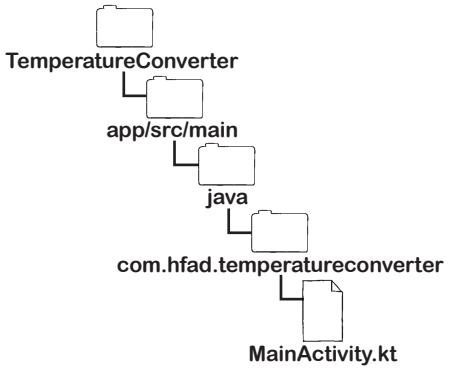
Вокруг краев пользовательского интерфейса будут добавлены отступы размером 16dp.



Отступы будут применяться к компоненту `Column`. Применение отступов к компонентам имеет такой же эффект, как с представлениями; вокруг краев компонента появляется дополнительное свободное пространство.

Для добавления отступов к компонентам используются **модификаторы** — объекты **`Modifier`**, позволяющие добавлять дополнительное поведение к компонентам Compose. Например, для добавления отступов 16dp к компоненту `Column` используется следующий код:

```
Column(modifier = Modifier.padding(16.dp)) {
    ...
}
```



Модификаторы в высшей степени гибки. Например, при написании компонентной функции `Header` мы использовали `Modifier` для задания высоты и ширины `Image`:

```
Image(
    ...
    modifier = Modifier.height(180.dp).fillMaxWidth()
)
```

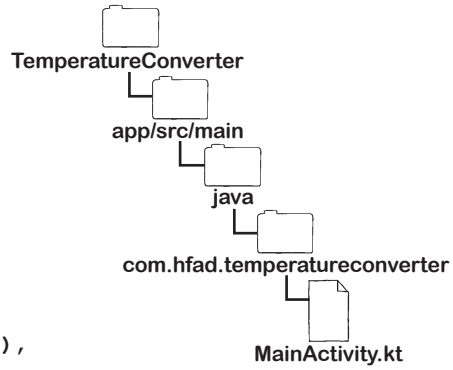
Отступы будут добавлены к компоненту `Column` приложения через несколько страниц. Но сначала разберемся, как выровнять кнопку по центру.

Выравнивание компонентов по центру Column или Row

Чтобы выровнять по центру один или несколько компонентов Compose, например `Button` или `Text`, можно воспользоваться компонентами `Column` и `Row`. Есть пара возможных решений в зависимости от того, что вы хотите сделать.

Выравнивание по центру всего содержимого Column

Если вы хотите разместить компоненты внутри компонента `Column`, чтобы все они были выровнены по центру, для этого можно назначить `Column` максимально возможную ширину, а затем задать его аргумент **`horizontalAlignment`**.



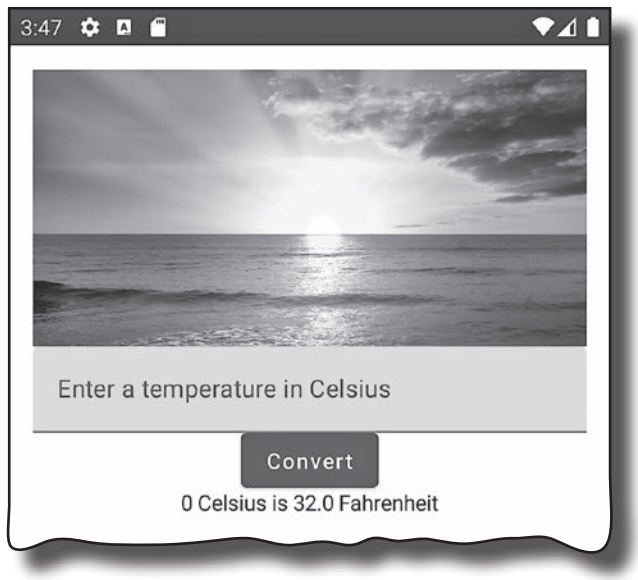
Например, для горизонтального выравнивания по центру всех компонентов в приложении Temperature Converter в `Column` добавляется следующий код (выделенный жирным шрифтом):

```
@Composable
fun MainActivityContent() {
    ...
    Column(modifier = Modifier.padding(16.dp).fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ...
    }
}
```

Этот код осуществляет горизонтальное выравнивание по центру всего содержимого столбца.

В результате приложение выглядит так:

Все компоненты выравниваются по центру горизонтально. Для Image и TextField эффект менее заметен, потому что они настроены на заполнение максимальной ширины.



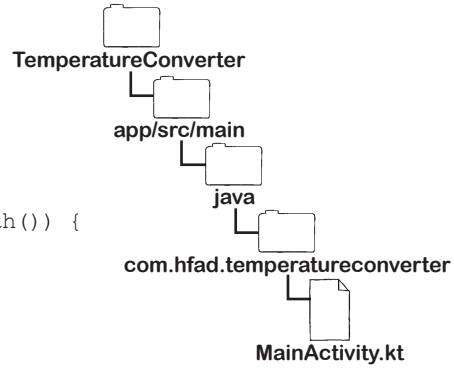
А если вы хотите выровнять по центру только один из компонентов?

Текст
Преобразователь
Изменение

Выравнивание по центру содержимого одного компонента

Если вы хотите выполнить горизонтальное выравнивание по центру только одного компонента, заключите его в `Row`. Измените компонент `Row`, назначив ему максимально возможную ширину, после чего задайте его атрибут **`horizontalArrangement`**.

Например, для компонента `ConvertButton` включение в `Row` и горизонтальное выравнивание по центру могут выполняться следующим образом:

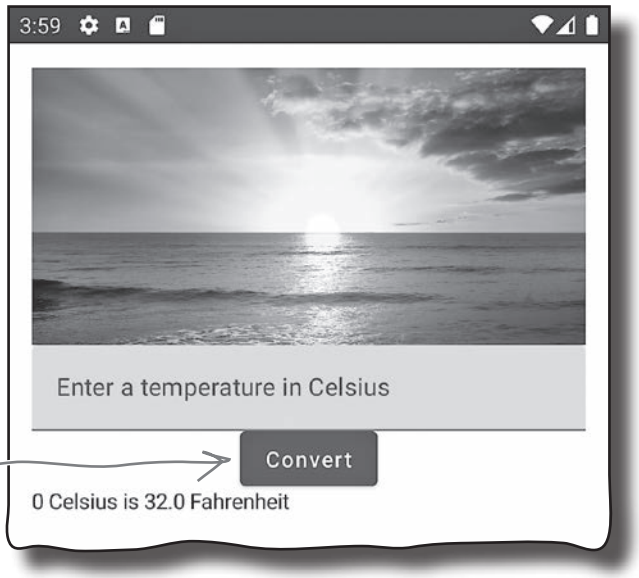



```
@Composable
fun MainActivityContent() {
    ...
    Column(modifier = Modifier.padding(16.dp).fillMaxWidth()) {
        ...
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            ConvertButton { ... }
        }
        ...
    }
}
```

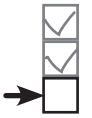
Здесь горизонтальное выравнивание по центру применяется только к компоненту ConvertButton.

Кнопка размещается в центре:

На этот раз по центру выравнивается только кнопка.



Итак, компоненты `Column` и `Row` могут использоваться для выравнивания и размещения компонентов. Но прежде чем обновлять *MainActivity.kt*, осталось рассмотреть еще один вопрос: применение тем к компонентам.



Снова о применении тем

Как вы узнали в главе 8, темы определяют общие особенности внешнего вида приложения. Например, назначение темы позволяет убрать панель приложения по умолчанию или изменить цвета приложения.

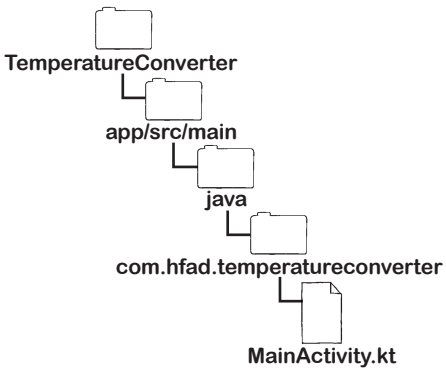
Вы уже знаете, что тему можно определить в одном из ресурсных файлов приложения, который указывается в *AndroidManifest.xml*. Тем самым вы применяете тему к приложению, включая все его представления. Однако при этом тема не распространяется на его *компоненты Compose*. Чтобы применить к ним стилевое оформление, необходимо действовать иначе.

Применение темы к компонентам Compose

Если вы хотите применить тему к композиции, придется использовать код Kotlin. Например, в следующем коде используется тема `MaterialTheme` и компоненты `Surface` для применения темы к `MainActivityContent`:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface {
                    MainActivityContent()
                }
            }
        }
    }
}
```

Применяет тему Material по умолчанию к MainActivityContent.



`MaterialTheme` используется для применения темы Material по умолчанию к компонентам Compose. В данном случае она применяется к `MainActivityContent` и компонентным функциям, которые вызываются этой функцией (таким, как `TemperatureText` и `ConvertButton`).

В приведенном коде также встречается компонент **`Surface`**, который используется для стилевого оформления поверхностей. В частности, он используется для применения 3D-эффектов и теней.

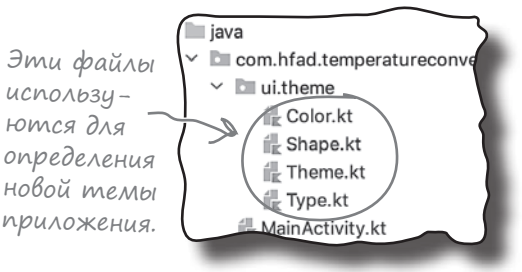
Если вы захотите переопределить тему `MaterialTheme` по умолчанию, определите новую тему в коде Kotlin. Android Studio обычно добавляет код темы при создании нового проекта с активностью Compose; рассмотрим этот код, чтобы понять, что он делает.

Текст
Преобразователь
Изменение

Android Studio включает дополнительный код темы

При создании проекта Temperature Converter среда Android Studio добавила дополнительные файлы Kotlin для определения новой темы. Это файлы с именами *Color.kt*, *Shape.kt*, *Theme.kt* и *Type.kt*, и находятся они в папке *app/src/main/java* из пакета *com.hfad.temperatureconverter.ui.theme*.

Самый важный из них — файл *Theme.kt*, определяющий новую тему для приложения. Тема называется `TemperatureConverterTheme`, а ее код выглядит так:



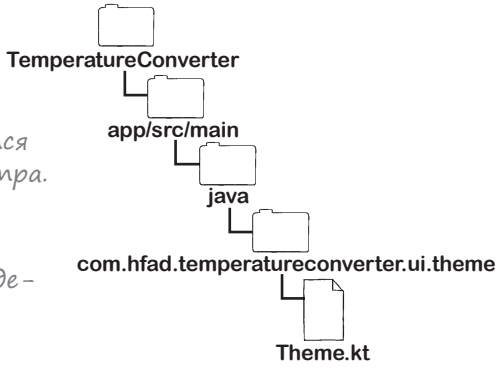
Эти файлы используются для определения новой темы приложения.

```
package com.hfad.temperatureconverter.ui.theme

import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material.MaterialTheme
import androidx.compose.material.darkColors
import androidx.compose.material.lightColors
import androidx.compose.runtime.Composable

private val DarkColorPalette = darkColors(
    primary = Purple200,
    primaryVariant = Purple700,
    secondary = Teal200
)

private val LightColorPalette = lightColors(
    primary = Purple500,
    primaryVariant = Purple700,
    secondary = Teal200

    /* Другие переопределяемые цвета по умолчанию
    background = Color.White,
    surface = Color.White,
    onPrimary = Color.White,
    onSecondary = Color.Black,
    onBackground = Color.Black,
    onSurface = Color.Black,
    */
)
```

Не беспокойтесь, если ваш код отличается от нашего. Мы просто приводим код Theme.kt, сгенерированный средой Android Studio; у вас может быть сгенерирован другой код в зависимости от версии Android Studio.

В этом коде определяется светлая и темная палитра.

Сами цвета определяются в Color.kt.



Продолжение на следующей странице.

Текст
Преобразователь
Изменение

Theme.kt (продолжение)

```
@Composable
fun TemperatureConverterTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable() () -> Unit
) {
    val colors = if (darkTheme) {
        DarkColorPalette
    } else {
        LightColorPalette
    }

    MaterialTheme(
        colors = colors,
        typography = Typography,
        shapes = Shapes,
        content = content
    )
}
```

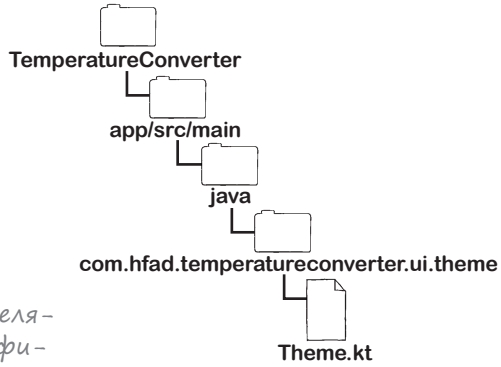
Компонентная функция для темы приложения.

Проверяем, использует устройство светлую или темную тему.

Назначение цветовой палитры.

Шрифтовые параметры темы определяются в Type.kt.

В файле Shape.kt определяются геометрические фигуры, используемые темой.



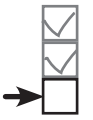
Как видите, тема переопределяет цвета, шрифты и геометрические фигуры `MaterialTheme`. Если вы захотите настроить какие-либо из этих аспектов, это можно сделать обновлением файлов в пакете *.ui.theme*.

Применение темы

После того как тема будет определена, ее можно применить к компонентам приложения. Например, для применения темы `TemperatureConverterTheme` к `MainActivityContent` используется следующий код:

```
...
    setContent {
        TemperatureConverterTheme {
            Surface {
                MainActivityContent()
            }
        }
    }
...
```

Тема TemperatureConverterTheme применяется к MainActivityContent, включая поверхности.

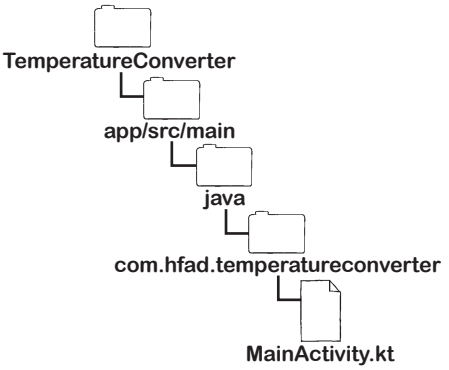


Текст
Преобразователь
Изменение

Полный код MainActivity.kt

Теперь вы знаете все необходимое для настройки внешнего вида пользовательского интерфейса и приведения его к нужному виду.

Ниже приведен полный код *MainActivity.kt*; обновите содержимое файла (изменения выделены жирным шрифтом):




```
package com.hfad.temperatureconverter

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.*
import androidx.compose.material.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.ContentScale
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface {
                    MainActivityContent()
                }
            }
        }
    }
}
```

Добавьте эти команды импортирования.

Тема Material по умолчанию применяется к композиции, включая все поверхности.

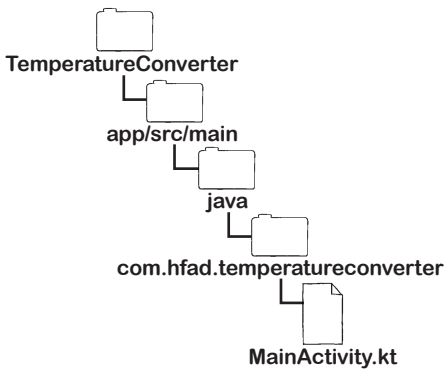
Продолжение на следующей странице.

MainActivity.kt (продолжение)

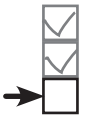
Текст
Преобразователь
Изменение

```
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}

@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}

@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}

@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}
```

Изменять код на этой странице не нужно.



Продолжение на следующей странице.



Текст
Преобразователь
Изменение

MainActivity.kt (продолжение)

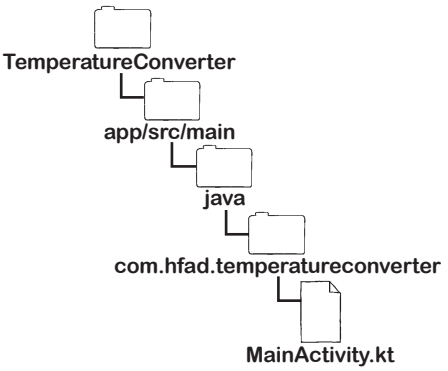
```
@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }

    Column(modifier = Modifier.padding(16.dp).fillMaxWidth()) {
        Header(R.drawable.sunrise, "sunrise image")
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            ConvertButton {
                newCelsius.value.toIntOrNull()?.let {
                    celsius.value = it
                }
            }
        }
        TemperatureText(celsius.value)
    }
}

@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MaterialTheme {
        Surface {
            MainActivityContent()
        }
    }
}
```

Пользовательский интерфейс отделяется от краев экрана отступами размером 16dp.

Добавьте компонент Row для горизонтального выравнивания Button по центру.

Закрывающая фигурная скобка для Row.

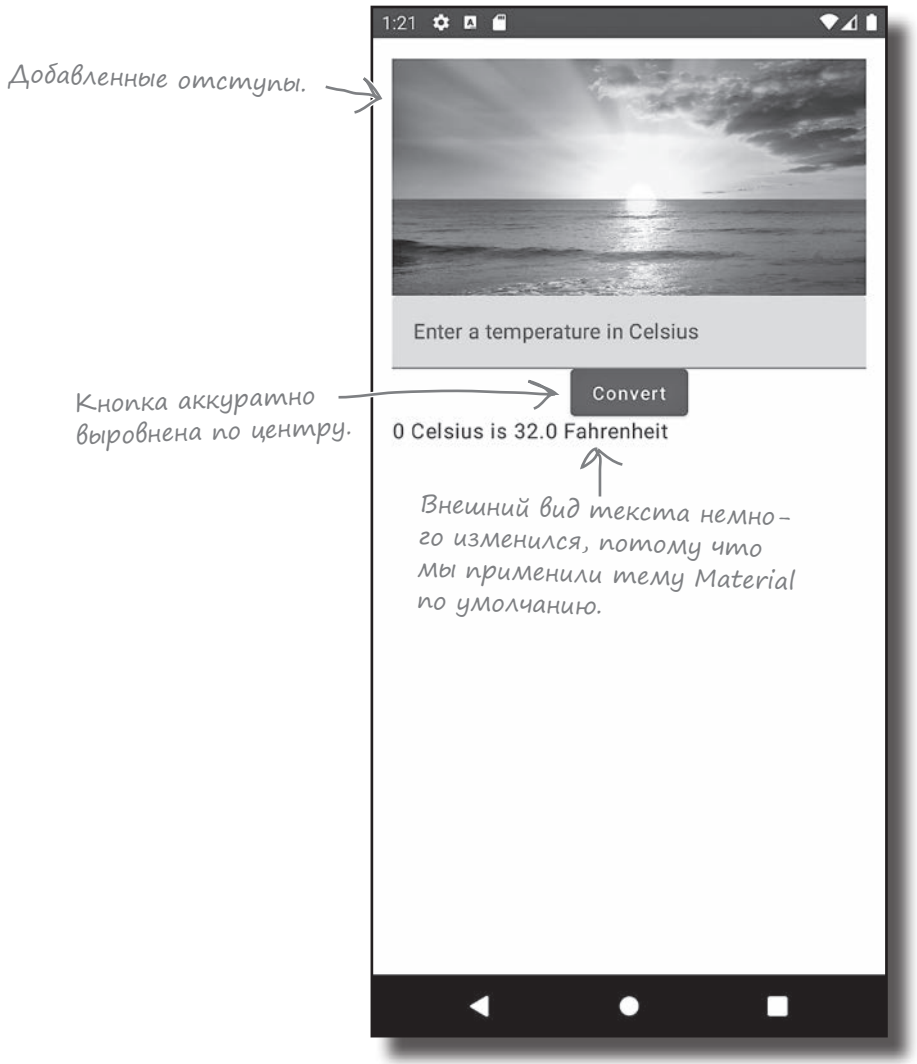
Применяется тема Material по умолчанию.



Проведем тест-драйв приложения.

Тест-драйв

При запуске приложения отображается активность `MainActivity`, а пользовательский интерфейс выглядит именно так, как нужно. Область компонентов отделена отступами от краев экрана, кнопка выравнивается по центру, а в интерфейсе используется тема Material по умолчанию.



Поздравляем! Вы научились строить пользовательские интерфейсы из компонентов Jetpack Compose вместо представлений.

В следующей главе мы пойдем еще дальше — вы узнаете, как интегрировать компоненты Compose в существующие пользовательские интерфейсы на базе `View`.

СТАНЬ Compose

Приведенная ниже компонентная функция включает компоненты TextField, Button и Text. Когда пользователь вводит имя в текстовом поле и щелкает на кнопке, в текстовой надписи должно появиться новое имя. Представьте себя на месте Compose и определите, будет ли функция работать так, как задумано. Если не будет, то как бы вы изменили ее?

```
@Composable
fun ChangeHello() {
    val name = mutableStateOf("friend")
    val newName = mutableStateOf("")

    TextField(
        value = newName.value,
        label = { Text("Enter your name") },
        onValueChange = { name.value = it }
    )

    Button(
        onClick = { name = newName }
    ) {
        Text("Update Hello")
    }

    Text("Hello ${name.value}")
}
```

СТАНЬ Compose. Решение

Приведенная ниже компонентная функция включает компоненты TextField, Button и Text. Когда пользователь вводит имя в текстовом поле и щелкает на кнопке, в текстовой надписи должно появиться новое имя. Представьте себя на месте Compose и определите, будет ли функция работать так, как задумано. Если не будет, то как бы вы изменили ее?

Чтобы функция работала так, как было запланировано, необходимо внести ряд изменений, показанных ниже.

```
@Composable
fun ChangeHello() {
    val name = remember { mutableStateOf("friend") }
    val newName = remember { mutableStateOf("") }

    Column {
        TextField(
            value = newName.value,
            label = { Text("Enter your name") },
            onValueChange = { newName.value = it }
        )

        Button(
            onClick = { name.value = newName.value }
        ) {
            Text("Update Hello")
        }

        Text("Hello ${name.value}")
    }
}
```

name и newName должны храниться в памяти.

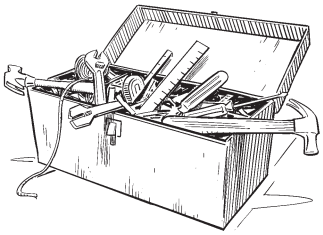
Без этого компоненты были бы наложены друг на друга.

Компонент TextField получает свое значение из newName.

Значение name.value должно обновляться по щелчку на кнопке.

Закрывающая фигурная скобка для Column.

Ваш инструментарий Android



Глава 18 осталась позади, а ваш инструментарий пополнился Jetpack Compose.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Jetpack Compose позволяет строить платформенные пользовательские интерфейсы в коде Kotlin.
- Пользовательский интерфейс строится из компонентов Compose (вместо представлений и макетов).
- Активности Compose расширяют класс `ComponentActivity`.
- Все компоненты Compose, выполняемые из `setContent()`, включаются в пользовательский интерфейс активности.
- Компонентной функцией называется функция, использующая один или несколько компонентов Compose. Она должна быть помечена аннотацией `@Composable`.
- Чтобы выполнять предварительный просмотр результатов компонентных функций без аргументов, используйте аннотацию `@Preview`.
- Компонент `Text` предназначен для вывода текста.
- Компоненты `Row` и `Column` предназначены для размещения компонентов по строкам и столбцам.
- Компонент `Image` предназначен для вывода изображений.
- Компонент `Button` предназначен для вывода кнопок.
- Компоненты перестраиваются только при обновлении значений, от которых они зависят.
- Функция `remember` используется для сохранения объектов в памяти.
- Функция `mutableStateOf()` создает объект `MutableState`, значения которого могут обновляться.
- Компонент `TextField` предназначен для вывода текстового поля.
- Модификаторы (`Modifier`) добавляют поведение к компонентам.
- `Column` и `Row` используются для выравнивания компонентов по центру.
- `MaterialTheme` используется для применения к компонентам темы Material по умолчанию.
- Тему по умолчанию можно переопределить; создайте новую тему в коде Kotlin и примените ее к пользовательскому интерфейсу.
- Компонент `Surface` используется для назначения стилей к поверхностям и применения 3D-эффектов.

Полная гармония

Лучшие результаты достигаются при совместной работе. К настоящему моменту вы научились строить пользовательские интерфейсы из представлений и компонентов Compose. А если вы хотите использовать их **одновременно**? В этой главе вы узнаете, как пользоваться преимуществами обеих технологий, добавляя компоненты Compose в пользовательские интерфейсы на базе представлений. Вы освоите средства, позволяющие компонентам **работать с моделями представлений**. Вы даже узнаете, как заставить их **реагировать на *обновления Live Data***. К концу главы у вас появится все необходимое для **использования компонентов Compose с представлениями** и даже для **перехода на пользовательские интерфейсы**, построенные **исключительно на базе Compose**.

Компоненты Compose могут включаться в интерфейсы на базе View

В предыдущей главе вы научились реализовывать пользовательские интерфейсы Compose на примере нового приложения Temperature Converter. Вместо того чтобы добавлять представления в файл макета, мы строили интерфейс вызовами компонентных функций из кода Kotlin активности.

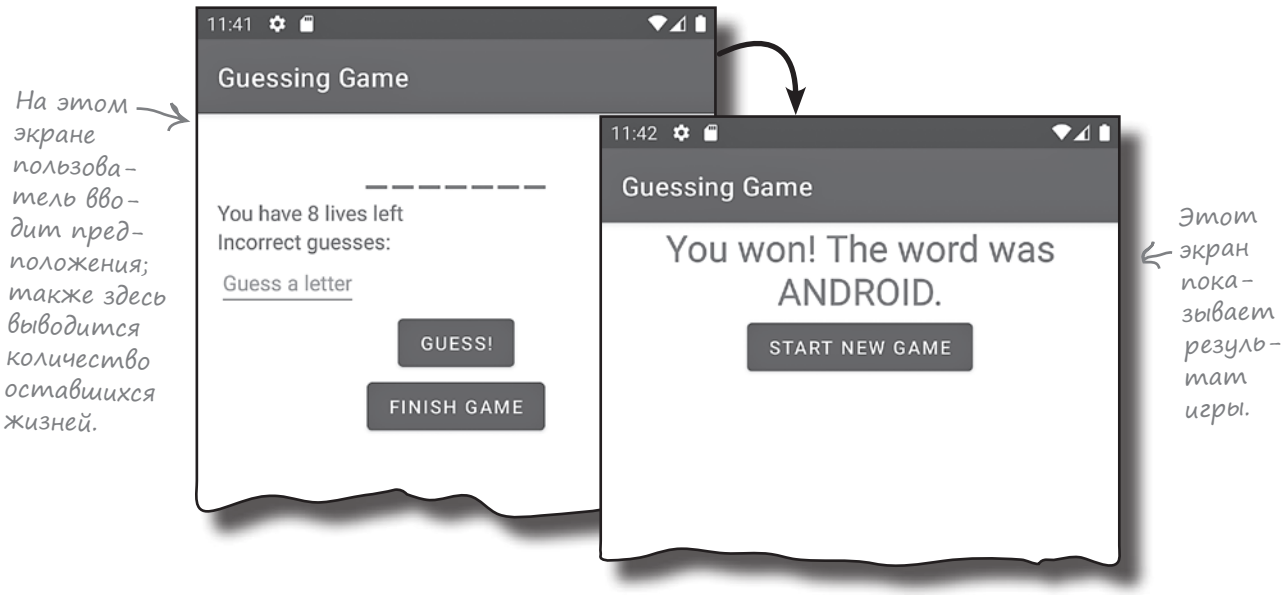
Однако в некоторых ситуациях требуется совместить представления с компонентами Compose *в одном* пользовательском интерфейсе, например, если вы хотите использовать элементы, доступные только в форме представлений или компонентов Compose, или хотите перевести части своего приложения на Compose.

А теперь хорошая новость: компоненты Compose можно добавить в пользовательский интерфейс, определенный в файле макета. Чтобы показать, как это делается, мы вернемся к приложению Guessing Game, созданному ранее в книге, и переведем его на Compose.

Возвращаемся к приложению Guessing Game

Наверняка вы помните, что в приложении Guessing Game пользователь вводит свои предположения о том, какие буквы входят в загаданное слово. Пользователь побеждает, если все буквы будут отгаданы успешно, и проигрывает, если у него заканчиваются жизни.

В текущей версии игра выглядит так:



Прежде чем переходить к замене представлений компонентами, давайте в общих чертах вспомним, как устроено приложение.

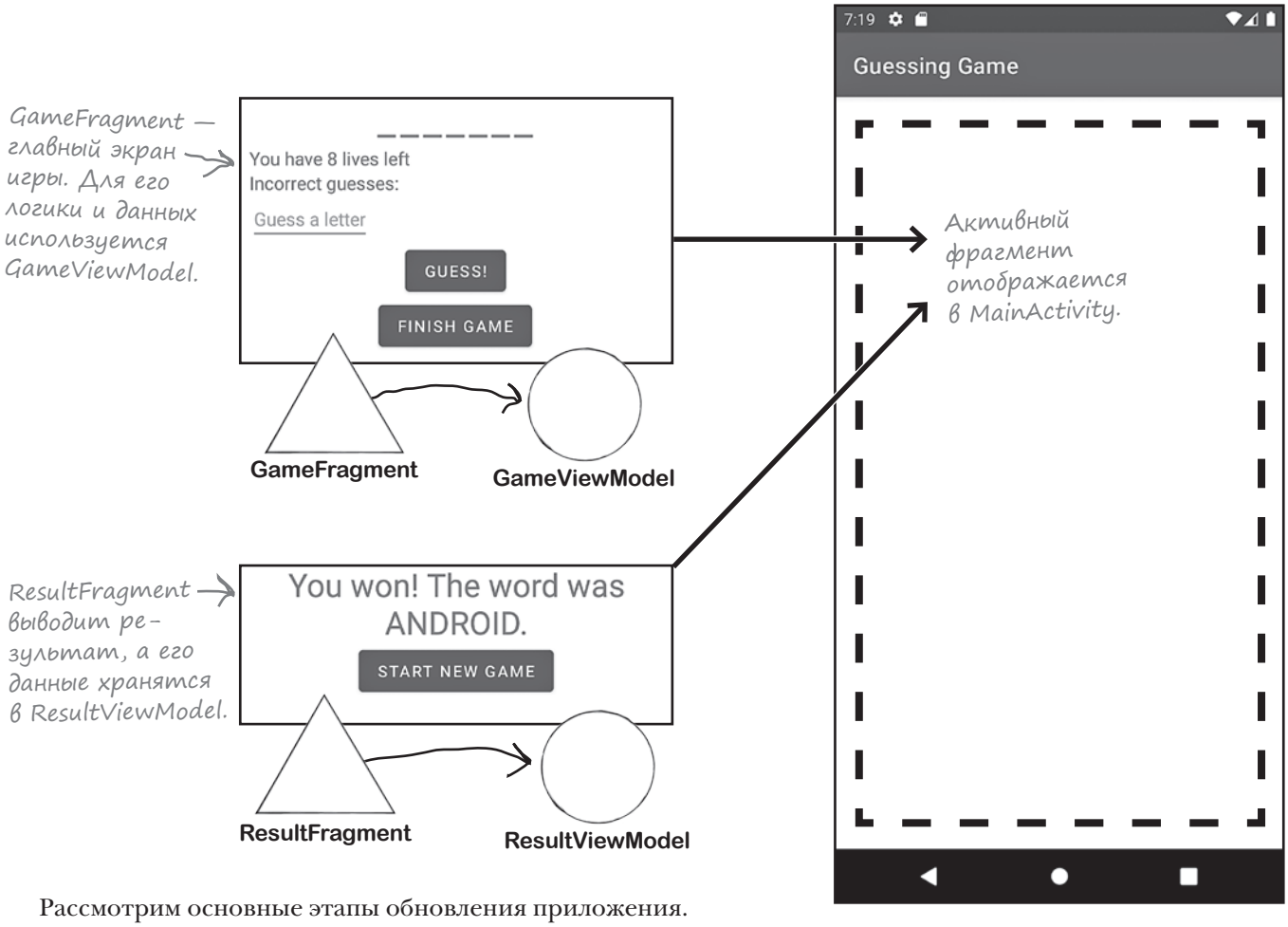
Структура приложения Guessing Game

Пользовательский интерфейс приложения Guessing Game состоит из двух фрагментов: `GameFragment` и `ResultFragment`.

`GameFragment` — главный экран приложения, на котором, собственно, проходит игра. Здесь выводится информация (количество оставшихся жизней и ошибочные предположения пользователя), а пользователь вводит свои предположения. Также здесь находится кнопка, по щелчку на которой пользователь немедленно завершает игру.

Фрагмент `ResultFragment` отображается при завершении игры. Он сообщает, выиграл ли пользователь, а также выводит загаданное слово.

Приложение также включает две модели представлений (`GameViewModel` и `ResultViewModel`), которые содержат логику и данные игры и сохраняют состояние при повороте устройства. `GameFragment` использует `GameViewModel`, а `ResultFragment` использует `ResultViewModel`:



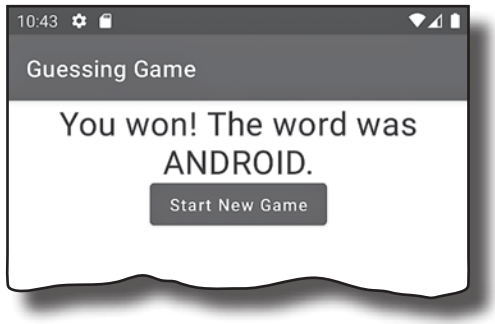
Рассмотрим основные этапы обновления приложения.

Что мы собираемся сделать

Замена представлений Guessing Game компонентами Compose будет выполнена в два этапа:

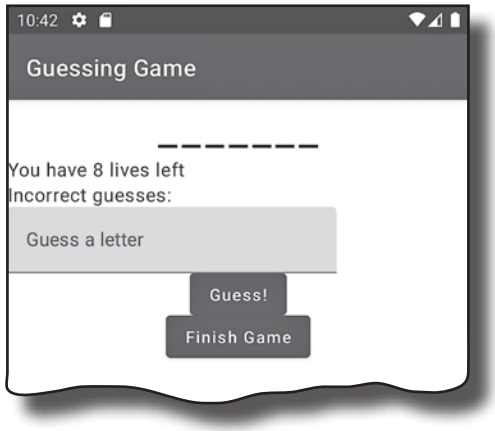
1. **Замена представлений ResultFragment компонентами Compose.** Мы добавим библиотеки Compose в файлы *build.gradle* приложения, а затем добавим компоненты в макет `ResultFragment` для замены текущих представлений. Когда вы будете уверены в том, что компоненты делают то, что требуется, представления будут удалены из пользовательского интерфейса.

Так должна выглядеть версия пользовательского интерфейса ResultFragment на базе Compose.



2. **Замена представлений GameFragment компонентами Compose.** Затем аналогичная процедура будет повторена для `GameFragment`. Сначала мы добавим компоненты Compose в макет, чтобы продублировать существующие представления. А когда вы будете уверены в том, что компоненты делают то, что требуется, представления будут удалены из пользовательского интерфейса.

Так должна выглядеть версия GameFragment на базе Compose.



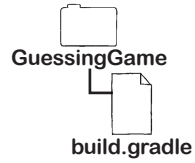
Начнем с добавления библиотек Compose в файлы *build.gradle* приложения.

Не забудьте!

Мы собираемся изменить приложение Guessing Game, которое строилось в главах с 11-й по 13-ю. Откройте проект этого приложения.

Обновление файла build.gradle проекта...

Начнем с добавления в файл *build.gradle* проекта новой переменной, определяющей используемую версию Compose. Откройте файл *GuessingGame/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `buildscript`:

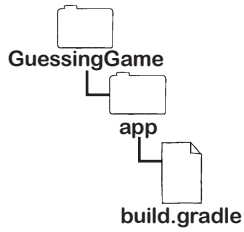



```
buildscript {
    ext.compose_version = "1.0.1"
}
```

Используемая версия библиотек Compose.

...а также обновление файла build.gradle проекта

В файл *build.gradle* приложения необходимо добавить некоторые настройки и библиотеки Compose, а также установить минимальный уровень SDK 21. Откройте файл *GuessingGame/app/build.gradle* и добавьте следующие строки (выделенные жирным шрифтом) в соответствующие разделы:




```
android {
    defaultConfig {
        minSdk 21
        ...
    }
    buildFeatures {
        ...
        compose true
    }
    composeOptions {
        kotlinCompilerExtensionVersion compose_version
    }
}
```

Убедитесь в том, что файл включает эту строку.

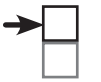
Включите использование Compose.

Добавьте этот раздел.

```
dependencies {
    ...
    implementation("androidx.compose.ui:ui:$compose_version")
    implementation("androidx.compose.ui:ui-tooling:$compose_version")
    implementation("androidx.compose.foundation:foundation:$compose_version")
    implementation("androidx.compose.material:material:$compose_version")
    implementation("androidx.compose.material:material-icons-core:$compose_version")
    implementation("androidx.compose.material:material-icons-extended:$compose_version")
    implementation("androidx.compose.runtime:runtime-livedata:$compose_version")
    ...
}
```

Необходимо добавить все эти зависимости Compose (да, мы серьезно).

После внесения всех изменений щелкните на ссылке Sync Now.



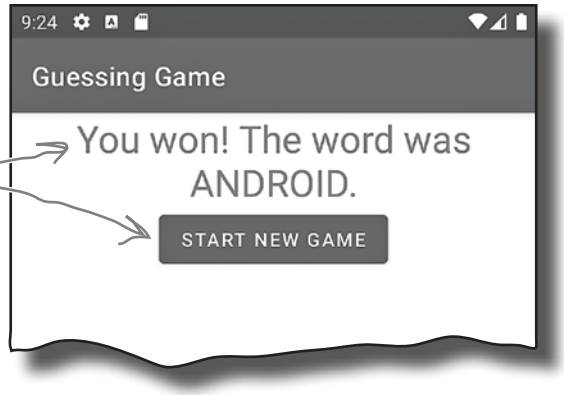
ResultFragment
GameFragment

Замена представлений в ResultFragment компонентами Compose

Мы обновили файлы *build.gradle* и включили в них зависимости Compose, теперь можно заменить представления приложений компонентами Compose. Начнем с более простого компонента `ResultFragment`.

Как вы помните, макет `ResultFragment` включает представление `TextView` для отображения результата игры и кнопку `Button` для запуска новой игры. Он выглядит так:

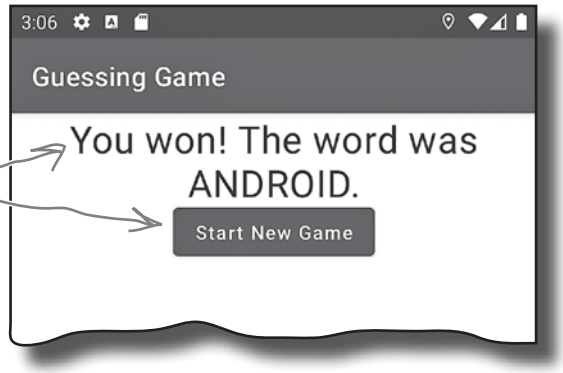
В настоящее время ResultFragment использует в своем пользовательском интерфейсе представления View.



Обратите внимание на незначительные различия между компонентами Compose и представлениями в теме Material по умолчанию. Чтобы изменить это поведение, можно переопределить тему или изменить свойства компонентов.

Эти представления можно заменить компонентами Compose, иначе говоря, вместо `TextView` будут использоваться компоненты `Text`, а вместо представлений `Button` — компоненты `Button`. Новый пользовательский интерфейс будет выглядеть так:

Представления View заменены компонентами Compose.



Новый пользовательский интерфейс будет строиться постепенно, так что на начальной стадии в `ResultFragment` будут использоваться как представления, так и компоненты Compose. Давайте разберемся, как добавить компоненты Compose в файл макета.

ComposeView позволяет включить компоненты Compose в макет

Чтобы включить компоненты Compose в пользовательский интерфейс на базе `View`, добавьте в файл макета элемент **`ComposeView`** — разновидность представлений, в которой могут отображаться представления Compose. Код выглядит так:

```
<androidx.compose.ui.platform.ComposeView
    android:id="@+id/compose_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

Это весь код, необходимый для добавления ComposeView в файл макета. Разместите его там, где в пользовательском интерфейсе должны находиться компоненты Compose.

`ComposeView` — своего рода представление-заместитель для любых компонентов Compose, которые вы хотите включить в пользовательский интерфейс из кода Kotlin. При запуске приложение отображает представления макета и заполняет `ComposeView` компонентами Compose.

ComposeView — разновидность View, которая действует как заместитель для компонентов Compose и позволяет использовать компоненты Compose в пользовательских интерфейсах на базе View.

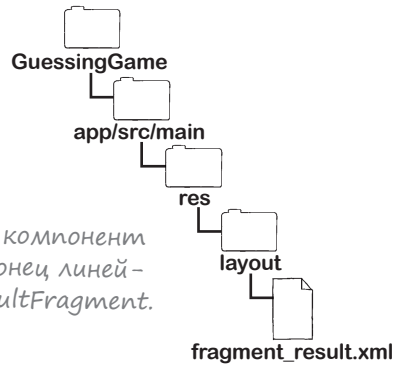
Добавление ComposeView в fragment_result.xml

В пользовательском интерфейсе `ResultFragment` должны присутствовать компоненты `Text` и `Button`, поэтому в файл макета необходимо добавить компонент `ComposeView`.

Ниже приведена новая версия *fragment_result.xml*; обновите файл (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout...>

    <data>
        <variable
            name="resultViewModel"
            type="com.hfad.guessinggame.ResultViewModel" />
    </data>

    <LinearLayout...>
        ...
        <androidx.compose.ui.platform.ComposeView
            android:id="@+id/compose_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>
    </LinearLayout>
</layout>
```

Добавьте этот компонент ComposeView в конец линейного макета ResultFragment.



После того как представление `ComposeView` было добавлено в макет, включим в него компоненты Compose.

ResultFragment
GameFragment

Добавление компонентов Compose в код Kotlin

После того как в макет будет включено представление `ComposeView`, к нему можно будет добавить компоненты Compose в методе `onCreateView()` фрагмента. Для этого используется следующий код:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
        composeView.setContent {
            //Добавление компонентов Compose
        }
    }
    ...
}
```

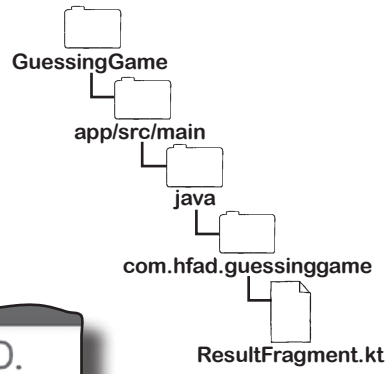
Включает компоненты Compose в пользовательский интерфейс.

Сообщает ComposeView, какие компоненты следует включить.

В этом коде вызывается `setContent()` для представления `ComposeView` из макета; он сообщает, какие компоненты необходимо включить. Результат применяется к заполненному макету фрагмента. Например, если вы захотите добавить компонент `Text` к макету `ResultFragment`, для этого будет использоваться следующий код:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
        composeView.setContent {
            Text("This is a composable")
        }
    }
    ...
}
```

Добавляет компонент Text в макет ResultFragment.



При выполнении этого кода компонент `Text` будет отображаться в представлении `ComposeView`:

Исходные представления макета.

Компонент Text из пользовательского интерфейса ResultFragment.

Добавление компонентной функции для содержимого фрагмента

Итак, вы научились добавлять компоненты Compose в `ComposeView`. Воспользуемся компонентом, добавленным в *fragment_result.xml*, для замены существующих представлений.

Начнем с добавления в *ResultFragment.kt* новой компонентной функции с именем `ResultFragmentContent`, которая будет использоваться для определения пользовательского интерфейса фрагмента. Функция будет вызываться из `setContent()`, чтобы все добавленные в нее компоненты Compose выполнялись при отображении `ResultFragment`.

Новый код приведен ниже; мы добавим его в *ResultFragment.kt* через несколько страниц:

```
...
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable

class ResultFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        ResultFragmentContent()
                    }
                }
            }
        }
        ...
    }
    ...
}

@Composable
fun ResultFragmentContent() {
}
```

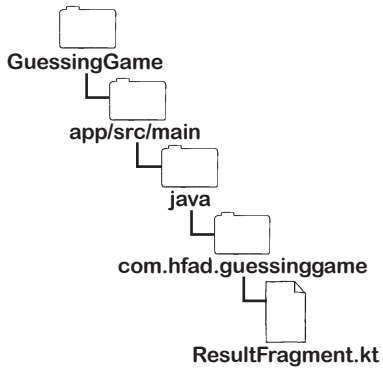
Импортируйте эти классы.

Добавляет ResultFragmentContent в ComposeView и применяет тему Material по умолчанию.

Применяет компоненты в пользовательском интерфейсе.

Остальной код изменять не нужно.

Компонентная функция ResultFragmentContent.



Замена кнопки Start New Game

После того как компонентная функция `ResultFragmentContent` будет добавлена в *ResultFragment.kt*, мы можем использовать ее для включения компонентов Compose в пользовательский интерфейс. Начнем с добавления кнопки.

`ResultFragment` в настоящее время включает кнопку `View` с надписью «Start new game», которая использует следующего слушателя `OnClickListener`, чтобы по щелчку на кнопке происходил переход к `GameFragment`:


```
binding.newGameButton.setOnClickListener {
    view.findNavController()
        .navigate(R.id.action_resultFragment_to_gameFragment)
}
```

По щелчку на кнопке будет выполнен этот код.

Чтобы смоделировать кнопку средствами Compose, можно создать новую компонентную функцию с именем `NewGameButton`, которая будет выполняться из `ResultFragmentContent`. К `NewGameButton` будет добавлен аргумент с лямбда-выражением, при помощи которого `ResultFragmentContent` сообщит, что должно происходить по щелчку.

Новый код приведен ниже; мы добавим его в *ResultFragment.kt* через пару страниц:

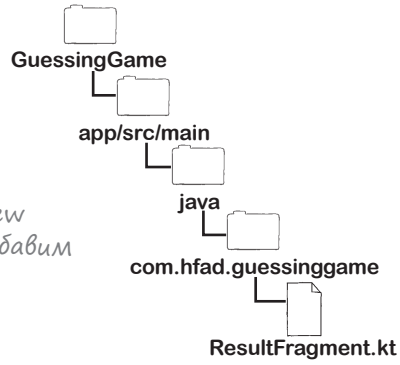
```
@Composable
fun NewGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Start New Game")
    }
}

@Composable
fun ResultFragmentContent(view: View) {
    Column(modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally) {
    NewGameButton {
        view.findNavController()
            .navigate(R.id.action_resultFragment_to_gameFragment)
    }
    }
}
```

Компонентная функция NewGameButton.

NewGameButton необходимо представление View для перехода к GameFragment, поэтому мы добавим в ResultFragmentContent аргумент View.

Компонент Column используется для горизонтального выравнивания кнопки по центру.

Добавляет NewGameButton в пользовательский интерфейс и обеспечивает переход по щелчку.

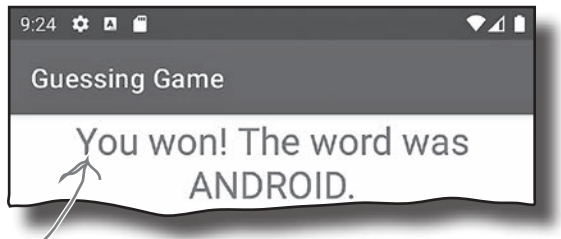


С кнопкой мы разобрались. А как насчет текста?

Замена TextView в ResultFragment

Напомним, что `ResultFragment` использует представление `TextView` в своем макете для отображения результата игры. Представление определяется следующим кодом:

```
<TextView
    android:id="@+id/won_lost"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:textSize="28sp"
    android:text="@{resultViewModel.result}" />
```



Текст TextView поступает из свойства result объекта ResultViewModel.

Вместо того чтобы использовать для вывода текста `TextView`, мы определим новую компонентную функцию `ResultText`, которая выводит текст в компоненте Compose `Text`. В функцию `ResultText` будет включен строковый аргумент, который будет использоваться `ResultFragmentContent` для передачи текста.

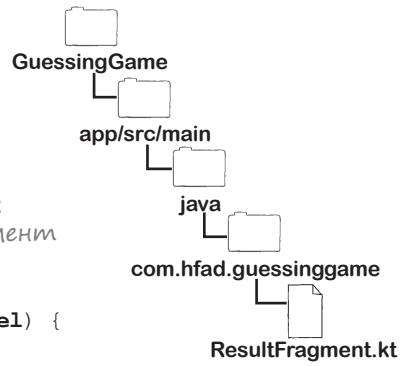
Ниже приведен новый код; он будет добавлен в файл *ResultFragment.kt* на следующей странице:

```
@Composable
fun ResultText(result: String) {
    Text(text = result,
         fontSize = 28.sp,
         textAlign = TextAlign.Center)
}

@Composable
fun ResultFragmentContent(view: View, viewModel: ResultViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ResultText(viewModel.result)
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

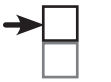
Компонентная функция ResultText.

ResultText получает свой текст от модели представления, поэтому мы добавим аргумент ResultViewModel к ResultFragmentContent.

Включаем компонент ResultText в пользовательский интерфейс и даем ему команду использовать свойство result модели представления для получения текста.



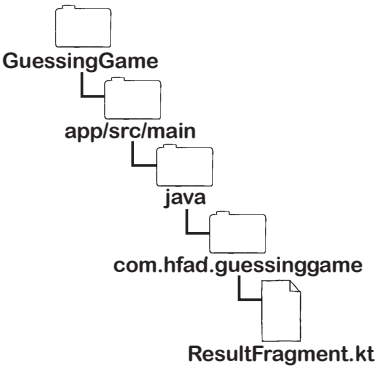
И это все, что необходимо для замены всех представлений `ResultFragment` компонентами. Посмотрим, как выглядит полный код.



ResultFragment
GameFragment

Обновленный код ResultFragment.kt

Ниже приведена новая версия кода *ResultFragment.kt*; обновите файл (изменения выделены жирным шрифтом):




```
...
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.Button
import androidx.compose.material.Text
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.sp

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        view?.let { ResultFragmentContent(it, viewModel) }
                    }
                }
            }
        }
        val view = binding.root
```

Импортируйте эти классы.

Добавьте эти строки.

Компоненты применяются в пользовательском интерфейсе.

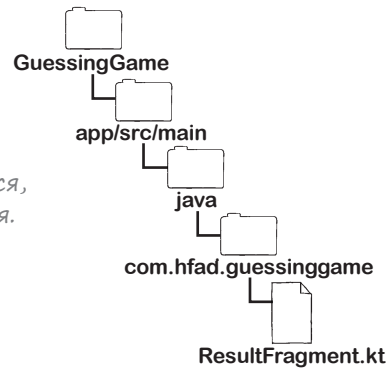
view вызывает метод getView() фрагмента, который возвращает корневое представление.

Продолжение на следующей странице.

ResultFragment.kt (продолжение)

```
        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this, viewModelFactory)
            .get(ResultViewModel::class.java)
        binding.resultViewModel = viewModel

        binding.newGameButton.setOnClickListener { ... }
        return view
    }

    override fun onDestroyView() { ... }
}
```

Этот код не приводится, так как он не изменился.



Добавляем компонентную функцию ResultText, которая выводит результат.

```
@Composable
fun ResultText(result: String) {
    Text(text = result,
        fontSize = 28.sp,
        textAlign = TextAlign.Center)
}
```

Добавляем NewGameButton для вывода кнопки.

```
@Composable
fun NewGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Start New Game")
    }
}
```

Добавляем эту компонентную функцию, которая будет использоваться для основного содержимого фрагмента.

```
@Composable
fun ResultFragmentContent(view: View, viewModel: ResultViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ResultText(viewModel.result)
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

В ResultFragmentContent добавляются компоненты ResultText и NewGameButton, выстроенные в столбец.

Проведем тест-драйв приложения.

Тест-драйв

ResultFragment
GameFragment

При запуске приложения отображается фрагмент `GameFragment`. Если щелкнуть на кнопке Finish Game, приложение переходит к `ResultFragment`.

`ResultFragment` отображает исходные представления и компоненты, которые мы только что добавили. Компонент `Text` вводит правильный текст, а по щелчку на компоненте `Button` происходит переход к `GameFragment`.

Компоненты, добавленные в `ResultFragment`, работают так, как планировалось. А значит, на следующем этапе представления можно удалить.

Часто Задаваемые Вопросы

В: Получается, представления и компоненты Compose в `ResultFragment` используют одну модель представления. Это так?

О: Да. Модель представления содержит свойство с результатом игры, а компоненты `TextView` и `Text` обращаются к нему.

В: Компоненты обычно используют модель представления?

О: Да, так часто бывает. Как и в пользовательских интерфейсах на базе `View`, модели представлений предоставляют удобный механизм управления состоянием любых компонентов, чтобы оно не терялось при повороте экрана.

В: Вы передаете модель представления `ResultFragmentContent` в аргументе. Разве функция не может просто получить ссылку на модель представления?

О: Компонентные функции могут получить ссылку на модель представления при помощи функции `viewModel()`, но на момент написания книги библиотека Compose еще не была полностью стабильной. Дополнительная информация об этой возможности (и ряде других) приведена в приложении.

ResultFragment
GameFragment

Необходимо удалить представления из ResultFragment...

Представления можно удалить из фрагмента или активности двумя способами:

1. **Удалить все лишние представления из файла макета.**
   Такой подход может быть полезен, если пользовательский интерфейс содержит как представления, так и компоненты Compose.

2. **Удалить весь файл макета.**
   Если пользовательский интерфейс содержит *только* компоненты, файл макета можно удалить и убрать любые ссылки на него из кода Kotlin активности или фрагмента.

В приложении Guessing Game мы заменили все представления `ResultFragment` компонентами Compose, поэтому все исходные представления уже не нужны. Это означает, что файл макета *fragment_result.xml* можно удалить, чтобы пользовательский интерфейс содержал только компоненты Compose.

...и обновить ResultFragment.kt

Прежде чем удалять файл макета, необходимо сначала удалить все ссылки на его представления из *ResultFragment.kt* и отказаться от использования связывания представлений.

Связывание представлений предоставляет вашему коду активностей и фрагментов простой способ обращения к представлениям файла макета. Так как мы удаляем файл макета ResultFragment, связывание представлений тоже становится лишним.

Также необходимо изменить метод `onCreateView()` фрагмента, чтобы вместо заполнения файла макета он добавлял компоненты Compose в пользовательский интерфейс фрагмента.

Как вы узнали в главе 18, с *активностью* это делается простым вызовом `setContent()` из ее метода `onCreate()`:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            //Код, в котором выполняются компоненты Compose
        }
    }
}
```

Код активности Compose. Вместо него должен использоваться код фрагмента Compose.

Однако с фрагментами потребуется несколько иной подход. Чтобы понять, с чем это связано, вернемся к методу `onCreateView()` фрагмента.

ResultFragment
GameFragment

onCreateView() возвращает корневое представление

Как вы уже знаете, метод `onCreateView()` фрагмента вызывается тогда, когда активности требуется отобразить пользовательский интерфейс фрагмента.

Если пользовательский интерфейс определяется в файле макета, код в методе `onCreateView()` заполняет макет иерархией представлений и возвращает корневое представление. Корневое представление добавляется в макет активности, которая отображает пользовательский интерфейс фрагмента.

Но что происходит, если файл макета не существует?

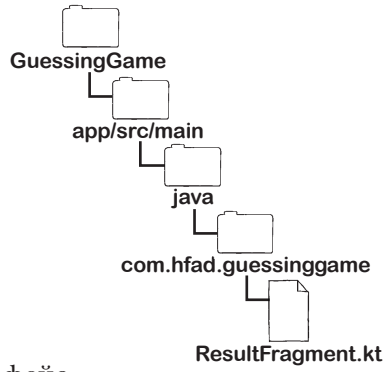
Получение ComposeView для интерфейсов Compose

Если пользовательский интерфейс фрагмента строится из компонентов Compose и не имеет файла макета, **метод `onCreateView()` все равно должен вернуть `View?`, в противном случае код не будет компилироваться**.

Проблема решается возвращением представления `ComposeView`, которое включает все компоненты Compose пользовательского интерфейса. Код выглядит примерно так:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
): View? {
    //Код, не использующий компоненты Compose
                Возвращает ComposeView.
    return ComposeView(requireContext()).apply {
        setContent {
            //Код Compose, определяющий UI
        }
    }
}
```

Компоненты добавляются в ComposeView.



Когда активности потребуется отобразить пользовательский интерфейс фрагмента, она вызывает метод `onCreateView()` фрагмента, как и прежде. Метод возвращает представление `ComposeView`, содержащее компоненты фрагмента, которое затем отображается активностью.

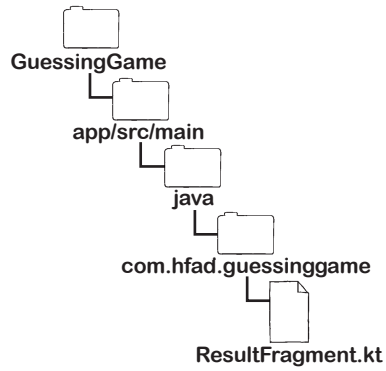
Это все, что необходимо знать для завершения кода *ResultFragment.kt*. Полный код приводится на нескольких ближайших страницах.

Полный код ResultFragment.kt

Ниже приведен полный код *ResultFragment.kt*; обновите файл (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentResultBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.Button
import androidx.compose.material.Text
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.sp
import androidx.compose.ui.platform.ComposeView

class ResultFragment : Fragment() {
    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: ResultViewModel
    lateinit var viewModelFactory: ResultViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val result = ResultFragmentArgs.fromBundle(requireArguments()).result
        viewModelFactory = ResultViewModelFactory(result)
        viewModel = ViewModelProvider(this@ResultFragment, viewModelFactory)
            .get(ResultViewModel::class.java)
```

Удалите эту строку.



Импортируйте этот класс.

Удалите эти строки. Представления более не используются, поэтому применять связывание представлений не обязательно.

Создайте модель представления, прежде чем возвращать ComposeView.

Продолжение на следующей странице.

ResultFragment
GameFragment

ResultFragment.kt (продолжение)

```
_binding = FragmentResultBinding.inflate(inflater, container, false).apply {
return ComposeView(requireContext()).apply {
    composeView.setContent {
        MaterialTheme {
            Surface {
                view?.let { ResultFragmentContent(it, viewModel) }
            }
        }
    }
}
val view = binding.root

val result = ResultFragmentArgs.fromBundle(requireArguments()).result
viewModelFactory = ResultViewModelFactory(result)
viewModel = ViewModelProvider(this, viewModelFactory)
    .get(ResultViewModel::class.java)
binding.resultViewModel = viewModel

binding.newGameButton.setOnClickListener {
    view.findNavController()
        .navigate(R.id.action_resultFragment_to_gameFragment)
}
return view
}

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
}

@Composable
fun ResultText(result: String) {
    Text(text = result,
        fontSize = 28.sp,
        textAlign = TextAlign.Center)
}
```

Удалите эту строку.

Возвращает ComposeView.

Удалите эту ссылку composeView, так как она относится к ComposeView в файле макета.

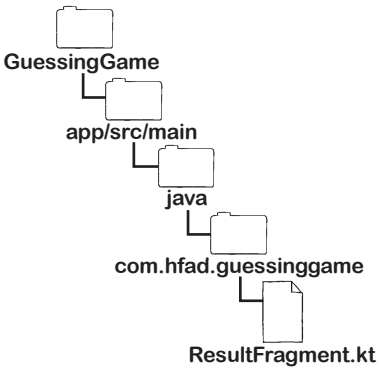
Связывание представлений более не используется, удалите эту строку.

Удалите эти строки, потому что мы получаем модель представления ранее в коде.

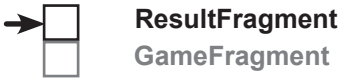
Представление кнопки более не используется, удалите эти строки.

Будьте внимательны: эту фигурную скобку удалять нельзя!

Удалите этот метод, потому что мы отказываемся от использования связывания представлений.



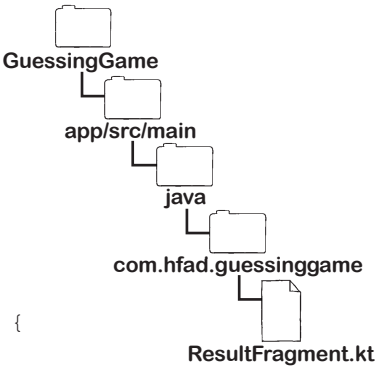
Продолжение на следующей странице.



ResultFragment.kt (продолжение)

```
@Composable
fun NewGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Start New Game")
    }
}

@Composable
fun ResultFragmentContent(view: View, viewModel: ResultViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
           horizontalAlignment = Alignment.CenterHorizontally) {
        ResultText(viewModel.result)
        NewGameButton {
            view.findNavController()
                .navigate(R.id.action_resultFragment_to_gameFragment)
        }
    }
}
```

Изменять код ResultFragment.kt на этой странице не нужно.



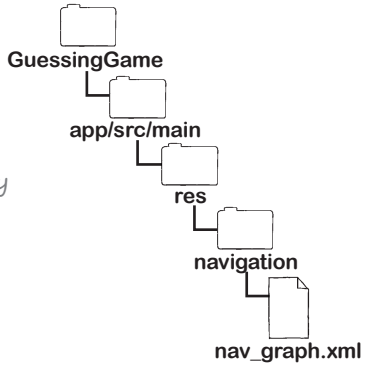
Удаление fragment_result.xml

Теперь `ResultFragment` не использует файл макета, поэтому мы можем удалить любые ссылки на *fragment_result.xml* из графа навигации и удалить файл макета.

Откройте файл *nav_graph.xml* из папки *app/src/main/res/navigation* и удалите ссылку `"@layout/fragment_result"` из раздела `ResultFragment`:

```
...
<fragment
    android:id="@+id/resultFragment"
    ...
    tools:layout="@layout/fragment_result" />
...
```

Удалите эту ссылку из графа навигации.



Щелкните правой кнопкой мыши на файле *fragment_result.xml* на панели проекта, выберите команду Refactor, а затем вариант Safe Delete. Если щелкнуть на кнопке OK и выбрать вариант с проведением рефакторинга, файл будет удален.

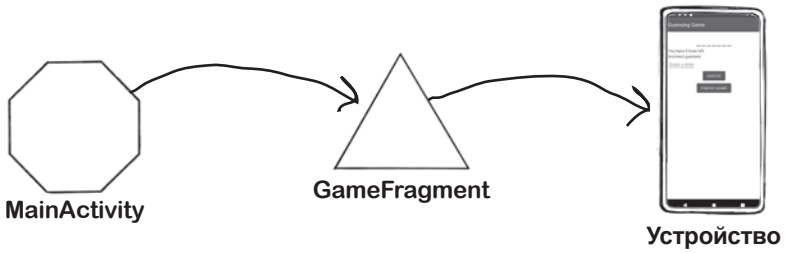
Посмотрим, что произойдет при запуске приложения.

ResultFragment
GameFragment

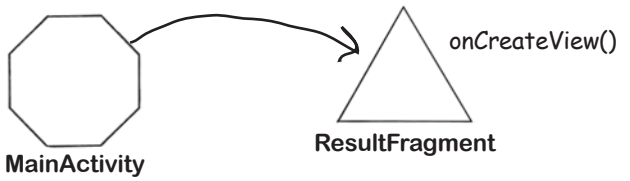
Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

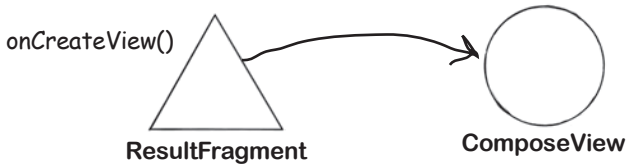
1. **MainActivity запускается и отображает GameFragment в своем макете.**



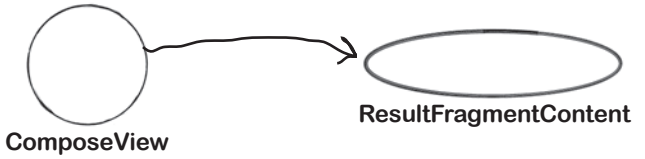
2. **Когда пользователь щелкает на кнопке Finish Game, а также в случае выигрыша или проигрыша приложение переходит к фрагменту ResultFragment и вызывает его метод onCreateView().**



3. **Метод onCreateView() создает ComposeView.**

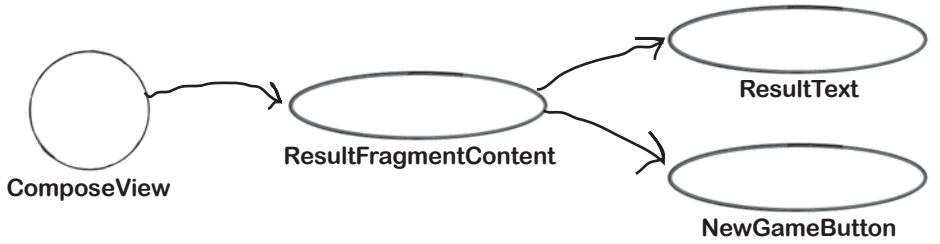


4. **Метод onCreateView() присваивает ResultFragmentContent содержимое ComposeView.**

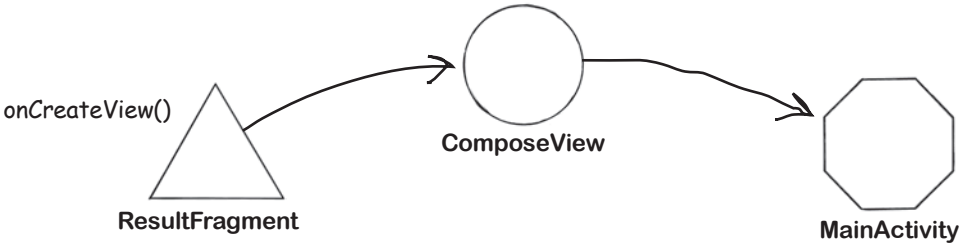


История продолжается

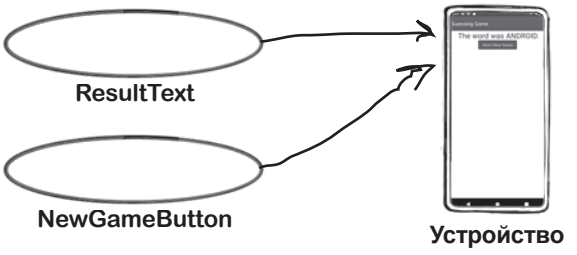
5 **При выполнении ResultFragmentContent выполняются компоненты ResultText и NewGameButton.**
Компоненты добавляются в `ComposeView`.



6 **Метод onCreateView() возвращает ComposeView активности MainActivity.**



7 **MainActivity отображает ComposeView в своем макете.**
Компоненты `ComposeView` отображаются на устройстве.



Проведем тест-драйв приложения.

Тест-драйв

ResultFragment
GameFragment

Если запустить приложение и щелкнуть на кнопке Finish Game фрагмента `GameFragment`, происходит переход к `ResultFragment`, как и прежде. Однако на этот раз пользовательский интерфейс строится из компонентов Compose.

И это все, что необходимо сделать для перевода `ResultFragment` на пользовательский интерфейс Compose. Но прежде чем переходить к работе над `GameFragment`, проверьте свои силы, выполнив упражнение.

Часто Задаваемые Вопросы

В: Вы сказали, что `ComposeView` является разновидностью `View`, в которую можно добавлять компоненты Compose. Существует ли разновидность компонента Compose, в которую можно добавлять представления `View`?

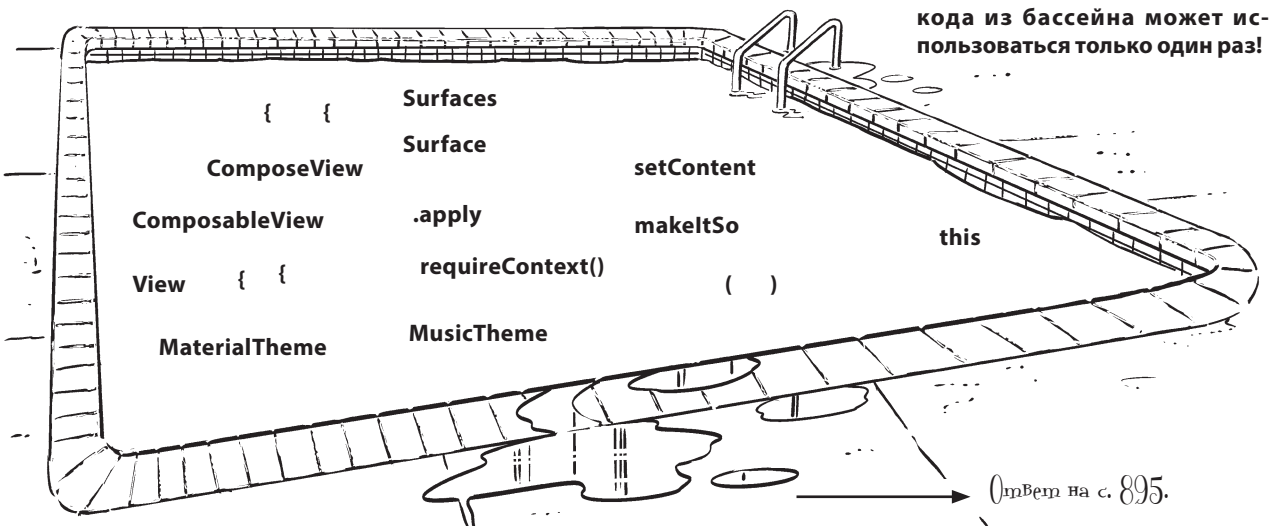
О: Отличный вопрос! Да, можно использовать `AndroidView` — компонент Compose, возвращающий `View`. Например, такая возможность может оказаться полезной, если у вас имеется пользовательский интерфейс Compose и вы хотите добавить в него компонент, существующий только в виде `View`, но не компонент Compose.

У бассейна

Выловите из бассейна сегменты кода и расставьте их в пустых строках в коде. Каждый сегмент кода может использоваться только один раз, при этом все сегменты использовать не обязательно. Ваша **задача** — написать код фрагмента с именем `MusicFragment`, который не имеет файла макета и использует для своего пользовательского интерфейса компонентную функцию с именем `MusicFragmentContent`. Тема с именем `MusicTheme` должна применяться к интерфейсу, включая все поверхности.

```
class MusicFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return ..........................................................................
            ..........................................................................
                ..........................................................................
                    ..........................................................................
                        MusicFragmentContent()
                    }
                }
            }
        }
    }
}
```

Внимание: каждый сегмент кода из бассейна может использоваться только один раз!



Ответ на с. 895.

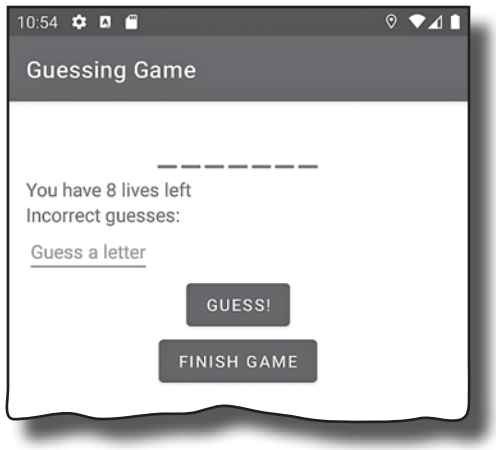
ResultFragment
GameFragment

GameFragment тоже переводится на использование компонентов Compose

Итак, мы успешно перевели `ResultFragment` на использование компонентов вместо представлений; теперь можно сделать то же самое с `GameFragment`.

Как вы помните, макет `GameFragment` включает представления `TextView`, `Button` и `EditText`, при помощи которых пользователь вводит свои предположения в игре. Напомним, как выглядит интерфейс игры:

В настоящее время в пользовательском интерфейсе GameFragment используются эти представления.



На нескольких ближайших страницах мы заменим эти представления компонентами Compose. После завершения работы пользовательский интерфейс будет выглядеть примерно так:

Представления будут заменены компонентами Compose.

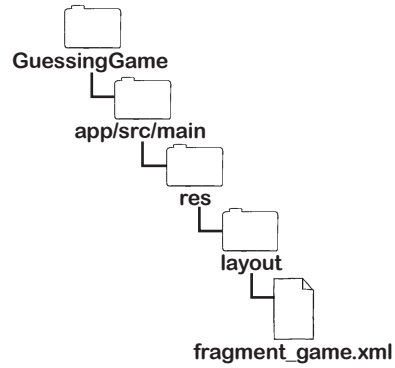
Как и на предыдущем этапе, мы начнем с добавления новых компонентов Compose в пользовательский интерфейс `GameFragment`; это означает, что в его макет необходимо добавить представление `ComposeView`. Код включения `ComposeView` приведен на следующей странице.

ResultFragment
GameFragment

Добавление ComposeView в fragment_game.xml

Ниже приведен код, добавляющий представление `ComposeView` в макет `GameFragment`; обновите файл *fragment_game.xml* (изменения выделены жирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".GameFragment">

    <data>
        <variable
            name="gameViewModel"
            type="com.hfad.guessinggame.GameViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp">

        ...

        <androidx.compose.ui.platform.ComposeView
            android:id="@+id/compose_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"/>
    </LinearLayout>
</layout>
```



Добавьте это представление ComposeView в конец линейного макета. Мы используем его для добавления компонентов Compose в пользовательский интерфейс GameFragment.

Файл макета успешно обновлен, можно переходить к добавлению в него компонентов Compose.

ResultFragment
GameFragment

Добавление компонентной функции для содержимого GameFragment

По аналогии с *ResultFragment.kt* мы добавим в *GameFragment.kt* новую компонентную функцию, которая будет использоваться для пользовательского интерфейса фрагмента. Функция с именем `GameFragmentContent` будет вызываться из `setContent()`, чтобы все добавленные в нее компоненты Compose выполнялись при отображении `GameFragment`.

Ниже показано, как выглядит новый код; мы добавим его в *GameFragment.kt* через несколько страниц:

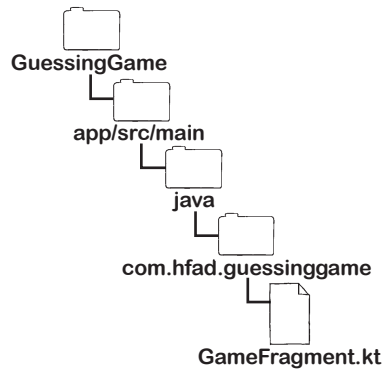
```
...
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable

class GameFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        GameFragmentContent()
                    }
                }
            }
        }
        ...
    }
    ...
}

@Composable
fun GameFragmentContent() {
}
```

Импортируйте эти классы.

Применяет компоненты Compose в пользовательском интерфейсе.

Добавляет GameFragmentContent в ComposeView макета и применяет тему Material по умолчанию.

Компонентная функция GameFragmentContent.



ResultFragment
GameFragment

Замена кнопки Finish Game

Как и в случае с `ResultFragment`, мы добавим компоненты Compose в компонентную функцию `GameFragmentContent`, чтобы они отображались в пользовательском интерфейсе `GameFragment`. Начнем с замены кнопки Finish Game.

Кнопка Finish Game определяется следующим кодом в макете фрагмента:

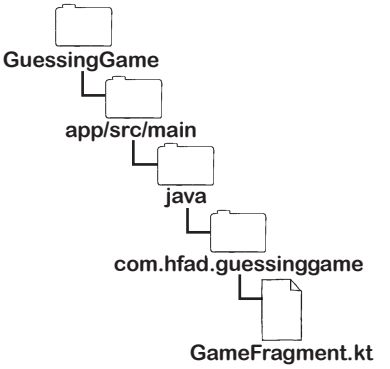
Этот код определяет кнопку Finish Game фрагмента GameFragment.

```
<Button
    android:id="@+id/finish_game_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Finish Game"
    android:onClick="@{() -> gameViewModel.finishGame()}" />
```

По щелчку на кнопке вызывается этот метод.

Как видите, по щелчку на кнопке вызывается метод `finishGame()` объекта `GameViewModel`.

Чтобы воспроизвести эту функциональность в Compose, создайте новую функцию с именем `FinishGameButton`, которая будет выполняться из `GameFragmentContent`. Новый код приведен ниже; мы добавим его в *GameFragment.kt* через несколько страниц:




```
@Composable
fun FinishGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Finish Game")
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    Column(modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally) {
        FinishGameButton {
            viewModel.finishGame()
        }
    }
}
```

Компонентная функция FinishGameButton.

FinishGameButton необходим доступ к объекту GameViewModel, поэтому мы добавим аргумент для этого объекта в GameFragmentContent.

Column используется для выравнивания кнопки по центру.

Добавляет FinishGameButton в пользовательский интерфейс и отдает команду вызывать метод finishGame() по щелчку на кнопке.

Замена EditText на TextField

ResultFragment
GameFragment

Следующим шагом станет замена представления `EditText`, в котором пользователь вводит буквы. Мы создадим новую компонентную функцию с именем `EnterGuess`, которая использует `TextField` и получает два аргумента: строку с введенной буквой и лямбда-выражение, которое указывает, что должно происходить при изменении значения.

Функция `EnterGuess` будет выполняться из `GameFragmentContent`, чтобы компонент был добавлен в пользовательский интерфейс фрагмента. Также мы добавим в `GameFragmentContent` объект `MutableState` с именем `guess`, который будет использоваться для управления состоянием `TextField`.

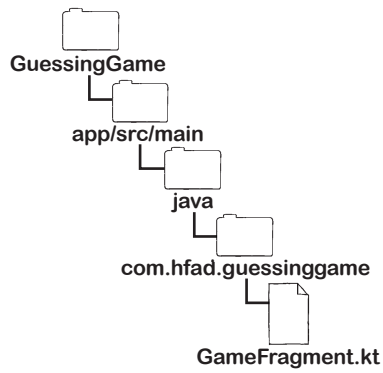
Новый код приводится ниже; мы добавим его в файл *GameFragment.kt* через пару страниц:

Существующий элемент EditText.

Для его замены будет использоваться компонент TextField.

Функция EnterGuess получает строку со значением и лямбда-выражение, которое определяет, что должно происходить при его обновлении.

```
@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) {
    TextField(
        value = guess,
        label = { Text("Guess a letter") },
        onValueChange = changed
    )
}
```




```
@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        EnterGuess(guess.value) { guess.value = it }
        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
        FinishGameButton {
            viewModel.finishGame()
        }
    }
    }
}
```

Будет использоваться компонентной функцией EnterGuess для значения TextField.

Добавляет EnterGuess в пользовательский интерфейс.

Для EnterGuess используется дополнительный компонент Column, чтобы поле не выравнивалось по центру.

ResultFragment
GameFragment

Замена кнопки Guess

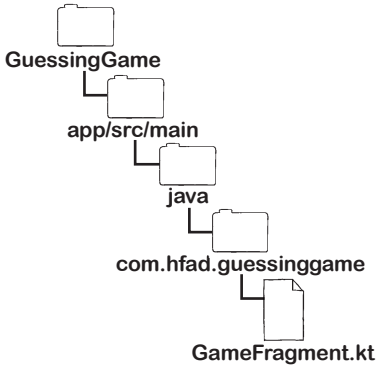
После добавления в интерфейс поля для ввода предположений будет добавлен компонент `Button`, который передает букву методу `makeGuess()` модели представления.

Для добавления кнопки мы воспользуемся новой компонентной функцией с именем `GuessButton`, которая выполняется из `GameFragmentContent`. Новый код приводится ниже; мы добавим его в файл *GameFragment.kt* на следующей странице:

Текущая версия кнопки. Мы заменим ее компонентом Button.

Компонентная функция GuessButton получает аргумент с лямбда-выражением, которое определяет поведение кнопки.

```
@Composable
fun GuessButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Guess!")
    }
}


@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```



Кнопка GuessButton добавляется во второй компонент Column, чтобы она выравнивалась горизонтально по центру.

По щелчку кнопка Button вызывает метод makeGuess() и присваивает переменной guess значение "": Это приводит к очистке текста в текстовом поле EnterGuess.

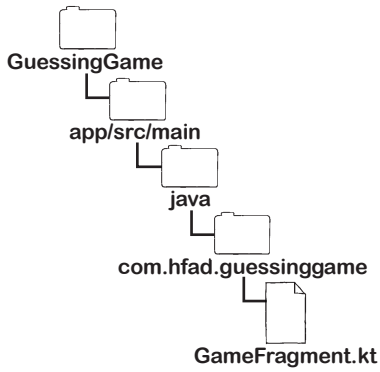
Мы заменили три представления `GameFragment` компонентами Compose. Прежде чем заниматься остальными компонентами, обновим *GameFragment.kt* и проведем тест-драйв приложения.

Обновленный код GameFragment.kt

ResultFragment
GameFragment

Ниже приведена новая версия *GameFragment.kt*; обновите файл (изменения выделены жирным шрифтом):

```
...
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier

class GameFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        GameFragmentContent(viewModel)
                    }
                }
            }
        }
        ...
        return view
    }
    ...
}

@Composable
fun FinishGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Finish Game")
    }
}
```



Импортируйте эти классы.

Добавьте эти строки.

Передает модель представления GameFragmentContent.

Компоненты Compose включаются в пользовательский интерфейс.

Добавляет компонентную функцию FinishGameButton.

Продолжение на следующей странице.

GameFragment.kt (продолжение)

```
@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) {
    TextField(
        value = guess,
        label = { Text("Guess a letter") },
        onValueChange = changed
    )
}

@Composable
fun GuessButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Guess!")
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```

Добавляет компонентную функцию EnterGuess.

Добавляет компонентную функцию GuessButton.

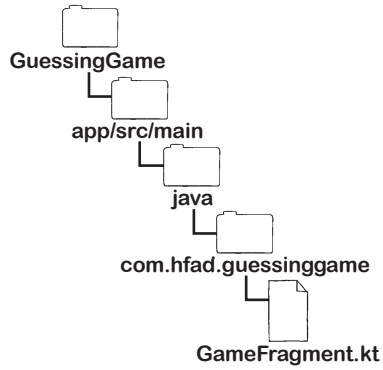
Добавляет объект GameFragmentContent, который будет использоваться для хранения содержимого фрагмента.

Определяет переменную Guess.

Добавляет EnterGuess в Column.

Другой компонент Column используется для выравнивания по центру GuessButton и FinishGameButton.

Выполняет GuessButton и FinishGameButton.



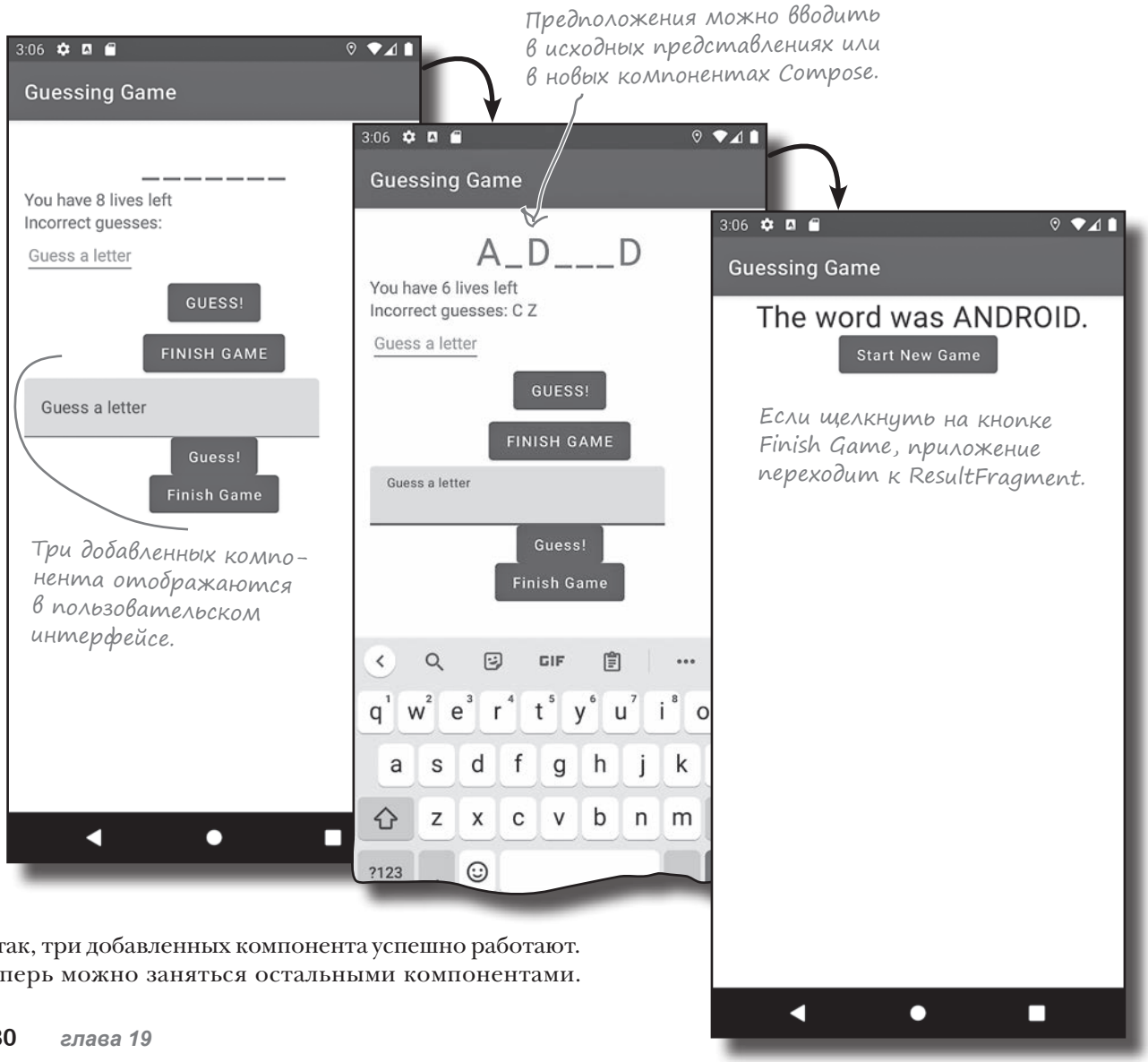
Проведем тест-драйв приложения.

Тест-драйв

ResultFragment
GameFragment

При запуске приложения отображается фрагмент `GameFragment`. Он включает все исходные представления, а также три дополнительных компонента, при помощи которых пользователь может вводить предположения и завершать игру.

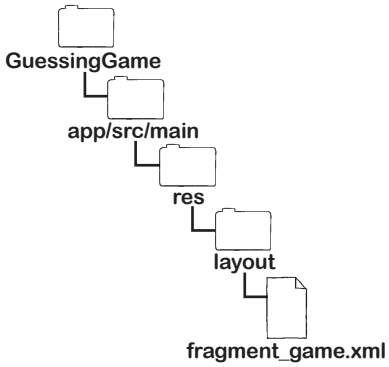
Когда вы используете компоненты `EnterGuess` и `GuessButton` для ввода предположений о том, какие буквы входят в загаданное слово, приложение регистрирует каждую догадку. Если предположение было правильным, буква добавляется в шаблон загаданного слова на экране, а если неправильным — количество оставшихся жизней обновляется и добавляется в список неправильных предположений.



Итак, три добавленных компонента успешно работают. Теперь можно заняться остальными компонентами.

Вывод ошибочных предположений в компоненте Text

На следующем шаге мы заменим представление `TextView`, которое использует механизм Live Data для отображения ошибочных предположений пользователя. Каждый раз, когда пользователь вводит отсутствующую букву, она добавляется в свойство `incorrectGuesses` модели представления, а `TextView` реагирует обновлением выводимого текста.



Напомним, как выглядит код `TextView`:

```
<TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
```

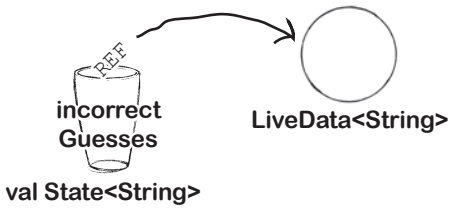
Текст текстового представления определяется присоединением значения свойства incorrectGuesses объекта GameViewModel к строковому ресурсу incorrect_guesses. Так как свойство incorrectGuesses использует механизм Live Data, текст остается актуальным.

`TextView` можно заменить компонентом Compose `Text`, в котором выводится тот же текст. Но что нужно сделать, чтобы текст обновлялся при изменении значения свойства `incorrectGuesses`?

Использование observeAsState() для реагирования на Live Data

Как вы узнали из главы 18, компоненты перестраиваются при появлении новых значений в объектах `State` или `MutableState`, от которых они зависят. Однако с объектами Live Data, такими как свойство `incorrectGuesses` модели представления, этого *не* происходит. Если вы попытаетесь использовать значение объекта `Live Data` с компонентом Compose, он не будет перерисовываться при обновлении значения; он просто продолжит использовать исходное значение объекта, и данные перестанут быть актуальными.

Если вы хотите, чтобы компонент Compose реагировал на обновления `Live Data`, воспользуйтесь функцией **`observeAsState()`**. Функция возвращает `State`-версию объекта `Live Data`, чтобы любые зависящие от него компоненты Compose перестраивались при изменении его значения.



Пример использования функции `observeAsState()`:

```
val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
```

observeAsState() позволяет компонентам реагировать на обновления Live Data так, как если бы они были объектами State.

Эта команда определяет переменную (с типом `State`), которая наблюдает за Live Data-свойством `incorrectGuesses` модели представления. Это означает, что при изменении его значения компоненты смогут отреагировать на него.

Применим этот способ на практике, заменив представление `TextView` в `GameFragment` компонентом Compose.

ResultFragment
GameFragment

Создание компонентной функции IncorrectGuessesText

Чтобы заменить представление `TextView` для ошибочных предположений, мы определим новую компонентную функцию с именем `IncorrectGuessesText`. Эта функция получает аргумент `GameViewModel`, наблюдает за свойством `incorrectGuesses` и использует компонент `Text` для вывода своего значения в пользовательском интерфейсе. При каждом обновлении свойства компонент `Text` перестраивается и выводит обновленный текст.

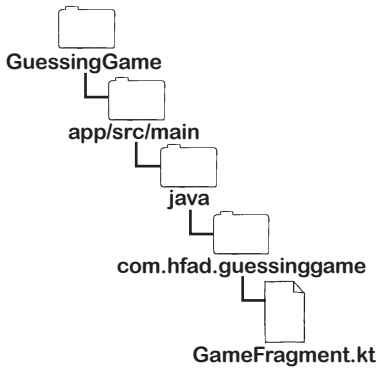
Функция `IncorrectGuessesText` выглядит так:

Функция принимает аргумент GameViewModel.

```
@Composable
fun IncorrectGuessesText(viewModel: GameViewModel) {
    val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
    incorrectGuesses.value?.let {
        Text(stringResource(R.string.incorrect_guesses, it))
    }
}
```

Функция stringResource() позволяет использовать строковые ресурсы с компонентами Compose.

Как видите, компонент `Text` в этом коде использует функцию `stringResource()` для назначения своего текста. Эта функция позволяет использовать строковые ресурсы с компонентами и передавать им аргументы.



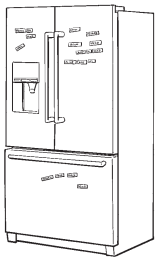
Выполнение IncorrectGuessesText из GameFragmentContent

Как и в случае с остальными компонентными функциями, созданными нами ранее, для включения компонента `IncorrectGuessesText` в пользовательский интерфейс `GameFragment` мы выполним его из компонентной функции `GameFragmentContent`. Новая версия кода выглядит так:

```
@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    ...
    Column(modifier = Modifier.fillMaxWidth()) {
        IncorrectGuessesText(viewModel)
        ...
    }
}
```

Модель представления передается новой функции.

Файл *GameFragment.kt* будет обновлен через несколько страниц. Но сначала попробуйте собрать воедино код двух оставшихся компонентов в следующем упражнении.



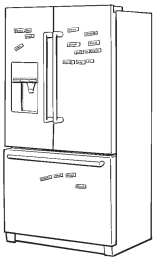
Развлечения с магнитами

Кто-то выложил на дверце холодильника код двух новых компонентных функций (`SecretWordDisplay` и `LivesLeftText`), заменяющих два оставшихся представления `GameFragment`. К сожалению, из-за беспорядка на кухне некоторые магниты упали на пол. Удастся ли вам восстановить код?

Функция `SecretWordDisplay` должна выводить свойство `secretWordDisplay` типа `LiveData<String>` из `GameViewModel`. Функция `LivesLeftText` должна выводить строковый ресурс `lives_left` и передавать ему свойство `LiveData<Int>` с именем `livesLeft` объекта `GameViewModel`. Обе функции должны реагировать на обновления данных Live Data.

```
@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {

    val display = viewModel.secretWordDisplay ..........

    display ..........{

        Text(..........)
    }
}

@Composable
fun LivesLeftText(viewModel: GameViewModel) {

    val livesLeft = viewModel.livesLeft ..........

    livesLeft ..........{

        Text(..........)

    }
}
```

Использовать все магниты необязательно.



Развлечения с магнитами. Решение

Кто-то выложил на дверце холодильника код двух новых компонентных функций (`SecretWordDisplay` и `LivesLeftText`), заменяющих два оставшихся представления `GameFragment`. К сожалению, из-за беспорядка на кухне некоторые магниты упали на пол. Удастся ли вам восстановить код?

Функция `SecretWordDisplay` должна выводить свойство `secretWordDisplay` типа `LiveData<String>` из `GameViewModel`. Функция `LivesLeftText` должна выводить строковый ресурс `lives_left` и передавать ему свойство `LiveData<Int>` с именем `livesLeft` объекта `GameViewModel`. Обе функции должны реагировать на обновления данных Live Data.

```
@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {
    val display = viewModel.secretWordDisplay .observeAsState()
    display .value ?.let {
        Text( it )
    }
}
```

Наблюдает за Live Data-свойством secretWordDisplay модели представления, как если бы оно было объектом State.

Значение переменной display выводится при условии, что оно отлично от null.

```
@Composable
fun LivesLeftText(viewModel: GameViewModel) {
    val livesLeft = viewModel.livesLeft .observeAsState()
    livesLeft .value ?.let {
        Text( stringResource ( R.string. lives_left , it ) )
    }
}
```

Наблюдает за Live Data-свойством livesLeft, как если бы оно было объектом State.

В качестве текста используется строковый ресурс lives_left, которому передается значение переменной livesLeft (при условии, что оно отлично от null).

Эти магниты не понадобились.

toState()

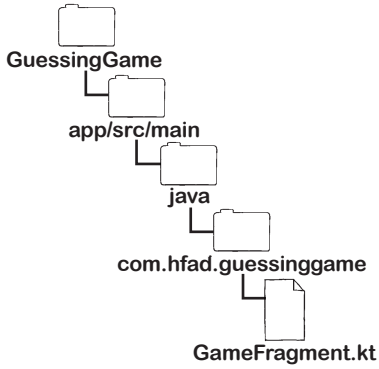
toState()

Обновленный код GameFragment.kt

Теперь вы знаете, как заменить все представления `GameFragment` компонентами Compose, и мы можем добавить компоненты в *GameFragment.kt*. Обновите файл (изменения выделены жирным шрифтом):

```
...
import androidx.compose.runtime.livedata.observeAsState
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.em
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Arrangement

class GameFragment : Fragment() {
    ...
}

@Composable
fun FinishGameButton(clicked: () -> Unit) { ... }

@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) { ... }

@Composable
fun GuessButton(clicked: () -> Unit) { ... }

@Composable
fun IncorrectGuessesText(viewModel: GameViewModel) {
    val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
    incorrectGuesses.value?.let {
        Text(stringResource(R.string.incorrect_guesses, it))
    }
}

@Composable
fun LivesLeftText(viewModel: GameViewModel) {
    val livesLeft = viewModel.livesLeft.observeAsState()
    livesLeft.value?.let {
        Text(stringResource(R.string.lives_left, it))
    }
}
```

Импортируйте эти классы.



Добавьте эти компонентные функции.

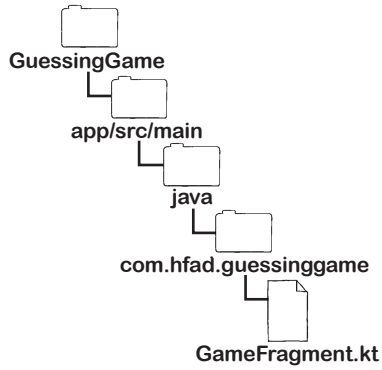
Продолжение на следующей странице.

ResultFragment
GameFragment

GameFragment.kt (продолжение)

Добавьте эту функцию.

```
@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {
    val display = viewModel.secretWordDisplay.observeAsState()
    display.value?.let {
        Text(text = it,
             letterSpacing = 0.1.em,
             fontSize = 36.sp)
    }
}

@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            SecretWordDisplay(viewModel)
        }
        LivesLeftText(viewModel)
        IncorrectGuessesText(viewModel)
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```



SecretWordDisplay выравнивается по центру в Row.

Добавьте компоненты LivesLeftText в IncorrectGuessesText в пользовательский интерфейс.

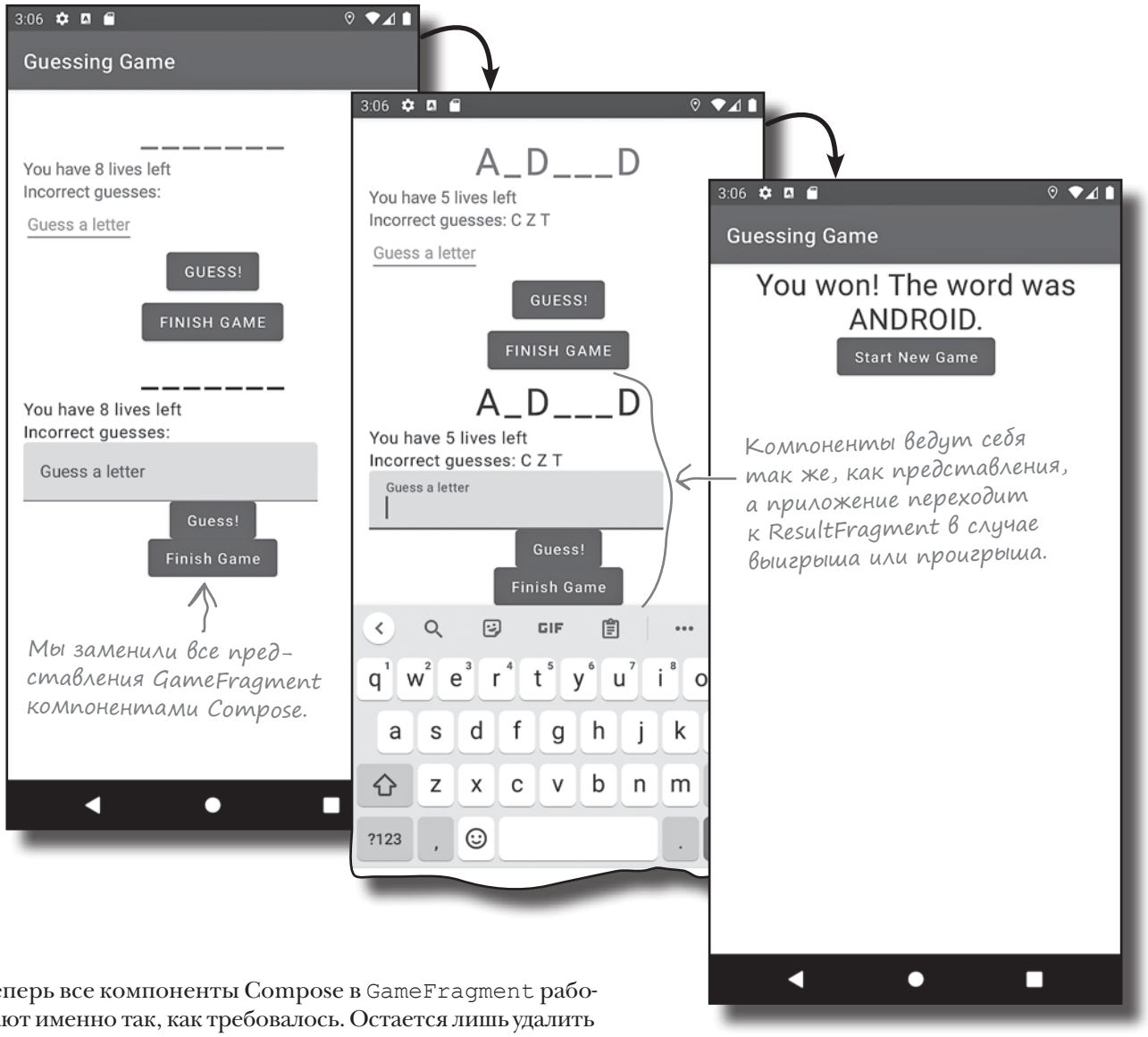
А теперь проведем тест-драйв приложения.

Тест-драйв

ResultFragment
GameFragment

При запуске приложения фрагмент `GameFragment` включает как все исходные представления, так и их аналоги Compose.

Когда пользователь вводит буквы, которые, по его мнению, могут входить в загаданное слово, текст в компонентах `SecretWordDisplay`, `LivesLeftText` и `IncorrectGuessesText` автоматически обновляется.



Теперь все компоненты Compose в `GameFragment` работают именно так, как требовалось. Остается лишь удалить представления из фрагмента.

ResultFragment
GameFragment

Удаление представлений из GameFragment.kt

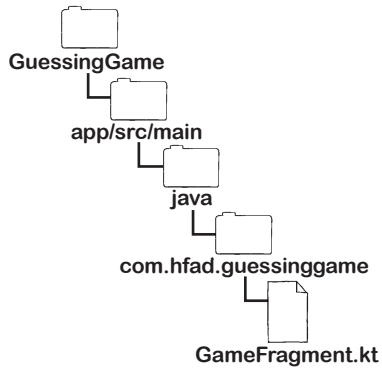
Как и в случае с `ResultFragment`, для исключения представлений из `GameFragment` мы удалим файл макета. Но сначала необходимо удалить все ссылки на представления из *GameFragment.kt*. Также будет отключено связывание данных, так как в новой версии оно не используется

Ниже приведен полный код новой версии; обновите файл *GameFragment.kt* (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer
import androidx.compose.runtime.Composable
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.runtime.livedata.observeAsState
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.em
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.ui.platform.ComposeView
```

Удалите эту строку.

Импортируйте этот класс.



Продолжение на следующей странице.

ResultFragment
GameFragment

GameFragment.kt (продолжение)

```
class GameFragment : Fragment() {
    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!
    lateinit var viewModel: GameViewModel

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view?.findNavController()?.navigate(action)
            }
        })

        _binding = FragmentGameBinding.inflate(inflater, container, false).apply {
        return ComposeView(requireContext()).apply {
            composeView.setContent {
                MaterialTheme {
                    Surface {
                        GameFragmentContent(viewModel)
                    }
                }
            }
        }

        val view = binding.root
        viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

        binding.gameViewModel = viewModel
        binding.lifecycleOwner = viewLifecycleOwner
```

Удалите эти строки.

Переместите код получения модели представления.

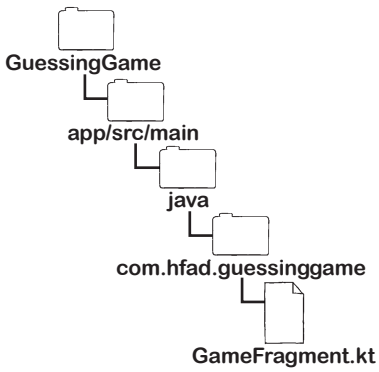
Переместите код перехода к ResultFragment при завершении игры.

Дополнительные ? необходимы, потому что «view» обращается к методу getView() фрагмента, который возвращает его корневое представление.

Удалите эту строку.

Удалите ссылку composeView.

Удалите эти строки.



Продолжение на следующей странице.

GameFragment.kt (продолжение)

ResultFragment
GameFragment

```
        viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
            if (newValue) {
                val action = GameFragmentDirections
                    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
                view.findNavController().navigate(action)
            }
        })
```

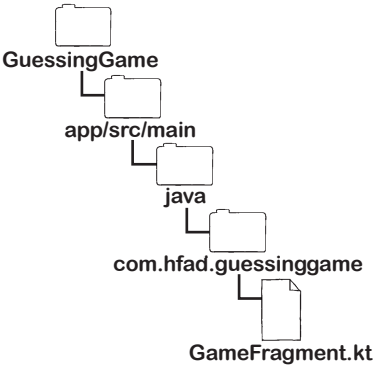
Удалите все эти строки, чтобы убрать ссылки на представления.

```
        binding.guessButton.setOnClickListener() {
            viewModel.makeGuess(binding.guess.text.toString().uppercase())
            binding.guess.text = null
        }
        return view
    }
```

Не удаляйте.

Удалите этот метод, он больше не нужен.

```
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}

@Composable
fun FinishGameButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Finish Game")
    }
}

@Composable
fun EnterGuess(guess: String, changed: (String) -> Unit) {
    TextField(
        value = guess,
        label = { Text("Guess a letter") },
        onValueChange = changed
    )
}
```

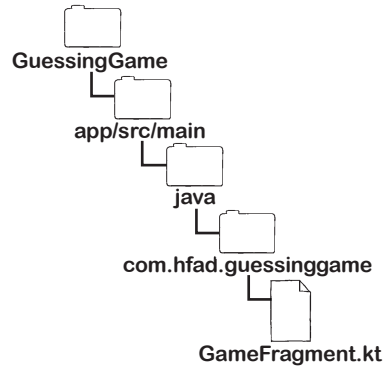


Продолжение на следующей странице.

GameFragment.kt (продолжение)

```
@Composable
fun GuessButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Guess!")
    }
}

@Composable
fun IncorrectGuessesText(viewModel: GameViewModel) {
    val incorrectGuesses = viewModel.incorrectGuesses.observeAsState()
    incorrectGuesses.value?.let {
        Text(stringResource(R.string.incorrect_guesses, it))
    }
}

@Composable
fun LivesLeftText(viewModel: GameViewModel) {
    val livesLeft = viewModel.livesLeft.observeAsState()
    livesLeft.value?.let {
        Text(stringResource(R.string.lives_left, it))
    }
}

@Composable
fun SecretWordDisplay(viewModel: GameViewModel) {
    val display = viewModel.secretWordDisplay.observeAsState()
    display.value?.let {
        Text(text = it,
             letterSpacing = 0.1.em,
             fontSize = 36.sp)
    }
}
```

Код на этой странице изменять не нужно.

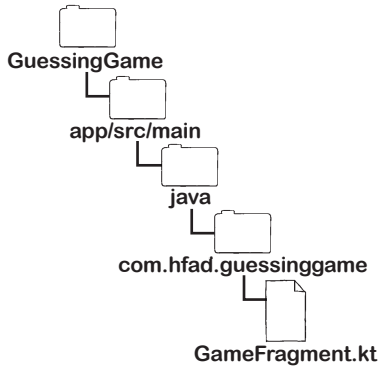


Продолжение на следующей странице.

ResultFragment
GameFragment

GameFragment.kt (продолжение)

```
@Composable
fun GameFragmentContent(viewModel: GameViewModel) {
    val guess = remember { mutableStateOf("") }

    Column(modifier = Modifier.fillMaxWidth()) {
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            SecretWordDisplay(viewModel)
        }
        LivesLeftText(viewModel)
        IncorrectGuessesText(viewModel)
        EnterGuess(guess.value) { guess.value = it }

        Column(modifier = Modifier.fillMaxWidth(),
            horizontalAlignment = Alignment.CenterHorizontally) {
            GuessButton {
                viewModel.makeGuess(guess.value.uppercase())
                guess.value = ""
            }
            FinishGameButton {
                viewModel.finishGame()
            }
        }
    }
}
```



Код на этой странице тоже остается без изменений.

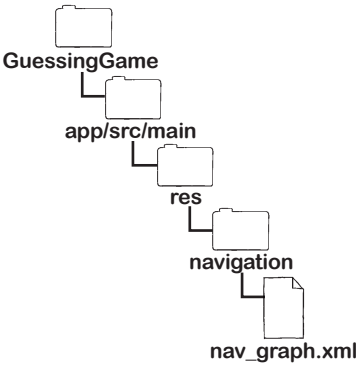
И это все изменения, которые необходимо внести в код *GameFragment.kt*, чтобы он не заполнял макет и не содержал ссылки на представления.

Так как фрагменту уже не нужен файл макета, после удаления всех ссылок на него из графа навигации этот файл можно удалить. Это будет сделано на следующей странице.

Удаление fragment_game.xml

Граф навигации включает ссылку на файл макета *fragment_game.xml*. Перед удалением файла макета ее необходимо удалить.

Откройте *nav_graph.xml* из папки *app/src/main/res/navigation* и удалите строку с упоминанием `"@layout/fragment_game"` из раздела `GameFragment`:




```
<fragment
    android:id="@+id/gameFragment"
    android:name="com.hfad.guessinggame.GameFragment"
    android:label="fragment_game"
    tools:layout="@layout/fragment_game" >
    <action ... />
</fragment>
```

Удалите эту ссылку.

Затем щелкните правой кнопкой мыши на файле *fragment_game.xml* на панели проекта, выберите команду Refactor и вариант Safe Delete. Если щелкнуть на кнопке OK и выбрать вариант проведения рефакторинга, файл будет удален.

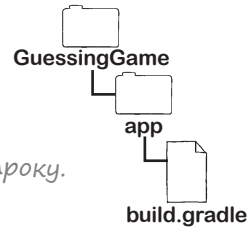
Отключение связывания данных

Остается внести последнее изменение в приложение Guessing Game: отключить связывание данных. Как вы помните, ранее мы включили связывание данных, чтобы представления из *fragment_game.xml* и *fragment_result.xml* могли взаимодействовать с моделью представления фрагментов. После того как файлы макетов будут удалены, связывание данных становится ненужным.

Чтобы отключить связывание данных, откройте файл *GuessingGame/app/build.gradle* и удалите строку связывания данных из раздела `buildFeatures`:

```
...
    buildFeatures {
        dataBinding true
        ...
    }
...
```

Удалите эту строку.



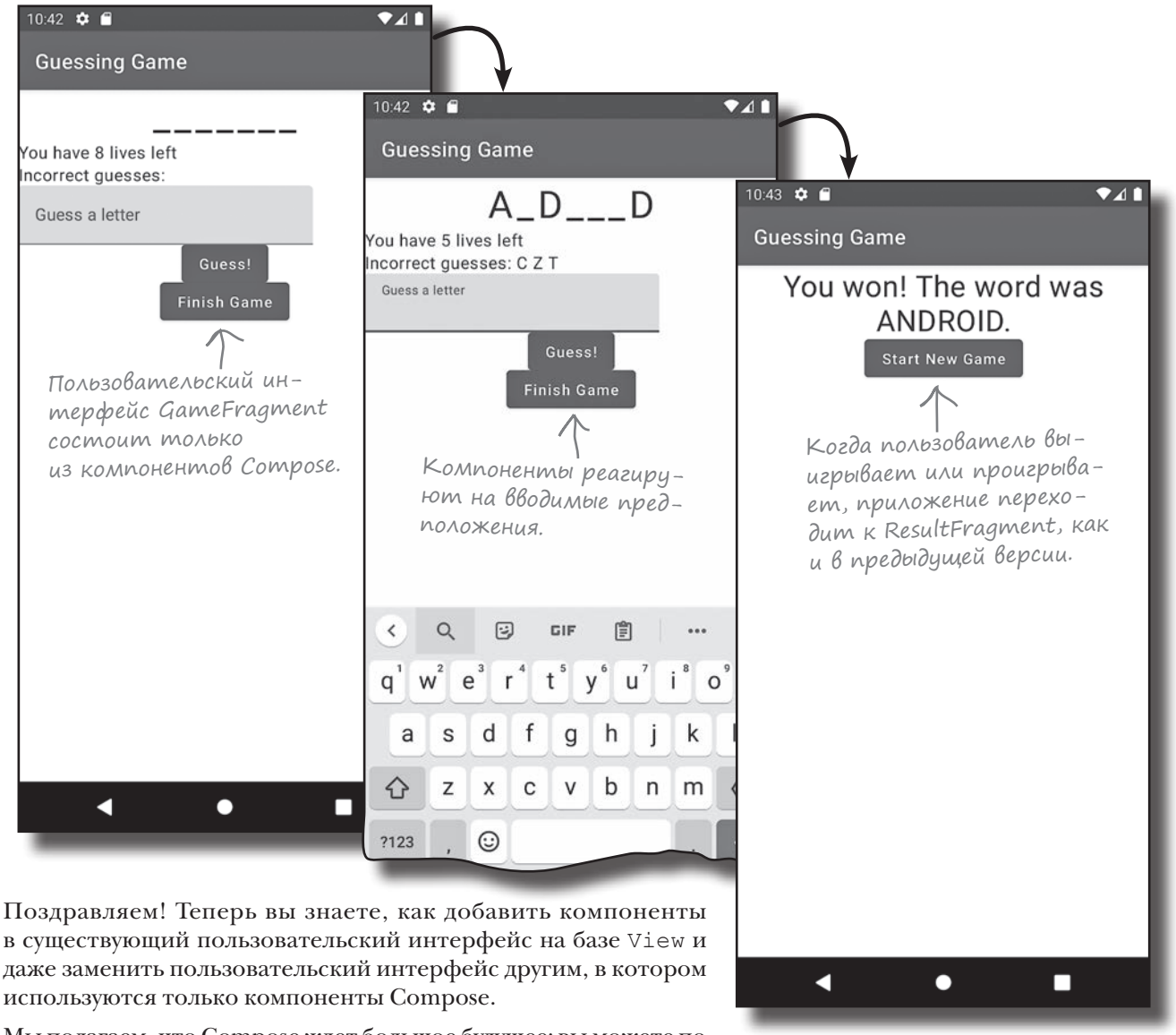
Когда это будет сделано, щелкните на ссылке Sync Now, чтобы синхронизировать изменения с остальными частями проекта.

Проведем итоговый тест-драйв приложения.

Тест-драйв

ResultFragment
GameFragment

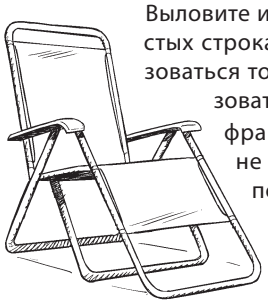
При выполнении приложения отображается фрагмент `GameFragment`. На этот раз пользовательский интерфейс строится исключительно из компонентов Compose. Они работают именно так, как нам нужно.



Поздравляем! Теперь вы знаете, как добавить компоненты в существующий пользовательский интерфейс на базе `View` и даже заменить пользовательский интерфейс другим, в котором используются только компоненты Compose.

Мы полагаем, что Compose ждет большое будущее; вы можете получить дополнительную информацию об этой технологии в приложении. А пока почему бы не добавить компоненты Compose в другие приложения, которые мы построили в этой книге?

У бассейна. Решение



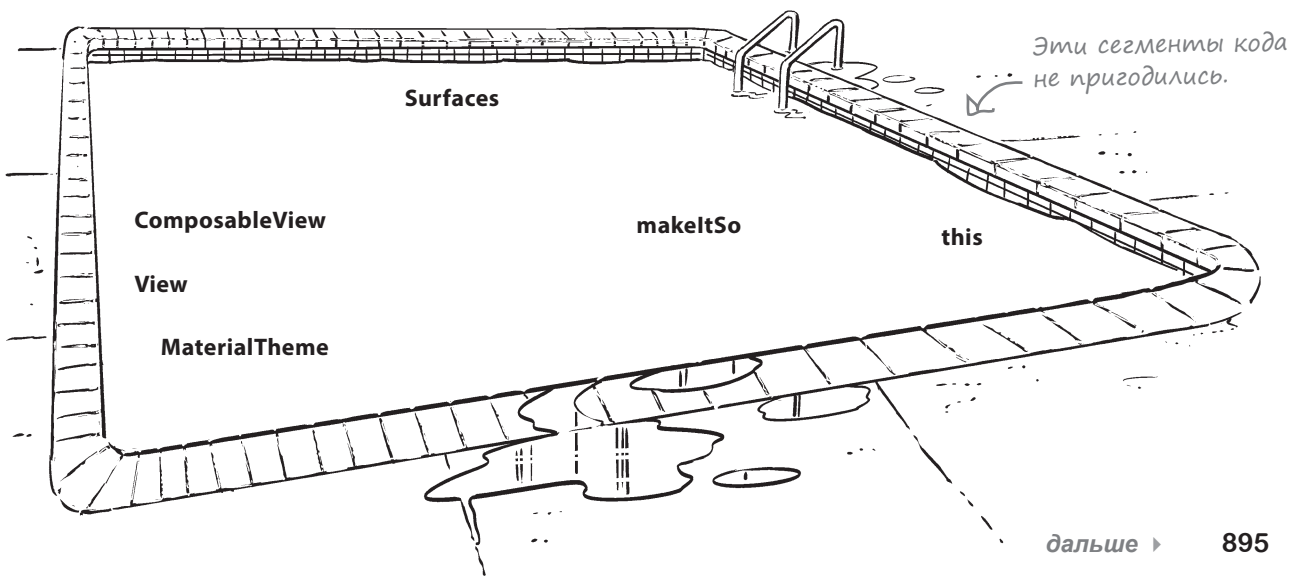
Выловите из бассейна сегменты кода и расставьте их в пустых строках в коде. Каждый сегмент кода может использоваться только один раз, при этом все сегменты использовать не обязательно. Ваша **задача** — написать код фрагмента с именем `MusicFragment`, который не имеет файла макета и использует для своего пользовательского интерфейса компонентную функцию с именем `MusicFragmentContent`. Тема с именем `MusicTheme` должна применяться к интерфейсу, включая все поверхности.

```
class MusicFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        return ComposeView ( requireContext() ) .apply {
            setContent {
                MusicTheme {
                    Surface {
                        MusicFragmentContent()
                    }
                }
            }
        }
    }
}
```

Возвращает ComposeView.

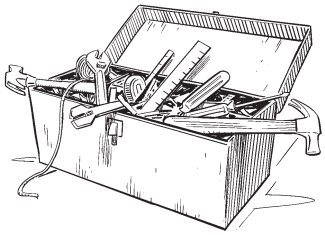
Метод setContent используется для добавления компонентов в пользовательский интерфейс.

Тема MusicTheme применяется к пользовательскому интерфейсу, включая поверхности.



Ваш инструментарий Android

ГЛАВА 19



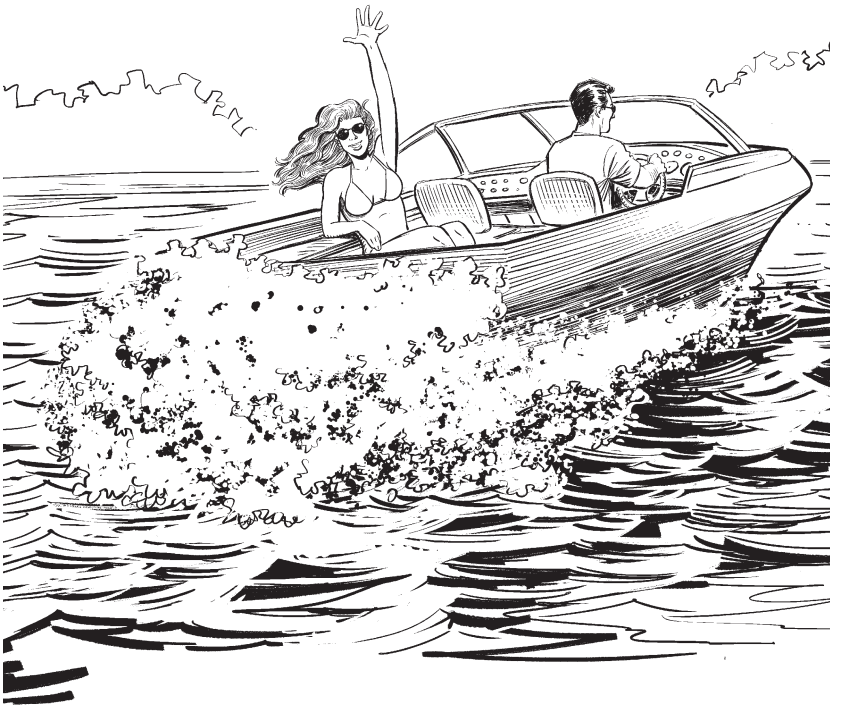
Глава 19 осталась позади, а ваш инструментарий пополнился интеграцией с Jetpack Compose.

Весь код для этой главы можно скачать: tinyurl.com/hfad3.

Ключевые моменты

- Приложения могут включать комбинации представлений `View` и компонентов Compose.
- Compose может использоваться в существующих приложениях при условии, что в них используется версия SDK 21 и выше и выбран язык Kotlin.
- Прежде чем использовать Compose в существующих пользовательских интерфейсах на базе `View`, необходимо добавить настройки и библиотеки Compose в файл *build.gradle*.
- `ComposeView` — представление, в которое можно добавлять компоненты Compose. Его можно рассматривать как своего рода заместитель для компонентов Compose, которые добавляются в файл макета.
- Компоненты Compose могут взаимодействовать со свойствами и методами модели представления.
- Используйте метод `observeAsState()` для того, чтобы компоненты Compose могли реагировать на обновления `Live Data`.
- После того как представления активности или фрагмента будут заменены компонентами Compose, файл макета можно будет удалить.

Пора расставаться



Было приятно видеть вас здесь, в Андроидвилле.

Жаль, что нам приходится расставаться, но пришло время применить на практике то, что вы узнали. Впрочем, в конце книги приводится кое-какая полезная информация, а потом можно переходить к самостоятельной работе. Всего доброго!

Десять важных тем, которые мы не рассмотрели

Даже после всего сказанного еще кое-что осталось. Нам хотелось бы рассмотреть еще ряд вопросов. Было бы неправильно делать вид, что их не существует; с другой стороны, нам не хотелось, чтобы нашу книгу можно было поднять только после интенсивной тренировки в спортзале. Прежде чем ставить книгу на полку, **ознакомьтесь с приложением**.

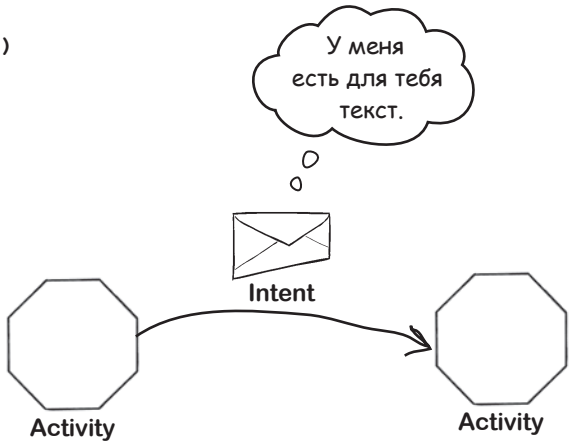
1. Совместное использование данных с другими приложениями

Если вы хотите организовать обмен простыми данными с другим приложением, это можно сделать с помощью интентов (**`Intent`**). Интент можно рассматривать как «намерение что-то сделать». Это разновидность сообщения, которая позволяет передать данные другому объекту (например, активности) во время выполнения.

Обмен данными с диспетчером Android

Если вы хотите передать текст другой активности, для этого можно воспользоваться кодом следующего вида:

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is some text.")
    type = "text/plain"
}
startActivity(sendIntent)
```



Сначала в коде создается объект `Intent` с именем `sendIntent`. Он использует свойство `type` для определения типа передаваемых данных (в нашем случае это простой текст) и метод `putExtra()` для присоединения данных (в нашем случае это некий текст).

Свойство `action` сообщает Android, какие типы активностей могут получать сообщение. В нашем случае значение задается следующей командой:

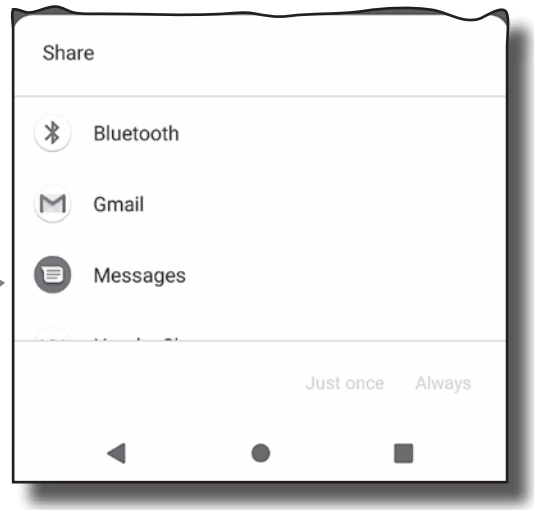
```
action = Intent.ACTION_SEND
```

Это означает, что сообщение Intent могут получать только активности, которые могут отправлять сообщения.

Отправка выполняется следующей командой:

```
startActivity(sendIntent)
```

Android ищет во всех приложениях на устройстве активности, способные получать интенты с заданным действием и типом. Если Android найдет несколько таких активностей, открывается экран определителя интентов (**Intent Resolver**). На нем пользователь может выбрать приложение, с которым он хочет обменяться данными.



Экран диспетчера Intent Resolver. В нем выводится список приложений на устройстве, способных отправлять простые текстовые данные.

Android запускает активность и передает ей данные.

Совместное использование данных через Android Sharesheet

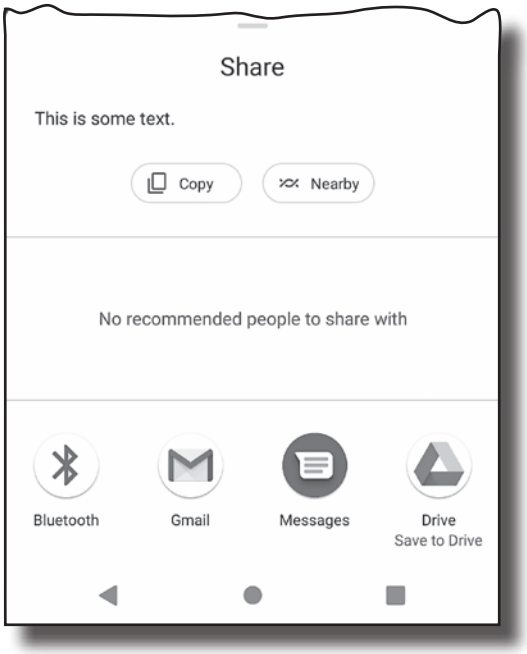
В большинстве случаев удобнее осуществлять совместное использование данных через Android Sharesheet. Этот вариант позволяет вам указать, с кем и как вы хотите совместно использовать данные. Код выглядит так:

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is some text.")
    type = "text/plain"
}
val shareIntent = Intent.createChooser(sendIntent, null)
startActivity(shareIntent)
```

Метод createChooser() используется для вызова Android Sharesheet.

Как видно из листинга, код включает дополнительный вызов метода `createChooser()` объекта `Intent`. При выполнении открывается экран Android Sharesheet:

Android Sharesheet превосходит Intent Resolver по гибкости и предоставляет больше возможностей.



Дополнительную информацию о совместном использовании данных с другими приложениями, а также о том, как организовать получение данных в ваших приложениях, можно найти по адресу

https://developer.android.com/training/sharing

2. WorkManager

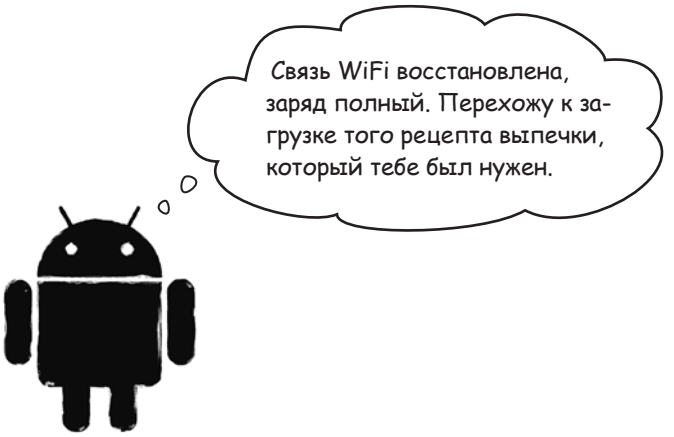
В некоторых ситуациях требуется, чтобы приложение обрабатывало данные в фоновом режиме, например если ему приходится обращаться к хранилищу данных или загружать большой файл по сети.

Как вы узнали в главе 14, для задач, которые должны запускаться немедленно, можно воспользоваться сопрограммами Kotlin. А если запуск задачи нужно отложить или требуется создать продолжительную задачу, выполнение которой должно продолжаться даже при перезапуске устройства?

Использование WorkManager для планирования задач с задержкой

Если вы хотите запланировать выполнение задачи в фоновом режиме, используйте **WorkManager** API. Этот интерфейс является частью Android Jetpack, он был спроекитрован для выполнения задач с большим временем выполнения, которые гарантированно должны продолжать выполнение даже при выходе пользователя из приложения или перезапуске устройства.

WorkManager даже можно использовать для запуска задач при выполнении некоторых условий (скажем, доступности WiFi) или формирования цепочек задач.



Дополнительную информацию о WorkManager — и о его использовании — можно найти по адресу

https://developer.android.com/topic/libraries/architecture/workmanager

3. Диалоговые окна и уведомления

В главе 9 вы узнали, как вывести простые сообщения на панелях Toast и Snackbar. Этот способ хорошо подходит для низкоприоритетных сообщений, которые отображаются в приложении и не требуют никаких действий со стороны пользователя.

Однако в некоторых ситуациях бывает нужно вывести сообщение, которое предлагает пользователю принять решение или отображается поверх пользовательского интерфейса приложения. В таких ситуациях обычно используются **диалоговые окна** и **оповещения**.

Диалоговые окна используются для принятия решений пользователем

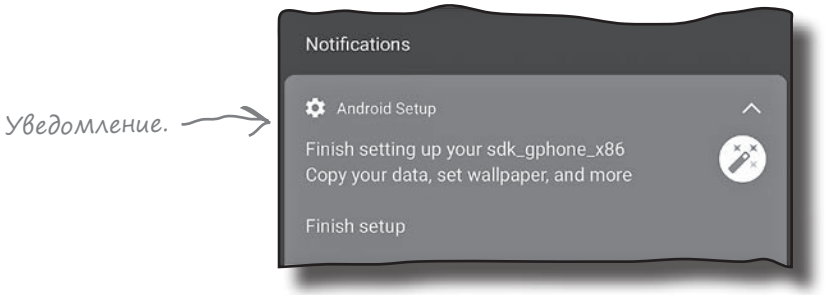
Диалоговым окном называется небольшое окно, отображающееся в середине экрана. Обычно диалоговые окна используются для ситуаций, когда пользователь должен принять решение для продолжения работы приложения:

Информация о создании и использовании диалоговых окон доступна по адресу

https://developer.android.com/guide/topics/ui/dialogs

Уведомления отображаются в пользовательском интерфейсе приложения

Если вы хотите напомнить пользователю о необходимости что-то сделать или оповестить о получении сообщения, используйте уведомления. Уведомления отображаются на панели состояния устройства и панели уведомлений, а также могут появляться на экране блокировки устройства:



Информация о создании и использовании уведомлений доступна по адресу

https://developer.android.com/guide/topics/ui/notifiers/notifications

4. Автоматизированное тестирование

Если ваше приложение, предназначенное для тысяч и даже миллионов пользователей, будет работать нестабильно, вы будете быстро терять пользователей. Впрочем, автоматизация тестирования позволяет избежать большинства проблем такого рода.

Среди тестовых фреймворков особенно популярны JUnit и Espresso. Когда вы создаете новый проект Android, Android Studio обычно включает зависимости для этих фреймворков в файл *build.gradle* приложения.

Средства автоматизированного тестирования обычно делится на две категории: **модульные тесты** и **инструментальные тесты**.

Модульные тесты

Модульные тесты работают на машине разработки и проверяют отдельные части (модули) вашего кода. Они размещаются в папке *app/src/test* вашего проекта и выглядят примерно так:

```
package com.hfad.myapp

import org.junit.Test
import org.junit.Assert.*

class ExampleUnitTest {
    @Test
    fun additionIsCorrect() {
        assertEquals(6, 3 + 3)
    }
}
```

Инструментальные тесты

Инструментальные тесты работают в эмуляторе или на физическом устройстве и проверяют работу полного приложения. Они размещаются в папке *app/src/androidTest* вашего проекта.

Пример инструментального теста приведен на следующей странице.

Пример инструментального теста

Ниже приведен инструментальный тест, который проверяет, выводится ли в компоненте Compose правильный текст:

```
package com.hfad.myapp

import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.ui.test.junit4.createComposeRule
import androidx.compose.ui.test.onNodeWithText
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class HelloTest {
    @get:Rule
    val composeTestRule = createComposeRule()

    @Test
    fun shouldShowHello() {
        composeTestRule.setContent {
            MaterialTheme {
                Surface {
                    Hello("Fred")
                }
            }
        }
        composeTestRule.onNodeWithText("Hello Fred!").assertExists()
    }
}
```

Дополнительная информация об автоматизированном тестировании в приложениях Android доступна по адресу

https://developer.android.com/training/testing

5. Поддержка разных размеров экрана

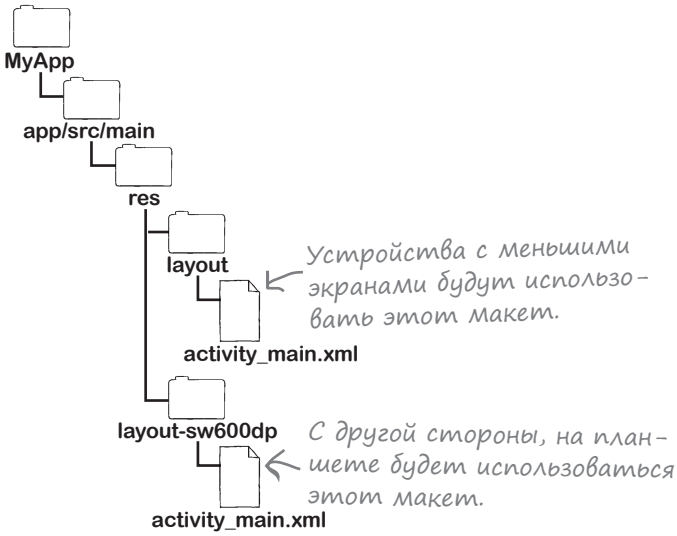
Существует множество устройств Android с разными формами и размерами экранов, и вы хотите, чтобы ваши приложения хорошо смотрелись на всех экранах. Вы можете использовать ряд полезных средств, включая макеты с ограничениями (о которых вы узнали в главе 4), определения альтернативных макетов и макеты с выдвижными панелями.

Определение альтерамивных макетов

Приложения Android могут содержать разные версии одного файла макета, предназначенные для разных спецификаций экранов. Для этого создаются разные папки макетов, которым присваиваются соответствующие имена, и в каждую папку добавляется отдельный макет.

Чтобы определить один макет для устройств с 600dp и выше (например, 7-дюймовых планшетов) и другой для устройств меньшего размера, создайте в папке *app/src/main/res* новую папку с именем *layout-sw600dp*. Затем поместите макет, который должен использоваться для больших устройств, в папку *layout-sw600dp*, а макет для меньших устройств — в папку *layout*.

Если приложение выполняется на телефоне, оно использует макет из папки *layout*, как обычно, а при выполнении на устройстве с большим экраном будет использоваться макет из папки *layout-sw600dp*.



Дополнительная информация об использовании квалификаторов ширины доступна по адресу

https://developer.android.com/training/multiscreen/screensizes#alternative-layouts

Использование SlidingPaneLayout

Некоторые макеты содержат списки записей, у которых по щелчку отображается подробная информация об элементе. На меньших устройствах подробная информация может отображаться на отдельном экране, но на больших устройствах вы, возможно, предпочтете выводить список и подробную информацию параллельно.

В таких ситуациях можно воспользоваться макетом `SlidingPaneLayout`, чтобы определить отдельные панели для списка и подробной информации. Код выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.slidingpanelayout.widget.SlidingPaneLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="280dp"
        android:layout_height="match_parent"
        android:layout_gravity="start" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="300dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />
</androidx.slidingpanelayout.widget.SlidingPaneLayout>
```

Панель для списка (представление с переработкой).

Панель для вывода подробной информации (FragmentContainerView).

Макет использует атрибуты `layout-width` каждой панели для определения того, как они должны размещаться рядом на устройстве. Если устройство имеет достаточно широкий экран, то макет отображает их рядом друг с другом, а если нет, то они отображаются на разных экранах.

Дополнительную информацию можно найти по адресу

https://developer.android.com/reference/androidx/slidingpanelayout/widget/SlidingPaneLayout

6. Другие возможности Compose

В главах 18 и 19 рассматривалась тема построения пользовательских интерфейсов на базе Compose. По нашему мнению, у этой технологии большое будущее, но на момент написания книги она была настолько новой, что некоторые библиотеки работали нестабильно. Мы решили, что перед расставанием стоит бросить взгляд в будущее; ниже перечислены некоторые библиотеки и средства, которые показались нам наиболее интересными.

Библиотека Compose ViewModel

В главе 19 мы передавали существующую модель представления компонентным функциям, чтобы они могли обращаться к свойствам и методам модели представления. При использовании библиотеки модели представлений Compose это становится излишним; компонент Compose может иметь свою модель представления.

Включите следующую зависимость в файл *build.gradle* приложения:

```
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:1.0.0-alpha07"
```

Затем добавьте следующий код в компонент Compose:

```
val viewModel: ResultViewModel = viewModel(
    factory = ResultViewModelFactory(result)
)
```

Эта строка необходима только в том случае, если модель представления должна генерироваться с использованием фабрики моделей представлений.

Библиотека макетов с ограничениями

В главе 18 было показано, как разместить компоненты Compose по строкам и столбцам. Но если вам понадобятся более гибкие средства размещения, в вашем распоряжении библиотека макетов с ограничениями. Как нетрудно догадаться, эта библиотека дает возможность размещать компоненты Compose с использованием ограничений.

Если вам захочется больше узнать об этой библиотеке и о том, как ее использовать, информация доступна по адресу

https://developer.android.com/jetpack/compose/layouts/constraintlayout

Компонент Navigation для Compose

Как вам уже известно, для перехода между экранами в пользовательских интерфейсах на базе `View` используется компонент Navigation. Для каждого экрана определяется отдельный фрагмент, а компонент Navigation решает, какой фрагмент должен отображаться.

При использовании навигации Compose фрагменты не нужны.

При использовании Compose-компонента Navigation применять фрагменты не обязательно. Вместо этого для каждого экрана определяется *компонент Compose*, а компонент Navigation решает, какой из компонентов должен отображаться.

Чтобы использовать Compose-компонент Navigation, добавьте следующую зависимость в файл *build.gradle* приложения:

```
implementation("androidx.navigation:navigation-compose:2.4.0-alpha08")
```

Затем воспользуйтесь этим кодом:

```
package com.hfad.myapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.Text
import androidx.compose.material.Button
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.runtime.Composable
import androidx.navigation.NavController
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface {
                    MainActivityContent()
                }
            }
        }
    }
}
```

Продолжение на следующей странице.

Код навигации Compose (продолжение)

```
@Composable
fun ScreenOne(navController: NavController) {
    Button(onClick = { navController.navigate("two") }) {
        Text("Navigate to screen two")
    }
}

@Composable
fun ScreenTwo() {
    Text("This is screen two")
}

@Composable
fun MainActivityContent() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "one") {
        composable("one") { ScreenOne(navController) }
        composable("two") { ScreenTwo() }
    }
}
```

Экран ScreenOne содержит кнопку Button для перехода к ScreenTwo.

Экран ScreenTwo содержит компонент Text.

При выполнении приложение выглядит так:

За свежей информацией о Compose обращайтесь по адресу

https://developer.android.com/jetpack/compose

7. Retrofit

В главе 14 вы узнали, как сохранять информацию в базе данных Room на устройстве. Но что, если данные должны храниться на сетевом сервере?

В этом случае можно воспользоваться Retrofit. Это сторонний REST-клиент для Android, Java и Kotlin, который позволяет выдавать сетевые запросы, взаимодействовать с REST API и загружать данные в формате JSON и XML.

Дополнительная информация о Retrofit доступна по адресу

https://square.github.io/retrofit/

Если вам захочется больше узнать об использовании Retrofit в архитектуре ваших приложений, зайдите на следующую страницу:

https://developer.android.com/jetpack/guide

8. Android Game Development Kit

Если вас интересует тема разработки игр для Android, вам стоит поближе познакомиться с Android Game Development Kit (AGDK). Это набор библиотек и программных средств для разработки, оптимизации и публикации игр для Android.

Android Game Development Kit включает библиотеки для разработки игр C/C++. Например, библиотека `GameActivity` наследует от `AppCompatActivity` и использует API языка C. Также реализованы такие возможности, как использование экранной клавиатуры из кода C и обработка ввода с игровых устройств.

За дополнительной информацией об Android Game Development Kit обращайтесь по адресу

https://developer.android.com/games/agdk

9. CameraX

Если вы хотите использовать камеру устройства в своем приложении, в Jetpack имеется библиотека, которая называется CameraX. Она предоставляет единый API, работающий на большинстве устройств Android, и решает проблемы совместимости устройств. Библиотека может использоваться для предварительного просмотра, сохранения и анализа графических изображений. Для нее даже существует дополнение Extensions, которое открывает доступ к возможностям встроенного приложения для работы с камерой, поставляемого с устройством.

Дополнительная информация о CameraX доступна по адресу

https://developer.android.com/training/camerax

10. Публикация приложения

После того как разработка приложения будет завершена, вероятно, вы захотите открыть к нему доступ другим пользователям. И скорее всего, для этого вы захотите опубликовать свое приложение в магазине приложений — таком, как Google Play.

Этот процесс состоит из двух стадий: подготовки приложения публикации и собственно выпуска.

Подготовка приложения к публикации

Прежде чем выпускать приложение, необходимо настроить конфигурацию, построить и протестировать его рабочую версию. На этом этапе решаются такие задачи, как выбор значка для приложения, удаление отладочного кода и изменение файла *AndroidManifest.xml*, чтобы приложение могло загружаться только теми устройствами, на которых оно должно работать.

Прежде чем публиковать приложение, обязательно протестируйте его *как минимум* на одном планшете и одном телефоне; убедитесь в том, что оно выглядит так, как ожидалось, и работает с приемлемым быстродействием.

За дополнительной информацией о подготовке приложения к публикации обращайтесь по адресу

http://developer.android.com/tools/publishing/preparing.html

Выпуск приложения

На этой стадии разработчик публикует приложение, продает его и распространяет информацию о нем.

Прежде чем публиковать приложение, мы рекомендуем ознакомиться со следующими ресурсами:

https://developer.android.com/distribute/best-practices/launch

Здесь доступны контрольные списки и рекомендации, упрощающие публикацию приложения и управление им.

Некоторые идеи относительно того, как лучше представить свое приложение пользователям и сформировать его репутацию среди пользователей, мы рекомендуем изучить в следующих документах:

https://developer.android.com/distribute/best-practices/engage

и:

https://developer.android.com/distribute/best-practices/grow