Become an Xcoder

Русское издание

Начни программировать под Mac OS X используя Objective-C

Bert Altenberg, Alex Clarke, Philippe Mougin

Лицензия

Уведомление об авторском праве

CocoaLab

CocoaLab делает эту книгу общедоступной в виде Wiki или pdf документа на различных языках, и размещенной на http://www.cocoalab.com.

Оглавление

Введение

Apple обеспечивает вас всеми инструментами, необходимыми для создания великолепных Cocoa приложений бесплатно. Этот набор инструментов известен как Xcode, поставляется вместе с Mac OS X. Также вы можете скачать его из раздела разработчиков на веб-сайте Apple.

Существует несколько хороших книг по программированию для Mac, но в них предполагается, что у вас уже есть некоторый опыт в программировании. Это книга не такая. Она научит вас основам программирования, а конкретно Objective-C, используя Xcode. После 5 глав вы будите способны создать простую программу без графического интерфейса пользователя (GUI).

После прочтения нескольких глав вы будете знать, как создавать простые программы с GUI. Когда вы прочитаете этот книгу, вы сможете понять более сложные книги. Вы должны будете прочитать и их, т.к. там есть чему поучиться. Но пока не беспокойтесь об этом — эта книга дается легко.

Как пользоваться этой книгой

Как вы видите, некоторые абзацы отображаются отлично от основного текста подобно этому:

> *Некоторые детали, на которые стоит обратить внимание.*

Мы рекомендуем читать каждую главу (по крайней мере) дважды. При первом прочтении пропустите дополнительные вставки. При втором прочтении читайте вместе с дополнительными вставками. Вы повторите то, что вы уже изучили, а также узнаете некоторые интересные детали, которые были пропущены в первый раз. Читая книгу таким способом, вы неизбежно сделаете кривую вашего обучения похожей на крутой склон.

Эта книга содержит десятки примеров, состоящих из одной или более строк программного кода. Чтобы связать объяснения с примером, каждый пример имеет ссылку в виде числа в квадратных скобках, например, [1]. Большинство примеров из двух или более строк кода. Иногда, второе число используется для обозначения конкретной строки. Например, [1.2] относится к первой строке примера [1]. При больших фрагментах кода, ссылка ставится после строки кода, например:

```
//[1]
volume = baseArea * height; // [1.2]
```

Программирование — непростая работа. С вашей стороны требуется некоторая настойчивость и попытка опробовать все вещи в этой книге на себе. Вы не сможете научиться играть на фортепиано или водить машину только читая книги. То же самое можно сказать об обучении программированию. Эта книга в электронном формате, поэтому вы всегда имеете возможность переключиться на Xcode. Как и в главе 5, мы предлагаем вам пройти каждую главу три раза. Во второй раз попробуйте реализовать примеры, а затем внести небольшие изменения в код, чтобы изучить работу.

Перед началом

Мы написали эту книгу для вас. Так как она бесплатная, пожалуйста, позвольте мне сказать пару слов о содействии Mac.

Каждый пользователь Macintosh может помочь в продвижении их любимой компьютерной платформы с небольшими усилиями. Опишу как.

Чем более искусным пользователем Mac вы являетесь, тем легче вам будет убедить других людей его использовать. Поэтому оставайтесь в курсе событий, посещая сайты и читая журналы о Mac. Естественно, изучение Objective-C или AppleScript, и использование их на практике тоже необходимо. Для целей бизнеса использование AppleScript может сохранить много времени и денег.

Покажите миру, что не все используют PC, показывайте Macintosh. Можно носить на людях красивую майку с логотипом Mac, но есть способы показать Mac, даже находясь у себя дома. Если вы запустите Мониторинг системы (Приложения ▸ Служебные программы ▸ Мониторинг системы), Вы увидите, что ваш Mac редко использует все ресурсы своего процессора. Ученые создали несколько проектов для распределенных вычислений (РВ), например Folding@home или SETI@home, которые используют эти свободные ресурсы, обычно для общего блага.

Вы можете загрузить небольшую бесплатную программу РВ и начать обрабатывать специальные блоки данных для сети распределенных вычислений. Этот программа имеет самый низкий приоритет исполнения на всех компьютерах. Когда вы пользуетесь своим Mac, вам нужна вся его мощность, и программа РВ автоматически останавливается. Она не выгружается, но вы не заметите, что она работает. Как это помогает Mac? Большинство РВ-проектов занимается ранжированием сайтов. Если вы присоединитесь к команде Mac, вы можете внести ценный вклад для расчета ранжирования страниц. И пользователи других платформ могут видеть, как работает команда Mac. РВ могут использоваться и для других целей: математический расчетов, поиск лекарства и т.д. Выберите свой проект здесь:

http://distributedcomputing.info/projects.html

Только будьте осторожны, это может войти в привычку!

Будьте уверены — на платформе Macintosh лучшие программы. Но их очень сложно писать одному. Возьмите в привычку обратную связь с разработчиками программ, которые вы используете. Даже если вам что-либо не нравится в программе, расскажите об этом разработчикам. В докладе об ошибке предоставьте как можно более точное и полное описание ошибки и действий, которые к ней привели.

Платите за софт, который используете. Пока рынок ПО под Macintosh будет жив, разработчики будут продолжать создавать отличные программы.

Пожалуйста, свяжитесь хотя бы с тремя пользователями Macintosh, которых интересует программирование, и расскажите им об этой книге и где ее найти.

Хорошо, пока вы скачиваете программу РВ, давайте начнем!

Глава 1: Программа — это набор инструкций

Учась водить машину, вам приходится держать в голове сразу несколько вещей. Нужно одновременно помнить о педалях сцепления, газа и тормоза. Программирование тоже требует от вас внимания ко многим деталям, чтобы программа работала без сбоев. Когда вы начинаете учиться водить машину, вы уже так или иначе знакомы с ней. Когда же вы начинаете учиться программировать на Xcode, этого преимущества вы лишены. Поэтому, чтобы не взваливать на вас сразу слишком много, мы отложим разбор среды программирования до следующей главы. Сначала мы постараемся познакомить вас с кодом Objective-C, начиная со очень знакомых вам некоторых основ математики.

В начальных классах вы производили вычисления, заменяя многоточие на значение:

> *2 + 6 = ...*
> *... = 3 * 4 (звёздочка * используется для представления операции умножения на компьютерной клавиатуре)*

В старших классах многоточие вышло из моды и появились переменные, названные x и y (вместе с новым модным словом «алгебра»). Оглядываясь назад, можно подумать: и почему людей так пугает такое небольшое изменение в способе записи?

> *2 + 6 = x*
> *y = 3 * 4*

Переменные

В Objective-C тоже используются переменные. Переменные являются не более чем удобными именами для обозначения конкретных фрагментов данных, таких как числа. Ниже написано высказывание на Objective-C, а именно строка кода, где переменной присваивается определённое значение.

```
//[1]
x = 4;
```

Точка с запятой

Переменная x принимает значение 4. Следует заметить точку с запятой в конце строки. Точка с запятой нужна в конце каждой команды. Почему? Ну, код, приведенный в примере [1], может выглядеть понятным вам, но компьютер не знает что с ним делать. Специальная программа «компилятор» призвана перевести текст, который вы пишете, в единички и нолики, которые понимает ваш Mac. Компьютеру очень сложно понять введенный текст, например где кончается команда. Для этого и используется точка с запятой.

Если вы забудете поставить точку с запятой в вашем коде, код не скомпилируется, т.е. не переведется в код, который сможет выполнить Mac. Не сильно беспокойтесь об этом,

компилятор пожалуется, если не сможет скомпилировать код. Как мы увидим в следующей главе, он будет стараться помочь вам понять, что случилось.

Именование переменных

Хотя имена переменных и не имеют особого значения для компилятора, описательные имена переменных могут сделать программу гораздо проще для чтения и следовательно понимания людьми. Это большой плюс при отслеживании ошибок в коде.

> *Ошибки в программах по традиции называют «багами» (англ. bugs). Их поиск и исправление называют дебагингом.*

Таким образом в реальном коде следует избегать использования неописательных имен переменных вроде x. Например, переменную для ширины изображения можно было бы назвать `pictureWidth` [2].

```
//[2]
pictureWidth = 8;
```

Компиляторы обычно прощают отсутствие точки с запятой, но вы поймете, что программирование – это в основном детали. Например, нужно обращать внимание на заглавные и прописные буквы в названиях функций и переменных (case-sensitive). Например: `pictureWidth`, `pictureWIDTH` и `PictureWidth` – это три разные переменные. Согласно общим соглашениям, мы делаем переменные, соединяя несколько слов, первое слово без заглавных и все остальные начинаются с заглавных, как на примере [2]. Этот стиль часто называют camelCase. Выбрав этот метод, вы существенно сократите количество ошибок программирования, связанных с регистром.

Помните, что имя переменной обычно состоит из одного слова (или одного символа в крайнем случае).

> *Несмотря на то, что вы можете использовать любое имя для своих переменных, существуют определенные правила, которые необходимо соблюдать. Не буду пока перечислять их всех сразу, скажу лишь про одно, оно основное — переменные не должны совпадать с зарезервированными словами Objective-C. Используя переменные, состоящие из нескольких слов, например pictureWidth, вы никогда не нарушите это правило. И еще, старайтесь использовать строчные буквы в именах переменных, это обеспечит их читабельность и облегчит вам жизнь в дальнейшем. Если вам интересно узнать несколько правил, дочитайте это параграф. Кроме букв, можно пользоваться числами, но переменные не должны начинаться с них. Кроме этого допустим символ подчеркивания «_».*

Приведём несколько примеров названий переменных.

Правильные имена переменных:

```
door8k
do8or
do_or
```

Недопустимые:

```
door 8 (содежит пробел)
8door (начинается с цифры)
```

Не рекомендуется:

```
Door8 (начинается с заглавной буквы)
```

Использование переменных в вычислениях

Теперь, когда мы знаем, как присваивать переменным значения, мы можем осуществлять расчёты. Давайте посмотрим на код [3], осуществляющий расчет площади поверхности картинки.

```
//[3]
pictureWidth=8;
pictureHeight=6;
pictureSurfaceArea=pictureWidth*pictureHeight;
```

Удивительно, компилятор не обращает внимание на пробелы(за исключением имен переменных, команд и т.д.) Для большей читаемости кода вы можете использовать пробелы.

```
//[4]
pictureWidth = 8;
pictureHeight = 6;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

Целые и дробные

Теперь взглянем на пример [5], в частности на первые две строки.

```
//[5]
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

Числа можно разделить на два вида: целые и дробные. В выражениях [5.2] и [5.3] представлены оба вида. Целые используются для подсчета, например когда вам нужно сделать несколько повторений одной и той же инструкции (см. главу 7). Дробные или числа с плавающей точкой используются, например в вычислениях среднего числа попаданий в бейсболе.

Код из примера [5] не будет работать. Проблема в том, что комплятору нужно знать заранее какие именно переменные вы будете использовать в своей программе и на какие типы данных

они ссылаются, то есть целые числа или дробные с плавающей запятой. Это называется «объявить переменные»

```
//[6]
int pictureWidth;
float pictureHeight, pictureSurfaceArea;
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

В строке [6.2] `int` означает, что переменная `pictureWidth` целое число (integer). В следующей строке, мы объявляем сразу две переменные, это можно сделать при помощи запятой. Как вы можете видеть, обе переменные типа float, то есть могут хранить в них дробные числа. На самом деле, немного глупо то, что переменная `pictureWidth` отличается от других двух, потому что если вы умножите переменную типа `int` на переменную типа `float` результат расчетов будет `float`. Вот почему вам стоит объявить `pictureSurfaceArea` как `float` [6.3].

Почему компилятор должен знать, представляет ли переменная собой целое число или число с дробной частью?

Хорошо, компьютерным программам нужна часть компьютерной памяти. Компилятор резервирует место в памяти (байты) для каждой переменной в программе. Так как различные типы данных, в данном случае целые `int` и дробные `float`, требуют разные объемы памяти и имеют различные представления, компилятор должен зарезервировать правильный объем памяти и использовать верные представления о типе данных.

Что, если мы работаем с очень большими числами или десятичными числами очень высокой точности? И они не будут помещаться в несколько байтов, которые выделит для них компилятор, что с ними станет? Есть два ответа на этот вопрос: во-первых, числа как целые так и с плавающей точкой, имеют аналоги, которые могут хранить большие числа (или числа более высокой точности). В большинстве систем это `long` и `double`, соответственно. Но даже они могут переполняться, что приводит нас ко второму ответу: это будет ваша (как программиста) работа — отслеживать ошибки. В любом случае, это не та проблема, которая будет обсуждаться в первой главе.

Кстати, целые десятичные числа могут быть отрицательным, как вы знаете, например ваш банковский счет. Если вы знаете, что значение переменной никогда не бывает отрицательным, можно увеличить диапазон значений, которые вписываются в число доступных байтов.

```
//[7]
unsigned int chocolateBarsInStock;
```

Увы, нет такого понятия, как отрицательное число шоколадок на складе, так что беззнаковые целые `unsigned int` могли бы быть использованы здесь. Тип `unsigned int` представляет собой целые числа большие или равные нулю.

Объявление переменных

Есть возможность объявить и инициализировать переменную за один шаг.

```
//[8]
int x = 10;
float y= 3.5, z = 42;
```

Это позволит вам несколько сэкономить на написании кода.

Типы данных

Как мы только что увидели, данные, хранящиеся в переменной могут быть одним из нескольких конкретных типов, для примера целые и дробные числа.

В Objective-C, такие простые типы данных как эти, также известны как скалярные типы данных. Здесь приведен список общих скалярных типов данных доступных в Objective-C:

Имя	Тип	Пример
`void`	пусто	ничего
`int`	целое число	-1, 0, 1, 2
`unsigned`	беззнаковое целое число	0, 1, 2
`float`	числа с плавающей точкой	-0.333, 0.5, 1.23
`double`	числа с плавающей точкой удвоенной точности	0.5252525233323409
`char`	символ	a, b, c
`BOOL`	логический	0, 1, TRUE, FALSE, YES, NO

Математические операции

В предыдущем примере, мы выполнили операцию умножения. Используйте следующие символы, называемые операторами, для выполнения простых математических расчетов.

```
+ для сложения
- для вычитания
/ для деления
* для умножения
```

Используя операторы, мы можем выполнять широкий спектр расчетов. Если вы посмотрите на код профессиональных Objective-C программистов, вы найдете пару особенностей, возможно, потому что они ленивы.

Вместо записи вида `x = x + 1;` программисты часто прибегают к чему-то другому, как [9] или [10].

```
//[9]
x++;
//[10]
++x;
```

В любом случае это означает: увеличение x на одну единицу. В некоторых случаях важно, используется ++ до или после имени переменной. Обратите внимание на примеры [11] и [12].

```
//[11]
x = 10;
y = 2 * (x++);
//[12]
x = 10;
y = 2 * (++x);
```

В примере [11], после его выполнения, `y = 20`, а `x = 11`. В отличие от этого, в примере [12.2], увеличение x на единицу происходит до умножения на 2. Так, в конце концов получается, что `x = 11`, а `y = 22`. Код примера [12] эквивалентен примеру [13] .

```
//[13]
x = 10;
x++;
y = 2 * x;
```

Итак, программист фактически объединил два выражения в одно. Лично я считаю, что это делает программу трудно читаемой. Если вы экономите время и силы, то это прекрасно, но имейте ввиду, то что там может скрываться ошибка.

Скобки

Это будет банально, если вам удалось пройти среднюю школу, но скобки могут использоваться для определения порядка, выполнения операций. Обычно * и / имеют больший приоритет над + и –. Поэтому 2 * 3 + 4 = 10. А с помощью скобок, вы можете заставить это скромное сложение выполняться первым: 2 * (3 + 4) = 14.

Деление

Оператор деления заслуживает особого внимания, потому что он выполняет совершенно разные действия в зависимости от того, работает ли он с целыми числами или вещественными. Взгляните на следующие примеры [14, 15].

```
//[14]
int x = 5, y = 12, ratio;
ratio = y / x;
//[15]
float x = 5, y = 12, ratio;
ratio = y / x;
```

В первом случае [14], результатом будет 2. И только во втором случае [15], результатом будет именно то, чего вы, вероятно, ожидали: 2.4.

Логические операнды

Булевый тип является простым логическим представлением значений `true` или `false`. `1` (YES) и `0` (NO) отождествляемые с `true` и `false` значениями часто используются как взаимозаменяемые, и их можно считать эквивалентными:

TRUE	FALSE
1	0
YES	NO

Их часто используют, когда нужно принять решение о выполнении некоторого действия в зависимости от логического значения переменной или результата работы функции.

Остаток от деления

Оператор с которым вы, вероятно, знакомы % (остаток от деления). Он работает не так, как вы могли бы ожидать: оператор остатка от деления не используется для вычислений с процентами. Результатом оператора % будет остаток от целочисленного деления первого операнда на второй (если значение второго операнда равно нулю, то поведение оператора % неопределено).

```
//[16]
int x = 13, y = 5, remainder;
remainder = x % y;
```

Теперь значение remainder равно 3, потому что x равняется 2*y + 3.

Вот несколько примеров на остаток от деления:

```
21 % 7 равно 0
22 % 7 равно 1
23 % 7 равно 2
24 % 7 равно 3
27 % 7 равно 6
30 % 2 равно 0
31 % 2 равно 1
32 % 2 равно 0
33 % 2 равно 1
34 % 2 равно 0
50 % 9 равно 5
60 % 29 равно 2
```

Иногда он действительно может очень пригодиться, но учтите что работает он только с целочисленными переменными.

Часто этот оператор применяют, чтобы выяснить является ли число четным. Если оно четное, остаток от деления на два будет нулевой. Или ненулевой в противном случае. Вот пример:

```
//[17]
int anInt;
//Некий код, меняющий значение переменной anInt
if ((anInt % 2) == 0)
{
  NSLog(@"Чётное");
}
else
{
  NSLog(@"Нечётное");
}
```

Глава 2: Без комментариев? Нет, только не это!

Используя «говорящие» имена переменных, мы можем сделать наш код более читабельным и понятным [18].

```
//[18]
float pictureWidth, pictureHeight, pictureSurfaceArea;
pictureWidth = 8.0;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

До сих пор наши примеры кода содержали лишь несколько выражений, но даже очень простая программа может быстро вырасти до сотен или тысяч строк. Пересматривая свой код спустя несколько недель или месяцев, вам, возможно, будет трудно вспомнить почему вы поступили именно так. Здесь на помощь приходят комментарии. Комментарии помогут вам быстро понять свой код, какие части что делают, почему и в каком порядке. Некоторые программисты идут так далеко, что пишут комментарии к классам до начала кодирования. Это помогает им организовать собственное мышление и избегать ненужных действий.

Рекомендуется тратить какое-то время на комментирования своего кода. Мы можем вас уверить, вы сэкономите значительно больше времени из-за этого в будущем. Также, если вы отдадите ваш код кому-нибудь, ваши комментарии помогут ему быстрее адаптировать его к собственным нуждам.

Создание комментария

Чтобы создать комментарий, просто начните строку с двух символов косой черты.

```
// Это комментарий
```

В Xcode комментарии отображаются зеленым цветом. Если комментарий длинный и многострочный, поместите его между `/*` и `*/`.

```
/* Это комментарий
из двух строк */
```

Исключение блоков кода

Мы обсудим отладку программ в ближайшее время, Xcode имеет большие возможности для этого. Один из старинных способов отладки называется *outcommenting*. Разместив часть кода между `/* */`, вы можете временно отключать часть кода, чтобы проверить, работает ли остальная часть кода как и ожидалось. Это позволяет отловить ошибку. Если *outcommented* часть должна присвоить, например, значение конкретной переменной, вы можете включить временное выражение, где вы установите значение переменной пригодное для тестирования оставшейся части вашего кода.

Для чего нужны комментарии?

Важность комментариев трудно переоценить. Часто бывает полезно добавить пояснение (на простом человеческом языке) о том, что происходит в длинной последовательности выражений. Это позволит не ковыряться в коде, пытаясь понять что он делает, а сразу понять — является ли он частью проблемы, которую вы сейчас устраняете. Вы должны также использовать комментарии, чтобы выразить вещи, которые трудно или невозможно понять из кода. Например, если программа выполняет математические функции с помощью конкретной модели, подробно описанной в книге, вы должны поставить библиографические ссылки в комментарии, связанные с вашим кодом.

Иногда бывает полезно записать несколько комментариев перед написанием кода. Это поможет вам структурировать свои мысли и, как результат, программировать будет проще.

Код примеров в этой книге не содержит так много комментариев, как мы обычно используем, потому что они уже содержатся в пояснениях.

Глава 3: Функции

Самый длинный кусок кода, который мы видели до сих пор, включал всего 5 операторов. Программы из многих тысяч строк могут казаться излишне многословными, поэтому мы должны обсудить принципы и алгоритмы работы Objective-C на самых ранних стадиях.

Если бы программа состояла из длинной, непрерывной последовательности утверждений, то было бы трудно найти и исправить ошибки.

Кроме того, некоторые последовательности операторов могут появляться в вашей программе в нескольких местах. Если есть ошибка в последовательности операторов, то ее нужно исправить в нескольких местах. Ужас: очень легко забыть одну (или две)! Итак, люди придумали способ писать код, который упрощает исправление ошибок.

Решение этой проблемы заключается в том, что последовательность операторов составляет функцию. Например, вы можете написать функцию, которая позволяет рассчитать площадь круга. После того как вы проверили, что этот код работает надежно, вам никогда не придется проверять, есть ли в нем ошибки. Набор операторов, называемый функцией, имеет название, и вы можете вызывать его для выполнения кода. Это понятие использования функций настолько глобально, что всегда есть по крайней мере одна функция в программе: `main()`. Благодаря ей компилятор знает, где должно начаться выполнение кода после запуска.

Функция main()

Давайте взглянем на функцию `main()` подробнее [19].

```
//[19]
main()
{
  // Тело функции main(). Добавьте свой код сюда.
}
```

Инструкция [19.2] показывает название функции, т.е. `main`, завершенное открытием и закрытием скобок. В то время как `main` является зарезервированным словом, и присутствие функции `main()` является обязательным, если вы напишете свои собственные функции, то можете назвать их так, как вам нравится. Постановка там скобок — хорошая вещь, но мы не будем обсуждать это до конца главы. На строках [19.3, 19.5] есть фигурные скобки. Мы должны писать наш код между этими фигурными скобками {}. Все, что находится между ними, называется телом функции. Я взял код из первой главы и вставил его здесь [20].

```
//[20]
main()
{
  // Ниже объявляются переменные
  float pictureWidth, pictureHeight, pictureSurfaceArea;
  // Мы инициализируем переменные (устанавливаем их значения)
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  // Здесь производится расчет
  pictureSurfaceArea = pictureWidth * pictureHeight;
}
```

Наша первая функция

Если и дальше добавлять код в тело функции `main()`, то в итоге программу будет сложно отлаживать. Хотелось бы избежать неструктурированного кода. Давайте напишем другую программу с иной структурой. Помимо обязательной функции main() мы создадим функцию `circleArea()` [21].

```
//[21]
main()
{
  float pictureWidth, pictureHeight, pictureSurfaceArea;
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  pictureSurfaceArea = pictureWidth * pictureHeight;
}

circleArea() // [21.10]
{

}
```

Это просто, но наша пользовательская функция [21.10] пока ничего не делает. Заметим, что описание функции находится вне тела функции `main()`. Иначе — функции не вложенные.

Наша новая функция `circleArea()` должна вызываться из функции `main()`. Посмотрим, как это сделано [22].

```
//[22]
main()
{
  float pictureWidth, pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea; // [22.5]
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  circleRadius = 5.0; // [22.8]
  pictureSurfaceArea = pictureWidth * pictureHeight;
  // Здесь мы вызываем нашу функцию!
  circleSurfaceArea = circleArea(circleRadius); // [22.11]
}
```

Примечание: остальная часть программы не показана (см. [21]).

Передача аргументов

Мы добавили пару вещественных переменных [22.5] и инициализировали переменную `circleRadius`, т.е. установили ее значение [22.8]. Наиболее интересна строка [22.11], где вызывается функция `circleArea()`. Вы можете видеть, что название переменной `circleRadius` было помещено между круглыми скобками. Это аргумент функции `circleArea()`. Значение переменной `circleRadius` будет передано функции `circleArea()`. Когда функция `circleArea()` закончит вычисления, она должна вернуть результат. Давайте изменим функцию `circleArea()` [21], чтобы видеть это [23].

Примечание: показана только функция `circleArea()`.

```
//[23]
circleArea(float theRadius) // [23.2]
{
  float theArea;
  theArea = 3.14159 * theRadius * theRadius; // S = πR2 [23.5]
  return theArea;
}
```

В [23.2] мы определяем, что для функции `circleArea()` требуется входное значение вещественного типа. После получения оно хранится в переменной с именем `theRadius`. Мы используем вторую переменную `theArea` для хранения результатов расчетов [23.5], поэтому мы должны объявить ее [23.4] также, как объявили переменные в функции `main()` [22.4]. Следует заметить, что объявление переменной `theRadius` делается в скобках [23.2]. Строка [23.6] возвращает результат в программу, из которого функция была вызвана. В результате в строке [22.11] переменная `circleSurfaceArea` установлена в это значение.

Функция в примере [23] завершена за одним исключением. Мы не указали тип результата, который возвращается функцией. Компилятор требует от нас сделать это, поэтому у нас нет иного выбора, кроме как подчиниться и указать ему вещественный тип [24.2].

```
//[24]
float circleArea(float theRadius) //[24.2]
{
  float theArea;
  theArea = 3.14159 * theRadius * theRadius;
  return theArea;
}
```

Первое слово в строке [24.2] показывает, что данные, возвращаемые этой функцией (т.е. значение переменной `theArea`), — вещественного типа. Как программист, вы должны сделать переменную `circleSurfaceArea` в функции `main()` [22.5] такого же типа, чтобы компилятор не пилил нас за это.

Не все функции требуют параметров. Если их нет, скобки `()` требуются, но остаются пустыми.

```
//[25]
int throwDice()
{
  int noOfEyes;
  // Код для генерации случайного значения от 1 до 6
  return noOfEyes;
}
```

Возвращаемые значения

Не все функции возвращают значение. Если функция ничего не возвращает, она имеет тип `void`. Оператор `return` является опциональным. Если вы используете тип `void`, то ключевое слово `return` не должно сопровождаться значением/названием переменной.

```
//[26]
void beepXTimes(int x)
{
  // Код для мычания x раз
  return;
}
```

Если функция принимает более одного аргумента как, например, функция `pictureSurfaceArea()` ниже, эти аргументы разделяются запятыми.

```
//[27]
float pictureSurfaceArea(float theWidth, float theHeight)
{
  // Код для расчета площади поверхности
}
```

Функция `main()`, в соответствии с соглашением, должна возвратить целое число, и так и есть — у нее тоже есть оператор `return`. Он должен возвратить `0` [28.9], чтобы показать, что функция выполнилась без проблем. Т.к. функция `main()` возвращает целое число, мы должны написать `int` перед `main()` [28.1]. Теперь давайте напишем весь код, который мы имеем в одном списке.

```
//[28]
int main()
{
  float pictureWidth, pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea;
  pictureWidth = 8;
  pictureHeight = 4.5;
  circleRadius = 5.0;
  pictureSurfaceArea = pictureWidth * pictureHeight;
  circleSurfaceArea = circleArea(circleRadius); // [28.10]
  return 0; // [28.11]
}

float circleArea(float theRadius) // [28.14]
{
  float theArea;
  theArea = 3.14159 * theRadius * theRadius;
  return theArea;
}
```

Запускаем все это

Как вы можете видеть [28], у нас есть функция `main()` [28.2] и другая функция, описанная нами [28.14]. Если мы попытаемся скомпилировать этот код, компилятор продолжит ругаться. В строке [28.10] он не видит, что существует фунция `circleArea()`. Почему? Видимо, компилятор начинает чтение функции `main()` и вдруг сталкивается с тем, чего не знает. Он останавливается и дает вам предупреждение. Чтобы ублажить компилятор, нужно просто добавить описание этой функции перед функцией `main()` [29.2]. Ничего сложного в этом нет, поскольку оно такое, как и в строке [28.14], только заканчивается точкой с запятой. Теперь компилятор не будет удивлен, когда столкнется с вызовом этой функции.

```
//[29]
float circleArea(float theRadius); // объявление функции

int main()
{
  // Код фунции main здесь…
}
```

Примечание: остальная часть программы не показана (см. [28]).

Скоро мы сможем скомпилировать эту программу. Сначала несколько ненужных деталей.

Во время написания программ рекомендуется иметь в виду возможность повторного использования кода в будущем. Наша программа может иметь функцию `rectangleArea()`, как показано ниже [30], и функция может быть вызвана в функции `main()`. Это полезно даже в том случае, если код функции используется лишь один раз. Функцию `main()` теперь легче читать. Если понадобится отладить код, вам будет легче найти ошибку в вашей программе. Вы можете обнаружить, что она в функции.

Вместо того, чтобы долго искать в коде, вам надо просто проверить вычисления в функции, которые легко найти, благодаря открытию и закрытию фигурных скобок.

```
//[30]
float rectangleArea(float length, float width)
{
  return (length * width);
}
```

Как вы видите, в простых случаях, вроде этого, можно совместить вычисления и возврат результата в одном операторе [30.3]. На самом деле, лишняя переменная `theArea` *в [28.16] была нужна только как пример локального определения переменной в функции.*

Хотя функции, которые мы создали в этой главе, достаточно тривиальные, важно понимать главное — теперь вы можете свободно изменять тело функции не влияя на остальной код, который эту функцию вызывает. Конечно, это все верно только пока вы не измените заголовок функции (т.е. ее имя, список параметров и возвращаемое значение, указанные в первой строчке).

Например, вы можете изменить названия переменных внутри функции, и функция будет работать как и раньше (и на остальной код это не повлияет). Кто-то другой тоже может написать функцию, и вы сможете её использовать не зная, что происходит у неё внутри. вам надо только знать как эту функцию использовать, а для этого понадобятся:

- Название функции;
- Число, порядок и типы аргументов функции;
- Что функция возвращает (значение площади поверхности прямоугольника), и тип возвращаемого значения.

В примере [30], эти значения соответственно будут:

- `rectangleArea`;
- 2 аргумента, оба вещественного типа, где первый представляет длину, а второй ширину;
- Функция что-то возвращает и тип возвращаемого значения `float` (это видно по первому слову строки [30.2]).

Защищенные переменные

Код внутри функции защищен от воздействия извне (от основной программы и кода других функций)

Что это означает, что значение переменной внутри функции по умолчанию не влияет на любую другую переменную в любой другой функции, даже если он имеет такое же название. Это является наиболее важной чертой Objective-C. В главе 5, мы будем обсуждать данную проблему еще раз. Но сперва, мы разберемся с Xcode, запустим код из примера [29].

Глава 4: Вывод на экран

Мы добились хорошего прогресса в нашей программе, но мы еще не знаем как показать результаты вычислений. Язык Objective-C сам не знает как это сделать, но к счастью люди написали функции вывода на экран, которые мы и будем использовать. Есть много методов вывода на экран. В этой книге мы будем использовать функцию окружения Apple Cocoa — функцию `NSLog()`. Это хорошо, потому что теперь вам не прийдётся беспокоиться о выводе чего-либо на экран.

> *Функция* `NSLog()` *изначально была разработана для отображения ошибок, а не для вывода результатов работы программы. Но из-за простоты использования, мы адаптировали ее в этой книге для отображения результатов. Когда вы немного изучите Cocoa, вы сможете использовать более изощренные методы.*

Использование NSLog

Давайте взглянем на то, как используется функция `NSLog()`.

```
//[31]
int main()
{
  NSLog(@"Юлия – моя любимая актриса.");
  return 0;
}
```

При исполнении инструкций из примера [31] текст: «Юлия — моя любимая актриса.» будет выведен на экран.

Текст между `@"` и `"` называется строкой.

В дополнение к самой строке, функция `NSLog ()` выводит различную дополнительную информацию, текущую дату и имя приложения. Например на моей системе программа [31] выводит:

```
2005-12-22 17:39:23.084 test[399] Юлия - моя любимая актрисса.
```

Строка может иметь длину ноль и более символов.

Примечание: В следующих примерах показаны только те строки, которые представляют для нас интерес.

```
//[32]
NSLog(@"");
NSLog(@" ");
```

Выражение [32.2] содержит `0` символов, это называется пустой строкой (например, строка, длина которой равна `0`). Выражение [32.3] не является пустой строкой, несмотря на то, что она выглядит очень похоже. Она содержит один пробел, а значит ее длина равна 1.

Некоторые специальные последовательности символов в строках имеют особый смысл. Такие символы называются эскейп-символами.

Например, для того, чтобы последнее слово в нашем предложении напечаталось с начала новой строки, необходимо включить в выражение [33.2] специальный код. Таким кодом является \n, называемый символом новой строки.

```
//[33]
NSLog(@"Юлия – моя любимая \nактриса.");
```

Теперь вывод строки на экран будет выглядеть следующим образом (только относительный вывод будет показан):

```
Юлия – моя любимая
актриса.
```

Знак \ (обратный слеш) в [33.2] называется *эскейп-символом*, так как он служит сигналом для функции NSLog() о том, что следующий символ не обычная буква, а символ, имеющий особое значение: в нашем примере n означает «начать новую строку».

В редком случае, когда вы хотите напечатать обратный слеш \, может показаться, что у вас есть проблема. Если у обратного слеша есть специальное значение, то как можно его напечатать? Хорошо, мы помещаем еще один обратный слеш перед или после первого. Это сообщает функции NSLog(), *что второй обратный слеш должен быть напечатан и что любое специальное значение должно быть проигнорировано. Например:*

```
//[34]
NSLog(@"Юлия – моя любимая актриса.\\n");
```

Инструкция [34.2] выдаст

```
Юлия – моя любимая актриса.\n
```

Печать значений переменных

Пока мы показывали только статические строки. Давайте напечатаем вычисленное значение на экран.

```
//[35]
int x, integerToDisplay;
x = 1;
integerToDisplay = 5 + x;
NSLog(@"Целое значение – %d.", integerToDisplay);
```

Имейте в виду, что в скобках у нас стоят строка, запятая и имя переменной. Строка содержит нечто странное: %d. Как и обратный слеш \, символ процента % имеет особое значение. Если после него следует d (малое десятичное число), то после выполнения кода вместо %d будет показано то, что стоит после запятой, т.е. текущее значение переменной integerToDisplay. Запуск примера [35] приводит к

```
Целое значение – 6
```

Для печати вещественных значений вам следует использовать %f вместо %d

```
//[36]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"Вещественное значение – %f.", floatToDisplay);
```

Вы сами можете указать, сколько оставить цифр после точки. Чтобы показать 2 значимые цифры,

поставьте .2 между % и f, вот так:

```
//[37]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"Вещественное значение с 2 значимыми цифрами – %.2f.", ↵
floatToDisplay);
```

Позже, когда Вы узнаете как повторять вычисления, Вы можете захотеть создать таблицу. Представьте, например, таблицу для перевода из градусов Фаренгейта в градусы Цельсия. Если вы хотите чтобы таблица выглядела опрятно, значения в столбцах должны быть одинаковой ширины. Вы можете указать нужную ширину (количество символов) непосредственно между % и f (или между % и d)

Однако, если Вы укажете ширину меньше чем реальная ширина числа, ваше указание будет проигнорировано.

```
//[38]
int x = 123456;
NSLog(@"%2d", x);
NSLog(@"%4d", x);
NSLog(@"%6d", x);
NSLog(@"%8d", x);
```

Пример [38] имеет следующий вывод:

```
123456
123456
123456
  123456
```

В первые двух выражениях, [38.3, 38.4] мы запрашиваем слишком мало пространства для количества символов, которое будет отображаться, но выражение все равно выводится целиком. Только выражение, [38,6] задает ширину большую, чем значение, поэтому сейчас мы видим появление дополнительного пространства, свидетельствует о размере места для выводимого числа.

Кроме того, можно объединить параметры ширины и количества десятичных чисел, который должны отображаться.

```
//[39]
float x=1234.5678;
NSLog(@"Зарезервировать 10 символов, и отобразить 2 значащих цифры.");
NSLog(@"%10.2d", x);
```

Отображение нескольких значений

Конечно, можно отображать более чем одно значение, или любое сочетание значений [40.4]. Вы должны убедиться, что вы правильно указываете тип данных (`int`, `float`), используя `%d` и `%f` соответственно.

```
//[40]
int x = 8;
float pi = 3.14159;
NSLog(@"Целое значение – %d, вещественное значение – %f.", x, pi);
```

Соответствие символов значениям

Одна из самых распространённых ошибок начинающих программистов это неправильный выбор типов данных для `NSLog()` и других функций. Если результаты ваших программ кажутся вам странными или если программа завершает работу нестандартно, без причины. Проверьте типы данных!

Если у вас получилось в первый раз, это не значит, что во второй получится отобразить нормально.

```
//[40b]
int x = 8;
float pi = 3.1416;
NSLog(@"Целое значение – %f, вещественное значение – %f.", x, pi);
// На самом деле должно быть: NSLog(@"Целое значение – %d, вещественное ↵
значение – %f.", x, pi);
```

Получим следующий выход:

```
Целое значение – 0.000000, вещественное значение – 0.000000.
```

Использование фреймворка Foundation

От первого запуска программы нас отделяет только один вопрос.

Как наша программа узнает об этой полезной функции `NSLog()`? Ну, она и не узнает, пока мы ей не скажем. Чтобы это сделать, наша программа должна сообщить компилятору, чтобы тот импортировал библиотеку «сладостей», включая и функцию `NSLog()`. Используйте следующую конструкцию:

```
#import <Foundation/Foundation.h>
```

Эта команда должна быть первой в вашей программе. Собрав вместе все, чему мы научились в этой главе, мы получим код, который мы запустим в следующей главе:

```
//[41]
#import <foundation/foundation.h>

float circleArea(float theRadius);
float rectangleArea(float width, float height);

int main()
{
  float pictureWidth, pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea;
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  circleRadius = 5.0;
  pictureSurfaceArea = rectangleArea(pictureWidth, pictureHeight);
  circleSurfaceArea = circleArea(circleRadius);
  NSLog(@"Площадь окружности: %10.2f.", circleSurfaceArea);
  NSLog(@"Площадь картинки: %f. ", pictureSurfaceArea);
  return 0;
}

float circleArea(float theRadius) // первая пользовательская функция
{
  float theArea;
  theArea = 3.14159 * theRadius * theRadius;
  return theArea;
}

float rectangleArea(float width, float height)
{
  return width*height;
}
```

Глава 5: Компиляция и Запуск программ

Полученный нами код, пока что, не более чем читаемый текст. И хотя для нас это - не литературное произведение, Mac'у он ещё более непонятен. С ним он не сможет сделать ровным счётом ничего! Для преобразования вашего программного кода в исполняемый Mac'ом необходима специальная программа, называемая компилятором. Она является частью программной среды Xcode, которую вам нужно установить с диска, идущего совместно с копией Mac OS X. В любом случае, проверьте, что у вас установлена последняя версия, которую можно скачать в разделе для разработчиков http://developer.apple.com (необходима бесплатная регистрация).

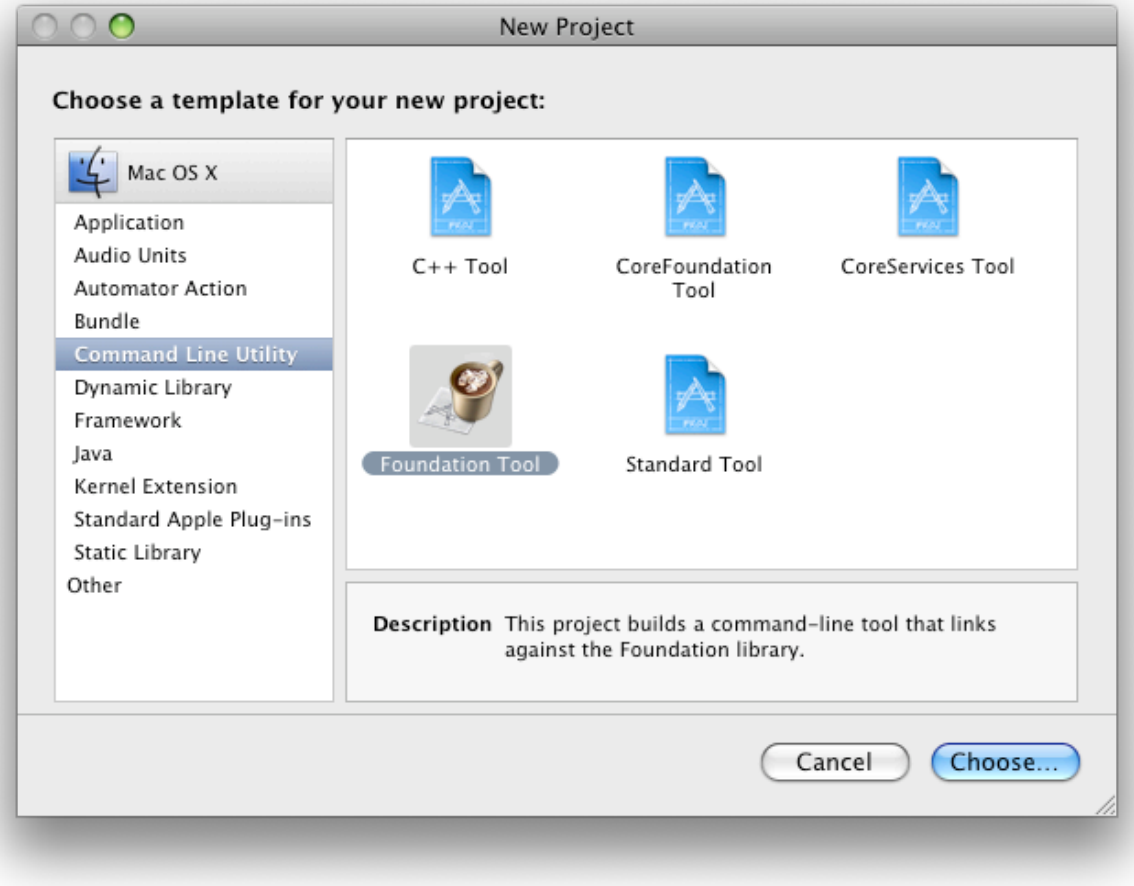
Создание проекта

Запустим, теперь, Xcode, который располагается в папке /Development/Applecation/ на вашем диске или же просто введите в Spotlight «xcode» и нажите клавишу ⏎ на клавиатуре. При первом запуске будет задано несколько вопросов. На все из них можно ответить утвердительно и в случае чего никогда не поздно изменить их в настройках. Для того чтобы начать работу, нужно создать новый проект (File ▸ New Project… ⇧⌘N). При этом появится диалоговое окно с возможными типами проектов.

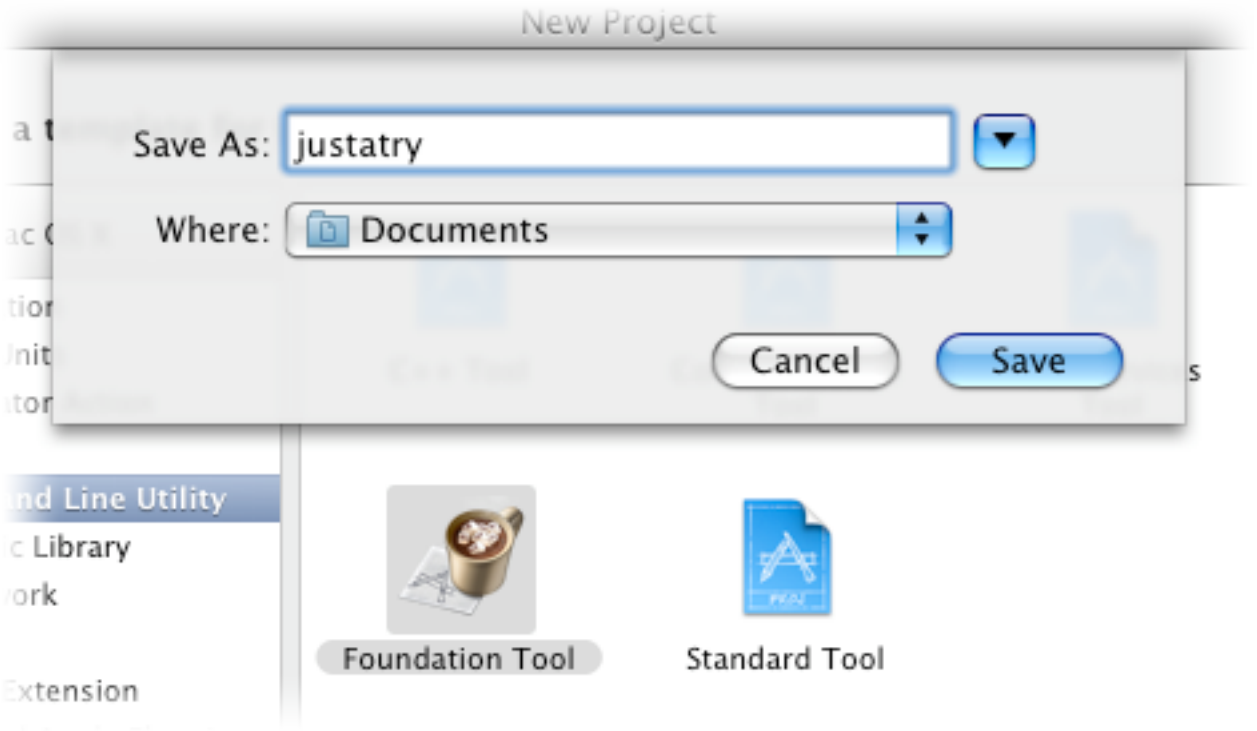
Мы хотим создать очень простую программу на Objective-C, без графического пользовательского интерфейса (GUI). Промотайте вниз и выберите Foundation Tool под секцией Command Line Utility.

Введите имя вашего приложения, например «`justatry`». Выберите место на диске, куда вы сохраните ваш проект и сохраните (нажмите кнопку Save).

Проект, который мы собираемся создать, можно запустить из Терминала (Terminal). Если вы хотите сделать это, и избежать некоторых трудностей, убедитесь, что название вашего проекта состоит из одного слова. Кроме того, обычно названия программ, запускающихся из Терминала, начинаются с маленькой буквы. С другой стороны, название программ с графическим интерфейсом пользователя должно начинаться с заглавной.



Мастер Xcode позволяет создавать новые проекты.



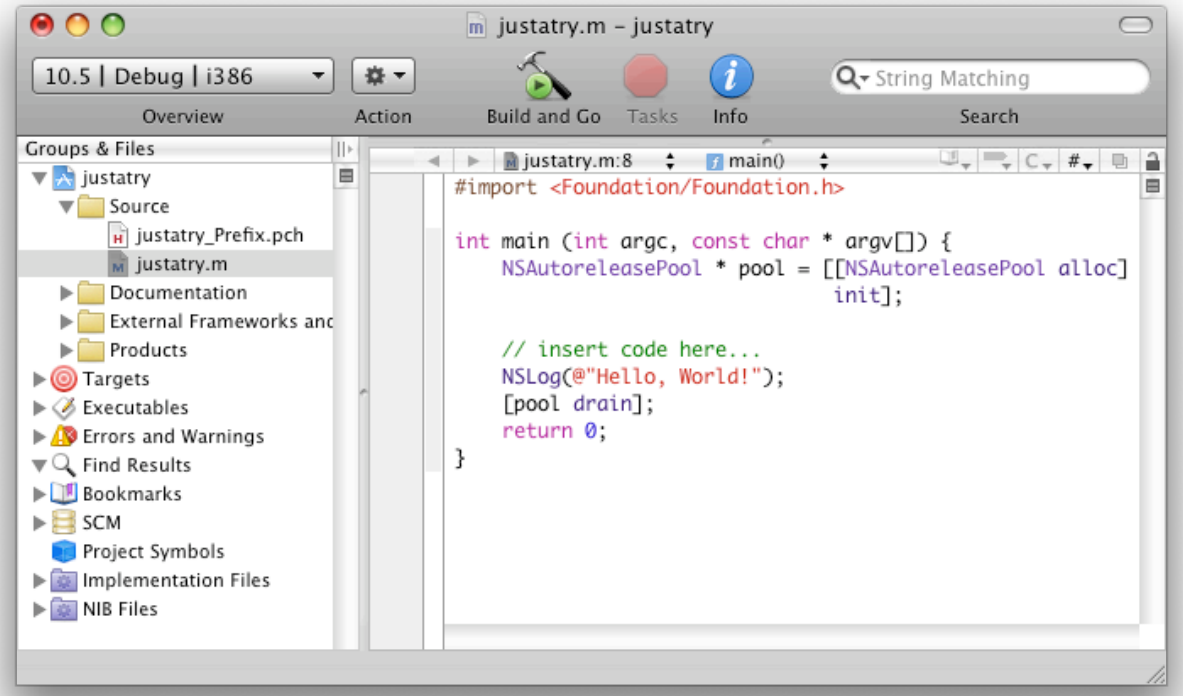
Выбор имени и месторасположения нового проекта.

Обзор Xcode

Теперь перед вами окно, которое вы, как программист, будете видеть очень часто. Окно состоит из двух фреймов. Основной фрейм располагается справа и в основном в нём происходит работа с исходными кодами. «Groups & Files» — располагается слева и служит для доступа ко всем файлам, из которых состоит ваша программа. Сейчас их не слишком много, но позже, когда вы будете работать над многоязычными графическими программами, все файлы для описания интерфейса и различных языков будут именно тут. Файлы сгруппированы и хранятся в папках, но не стоит их искать на вашем жестком диске.

Xcode предоставляет эти виртуальные папки (Groups) для организации вашего проекта.

В поле ввода «Groups & Files», попробуйте открыть группу jystatray, для того чтобы начать работать с исходным кодом в файле с именем justatry.m [42]. Вы помните, что каждая программа должна содержать функцию с именем `main()`? Ну вот, этот файл, содержит эту `main()` функцию. Позже в этой главе мы собираемся изменить его, чтобы включить его в код нашей программы. Если вы открываете justatry.m, дважды щелкнув его значок, вас ждет приятный сюрприз. Apple уже создала `main()` функцию для вас.



Xcode, отображающий main() функцию.

```
//[42]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) // [42.3]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init]; // [42.5]

  // insert code here…
  NSLog(@"Hello, World!");
  [pool drain]; //[42.9]
  return 0;
}
```

Давайте взглянем на эту программу и поищем то, что вы уже знаете:

- Директива `import`, начинающаяся символом `#`, нужна для использования таких функций как `NSLog`;
- Функция `main()`;
- Фигурные скобки, которые должны содержать тело нашей программы;
- Комментарий, приглашающий писать код здесь;
- Команда `NSLog()` для вывода строки на экран;
- Команда `return 0`.

Есть здесь и то, что вы пока не знаете:

- Забавный список аргументов в скобках функции `main()` [42.3];
- Оператор, начинающийся на `NSAutoreleasePool` [42.5];
- Еще один оператор со словами `pool` и `drain` [42.9].

Лично я не очень-то счастлив, когда авторы книг предлагают мне, как читателю, ознакомиться с кодом, полным непонятных операторов, обещая при этом все разъяснить позже. Ага, как же. Имено поэтому я слегка отклонился от темы чтобы познакомить вас с концепцией «функций», так что вам не прийдётся сталкиваться с большим количеством новых понятий.

Вы уже знаете, что функции это способ организовать программу, что в каждой программе есть функция main(), и как функции выглядят. Тем не менее, должен признаться, я не могу сейчас полностью объяснить все, что вы видите в примере [42]. Я очень сожалею, но пока просто не обращайте внимания на эти строки ([42.3, 42.5 и 42.9]).

Есть еще несколько моментов в Objective-C, с которыми стоит познакомится в самом начале и которые позволят создавать простые программы. Две последних главы были довольно сложные, но в следующих трёх главах все будет просто (нам нужно экономить силы для штурма следующей вершины).

Если вы действительно не хотите быть оставленными без каких-либо объяснений, вот вам резюме:

- Аргументы в `main()` [42.3] функции необходимы для запуска программ из Терминала. При помощи этих аргументов передаются параметры в программу (прим. редактора);
- Ваша программа резервирует память [42.5]. Память, которая нужна другим программам, когда ваша программа закончила работу. Ваша задача резервировать память, которая нужна вашей программе;
- И не менее важно: освобождение памяти, когда она не нужна [42.9].

Build and Go

Давайте запустим программу [42]. Нажмите вторую иконку «с молотком» помеченную как «Build and Go» для постройки (сборки) и запуска приложения.

Кнопка Build and Go

Программа выполняется и результаты выводятся в окно «Console» (сделать видимым его можно, находясь в Xcode, выбрав пункты меню Run ▸ Console ⇧⌘R), вместе с дополнительной информацией. Последняя строчка сообщает нам, что программа завершилась с кодом 0. Этот как раз тот 0, который вернула функция `main()`, которую мы обсуждали в Главе 3 [28.11]. Значит наша программа выполнилась до последней строчки и нигде не прервалась раньше времени. Уже хорошо!

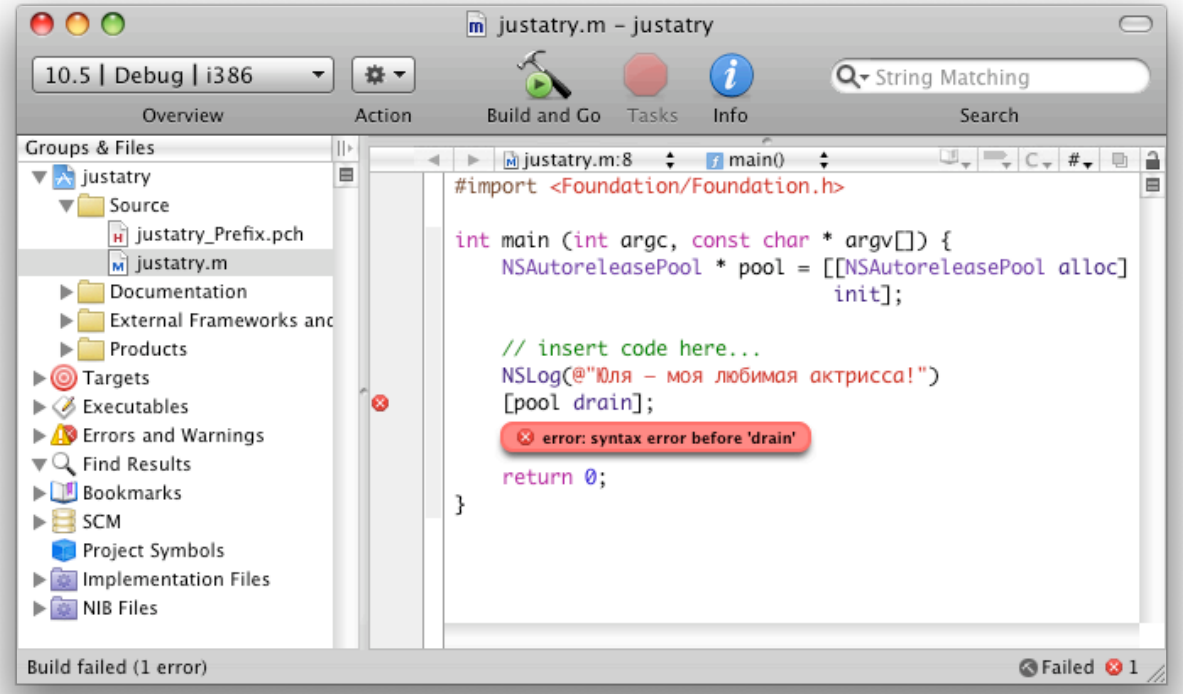
Баггинг

Вернемся к примеру [42] и посмотрим, что произойдет, если имеется ошибка в программе. Например, я поменял функцию `NSLog()`, но забыл поставить точку с запятой в конце команды.

```
//[43]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  // insert code here...
  NSLog(@"Юля – моя любимая актрисса") // Упс, забыл поставить точку с ↵
запятой!
  [pool drain]; //[43.9]
  return 0;
}
```

Чтобы построить приложение, нажмите иконку «Build and Go» на панели. Перед командой [43.9] появится красный круг.



Xcode указывает на ошибку компиляции.

Если вы кликните его, строчка под тулбаром покажет короткое пояснение ошибки:

```
error: syntax error before "drain"
```

Синтаксический анализ, или парсинг — один из первых этапов работы компилятора: он проходит по каждой строчке кода, чтобы проверить может ли он её понять. И вы должны помочь ему в этом, размещая подсказки. Например, для директивы `import` надо использовать символ #, а конец оператора надо отмечать с помощью точки с запятой (;). В нашем случае, когда компилятор доходит до строчки [43.9], он замечает, что что-то не так. Однако, он не понимает, что проблема не в этой строке, а в предыдущей, где не хватает точки с запятой (компилятор считает данную запись из двух операторов одним оператором, которую не может разобрать — прим. редактора). Важный вывод для нас состоит в том, что хотя

компилятор пытается давать разумные пояснения, не стоит рассчитывать, что его сообщения обязательно укажут точную причину и место ошибки (хотя скорее всего он будет очень близок к этому).

Исправьте код добавив точку с запятой, и запустите программу еще раз, чтобы убедиться что теперь все в порядке.

Наше первое приложение

А сейчас давайте возьмем код из предыдущей главы и объединим его с кодом, который приготовил для нас Apple ([42]). В результате получим пример [44].

```
//[44]
#import <Foundation/Foundation.h>

float circleArea(float theRadius); // [44.4]
float rectangleArea(float width, float height); // [44.5]

int main (int argc, const char * argv[]) // [44.7]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  int pictureWidth;
  float pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea;
  pictureWidth = 8;
  pictureHeight = 4.5;
  circleRadius = 5.0;
  pictureSurfaceArea = pictureWidth * pictureHeight;
  circleSurfaceArea = circleArea(circleRadius);
  NSLog(@"Площадь картинки: %f. Площадь окружности: %10.2f.",
    pictureSurfaceArea, circleSurfaceArea);
  [pool drain];
  return 0;
}

float circleArea(float theRadius) // [44.24]
{
  float theArea;
  theArea = 3.14159 * theRadius * theRadius;
  return theArea;
}

float rectangleArea(float width, float height) // [44.31]
{
  return width *height;
}
```

Убедитесь, что вы понимаете структуру программы. У нас имеются заголовки [44.4, 44.5] созданных нами функций `circleArea()` [44.24] и `rectangleArea()` [44.31], находящиеся выше функции `main()` [44.7], где они и должны быть. Наши функции находятся вне

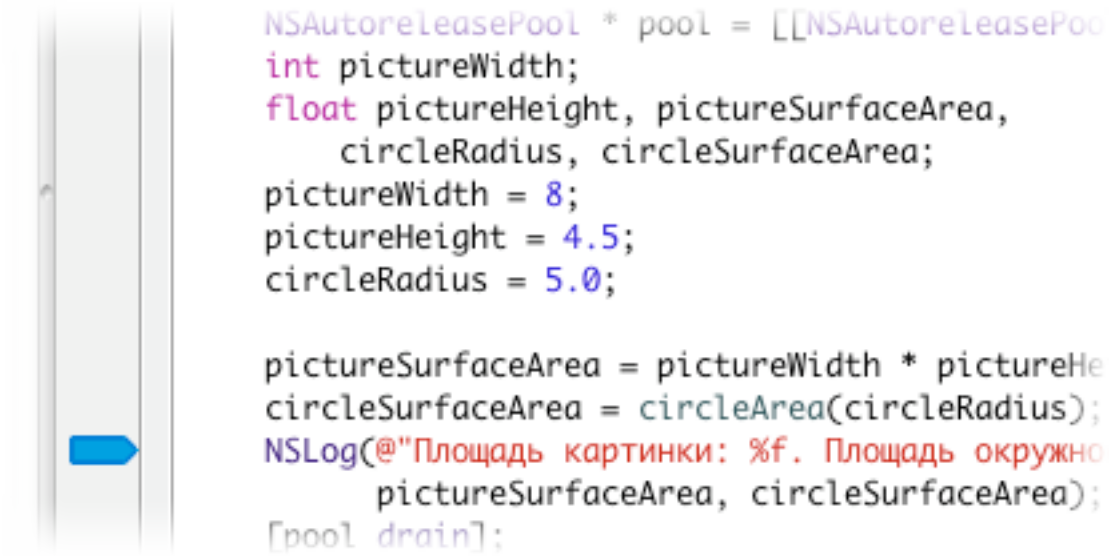
фигурных скобок функции `main()` [44.8, 44,22]. Мы набираем текст программы внутри тела функции `main()`, где это и советует делать Apple.

После исполнения кода, мы увидим:

```
Площадь картинки: 36.000000. Площадь окружности:      78.54.
The Debugger has exited with status 0.
```

Отладка

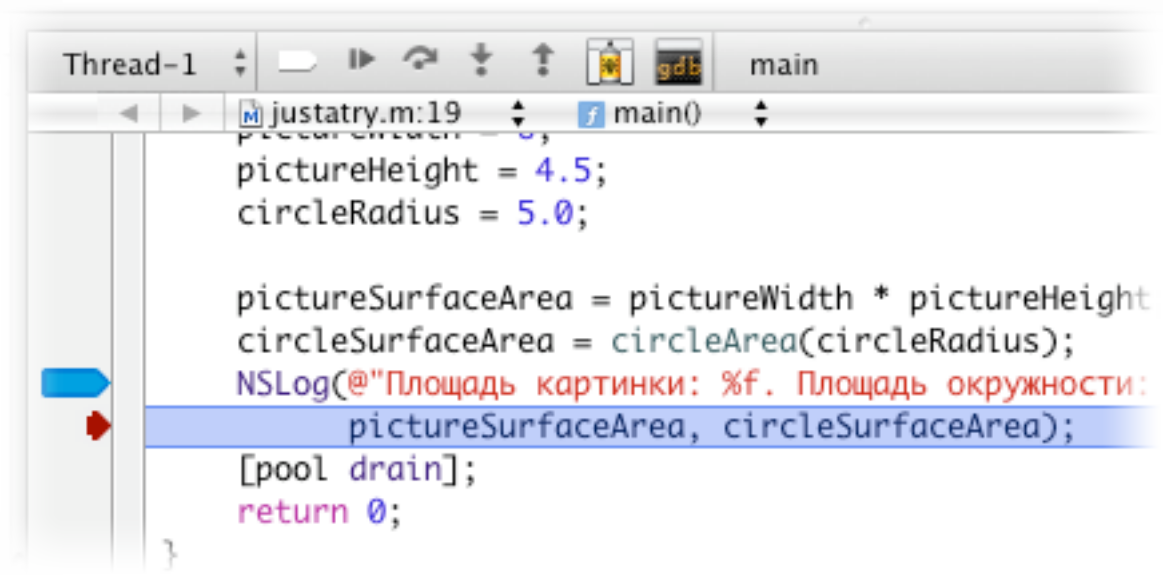
Когда программа усложняется, отладка выполняется труднее. Поэтому периодически возникает необходимость узнать, что происходит внутри программы в момент её выполнения. С этой задачей также с успехом справляется `Xcode`, необходимо лишь нажать левой кнопкой мыши на серой границе перед выражениями, для которых вы хотите узнать значения переменных и `Xcode` добавить в этом месте точку остановки — breakpoint, отмеченную голубой стрелкой.



Установка точки остановки в коде

Запомните, вы увидите значения переменных до выполнения строки с точкой остановки, так что ставьте точку остановки в следующей строке после изменения нужного значения.

Нажав «Build and Go (Debug)» (`Run ▸ Debug ⌥⌘Y`), перед вами будет оно `Xcode` в режиме отладки.

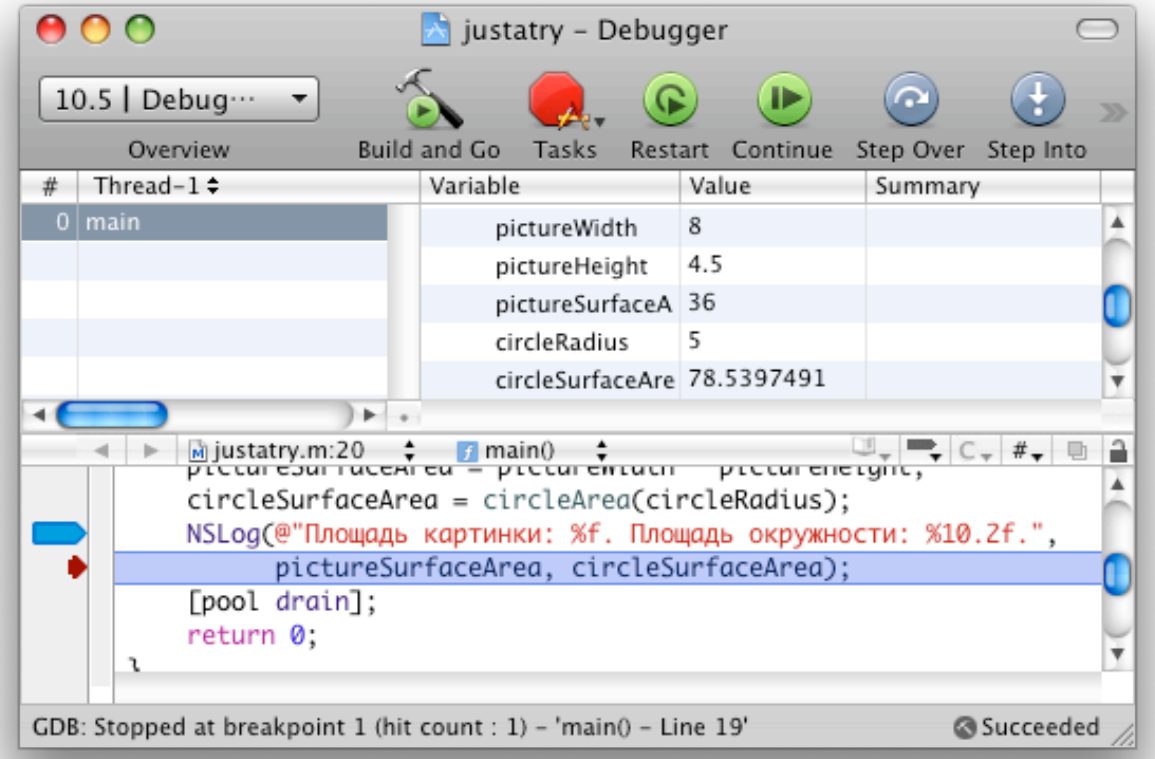


Xcode в режиме отладки позволяет выполнять программу шаг за шагом и проверять переменные.

Программа будет выполняться до того момента как достигнет первой точки остановки. Если выбрать верхнюю правую панель в отладчике (Debugger. Для открытия отладчика Run ▸ Debugger ⇧⌘Y), то вы сможете увидеть значения различных переменных. Любые значения, которые изменились с момента отображения последней точки остановки выделены красным цветом. Для продолжения выполнения нажмите кнопку «Continue». Отладчик — мощный инструмент, который поможет разобраться со многими ошибками, возникающими при написании программ. Поиграйтесь с ним некоторое время, чтобы поплотнее с ним ознакомиться.

Теперь у нас есть все, чтобы писать, отлаживать и запускать простые программы для Mac OS X.

Если вы не хотите делать программы с графическим пользовательским интерфейсом (GUI), единственное что вам нужно сейчас — увеличить ваши знания Objective-C, что позволит разрабатывать более сложные программы. В течение следующих нескольких глав мы будем делать именно это. После чего мы будем погружаться приложения с GUI.



Окно отладчика в процессе работы с программой.

Глава 6: Условные операторы

Оператор условного перехода if()

Для того, чтобы ваш код выполнял последовательность действий только если выполнено конкретное условие, существуют специальные ключевые слова [45.3].

```
//[45]
// age это целочисленная переменная, хранящая возраст пользователя
if (age > 30) // Символ > означает "больше чем"
{
  NSLog(@"Старше 30 лет."); //[45.5]
}
NSLog(@"Завершено."); //[45.7]
```

Строка [45.3] демонстрирует инструкцию `if()`, также известную, как условную инструкцию. Вы наверняка обратили внимание на фигурные скобки, которые содержат весь код, который вы хотите выполнить при условии, что логические выражения в скобках являются верными. Так, строка [45.5] будет распечатана, если условие `age > 30` выполнимо. Однако строка [45.7] будет распечатана независимо от того, выполнимо условие или нет, потому как находится не внутри фигурных скобок.

Оператор условного перехода if() (полная конструкция)

Мы также можем предоставить альтернативный набор инструкций, если условие не выполняется, используя команды `if` и `else` [46].

```
//[46]
// age это целочисленная переменная, хранящая возраст пользователя
if (age > 30) //[46.3]
{
  NSLog(@"Старше 30 лет."); //[46.5]
}
else
{
  NSLog(@"Моложе 30 лет."); //[46.9]
}
NSLog(@"Завершено.");
```

Строка кода [46.9] может быть выведена только в том случае, если условие [46.3] не выполняется.

Сравнения

Помимо тех операторов, которые мы привели в примерах, возможно применение следующих операторов сравнения числовых значений:

```
== равенство
>  больше чем
<  меньше чем
>= больше или равно
<= меньше или равно
!= не равно
```

Учтите, что оператор равенства это двойной знак равенства. Это легко забыть и пользоваться одним знаком равенства по привычке. Но одиночный знак равенства это оператор присваивания, который выполнит присваивание следующего по коду выражения, что является причиной проблем и кода, содержащего ошибки, у новичков. Теперь громогласно заявите: «Я никогда не забуду использовать двойной знак равенства для проверки на равенство!»

Операторы сравнения полезны когда вы хотите повторить некий блок кода некоторое количество раз. Это будет темой следующей главы. Сначала же мы обсудим некоторые аспекты использования управляющей конструкции условного оператора перехода, которые могут нам пригодиться.

Давайте внимательнее взглянем на то, как происходит сравнение. Операция сравнения возвращает только одно из двух значений — истину или ложь.

В Objective-C истина и ложь представляются `1` *или* `0` *соответственно. Даже есть специальный тип данных* `BOOL`*, который вы можете использовать для хранения таких значений. Чтобы обозначить значение «истина» можно использовать* `1` *или* `YES`*. Чтобы обозначить значение «ложь» можно использовать* `0` *или* `NO`*.*

```
//[47]
int x = 3;
BOOL y;
y = (x == 4); // y примет значение 0.
```

Можно проверять и больше условий. Если должны выполняться два и более условия, используется логический оператор И, представляемый двумя сиволами амперсанда &&. Если должно выполняться хотя бы одно условие, используйте логический оператор ИЛИ, представляемый двумя символами вертикальной черты ||.

```
//[48]
if ( (age >= 18) && (age < 65) )
{
  NSLog(@"Скорее всего приходиться работать.");
}
```

Кроме того, можно объединять условные выражения. Нужно всего лишь заключить одно выражение в фигурные скобки другого. Сначала будет проверено условие снаружи, а затем, если оно верно, следующее выражение внутри, и так далее:

```
//[49]
if (age >= 18)
{
  if (age < 65)
  {
    NSLog(@"Скорее всего приходиться работать.");
  }
}
```

Глава 7: Повторение операторов в цикле

Во всех случаях, которые мы рассмотрели выше, каждое выражение было выполнено один раз. Мы всегда можем повторять код в функциях, вызывая их неоднократно [50].

```
//[50]
NSLog(@"Юля – моя любимая актриса.");
NSLog(@"Юля – моя любимая актриса.");
NSLog(@"Юля – моя любимая актриса.");
```

Но даже так необходимо повторение вызова. Иногда вам нужно выполнить одно или несколько выражение некоторое количество раз. Как и во всех языках программирования, в Objective-C есть несколько путей решения этой проблемы.

for()

Если вы знаете количество повторов для выражения (или группы выражений), вы можете уточнить его, включив это количество в выражение примера [51]. Число повторов должно быть натуральным числом, так как вы не можете повторить операцию, скажем, 2.7 раз.

```
//[51]
int x;
for (x = 1; x <= 10; x++)
{
  NSLog(@"Юля - моя любимая актрисса."); //[51.5]
}
NSLog(@"Значение переменной x равно %d", x);
```

В этом примере строка [51.5] будет печатна 10 раз. Во-первых, x присваивается значение 1. Затем компьютер анализирует состояние выражения `x <= 10`. Это условие считается выполненным (x равен 1), поэтому выражения между фигурными скобками выполняются. Тогда, значение x увеличивается, на единицу выражением

x++. В результате значение x, становится равным 2, и снова сравнивается с 10. Поскольку x все еще меньше и не равно 10, выражения между фигурными скобками, выполняются еще раз. Как только x становится равно 11, условие `x <= 10`, больше не выполняется. Последнее выражение [51.7] было включено, чтобы доказать вам, что

`x = 11`, а не 10, после завершения цикла.

Когда-нибудь, вам нужно будет делать большие «шаги» чем просто x++. Все что нужно — заменить выражение на то, которое вам нужно. Следующий пример [52] переводит градусы по Фаренгйту в градусы по Цельсию.

```
//[52]
float celsius, tempInFahrenheit;
for (tempInFahrenheit = 0; tempInFahrenheit <= 200; ↵
  tempInFahrenheit = tempInFahrenheit +20)
{
  celsius = (tempInFahrenheit - 32.0) * 5.0 / 9.0;
  NSLog(@"%6.2f → %6.2f", tempInFahrenheit, celsius);
}
```

Эта программа выведет:

```
   0.00 → -17.78
  20.00 →  -6.67
  40.00 →   4.44
  60.00 →  15.56
  80.00 →  26.67
 100.00 →  37.78
 120.00 →  48.89
 140.00 →  60.00
 160.00 →  71.11
 180.00 →  82.22
 200.00 →  93.33
```

while()

В Objective-C имеются два других способа повторить ряд действий:

```
while () {}
```

и

```
do {} while ()
```

Конструкция `while () {}` по сути идентично циклу `for`, о котором говорилось ранее. Все начинается с проверки условия окончания. Если результат ложный, выражения внутри цикла не выполняются.

```
//[53]
int counter = 1;
while (counter <= 10)
{
  NSLog(@"Юлия - моя любимая актриса.\n");
  counter = counter + 1;
}
NSLog(@"Значение счетчика равно %d", counter);
```

В этом случае, значение счетчика равно 11, если вам это потребуется позже в вашей программе.

В конструкции do {} while (), команды между фигурными скобками будут выполнены как минимум 1 раз.

```
//[54]
int counter = 1;
do
{
  NSLog(@"Юлия - моя любимая актрисса.\n");
  counter = counter + 1;
} while (counter <= 10);
NSLog(@"Значение счетчика равно %d", counter);
```

Конечное значение счетчика ровно 11

Вы получили еще несколько навыков программирования, так что теперь перейдем к решению вопроса посложнее. В следующей части мы напишем первую программу использующую графический интерфейс пользователя (GUI).

Глава 8: Программа с графическим интерфейсом (GUI)

После увеличения наших знаний в Objective-C, мы готовы обсудить как создать программу с графическим интерфейсом (GUI). Я должен признаться кое в чем. Objective-C является фактически расширением языка программирования C. Большинство из того, что мы обсуждали до сих пор является просто C. Так чем же Objective-C отличается от просо C? Именно в «Objective» части. Objective-C связан с абстрактными понятиями, известными как объекты.

До сих пор мы в основном разбирались с числами. Как вы узнали, Objective-C свободно поддерживает концепцию чисел. То есть, она позволит вам создать числа в памяти и манипулировать ими с помощью математических операторов и математических функций. Это отлично, если рассматривать программу работающую с числами (например, калькулятор). Но что если ваша программа, скажем, аудиоплеер, и работает с такими сущностями как: песни, плейлисты, исполнители т.д.? Или что, если ваше приложение управляет воздушным движением и всем, что касается самолетов, полетов, аэропортов т.д.? Правда, было бы неплохо иметь возможность манипулировать такими вещами в Objective-C, также легко как мы работаем с числами?

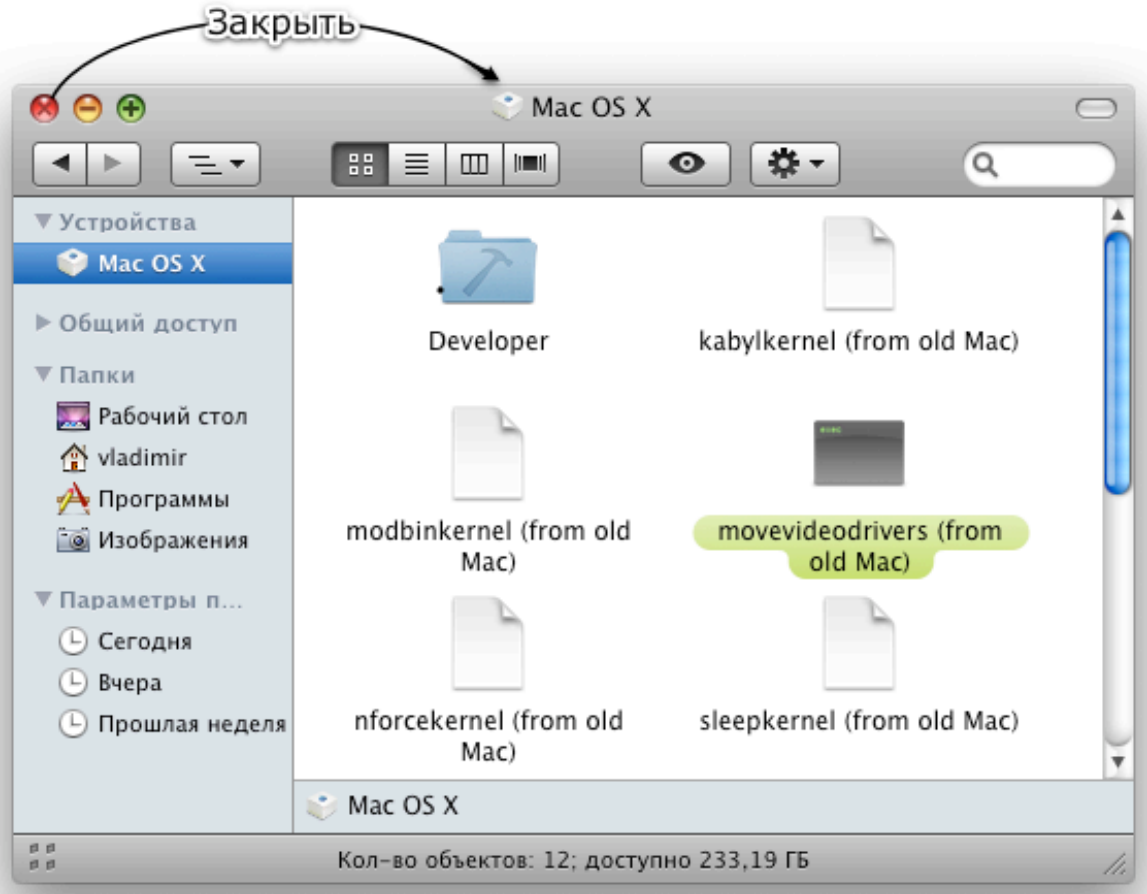
В этом основное преимущество объектов. С Objective-C, вы можете определить тип объектов, с которыми вам нужно работать, а затем написать приложение работающее с этими объектами.

Давайте взглянем на то, как окна обрабатываются в программах написанных на Objective-C, таких как Finder.

Взглянем на окно Finder в вашем Mac. Вверху слева, есть три кнопки. «Красная» — закрыть. Что же произойдет, если вы закроете окно, нажав красную кнопку? Сообщение будет направлено окну. В ответ на это сообщение, окно выполняет определенный код, с тем чтобы закрыть само себя.

Окно — это объект. Вы можете перемещать его по экрану. Эти три кнопки тоже объекты. Вы можете на них нажать. Эти объекты имеют визуальное представление на экране, но это не обязательно для всех объектов. Например, объект который обеспечивает соединение между Safari и выбранным сервером не имеет визуального представления на экране.

Объект (например окно) может содержать другие объекты (например кнопки).



Сообщение «закрыть» было отправлено окну.

Классы

Вы можете иметь столько окон браузера Safari, сколько вам захочется. Вы думаете, что программисты Apple:

- Запрограммировали каждое из этих окон заранее, используя всю силу своего ума для прогноза, сколько же окон вы захотите открыть;
- Сделали своего рода шаблон и позволили Safari создавать объект окна из него на ходу.

Конечно правильный ответ под номером 2. Они создали некоторый код, называемый классом, который описывает что такое окно, как оно выглядит и ведет себя. Когда вы создаете новое окно, на самом деле этот класс создает его для вас. Этот класс представляет абстрактное окно и любое конкретное окно на самом деле – это экземпляр этого класса (как, например, 76 это просто экземпляр абстрактного класса «число»).

Поля

Созданные вами окна располагаются в определенном месте экрана. Если вы свернете окно в Dock, а затем развернете его, оно займет на экране ровно ту позицию что и раньше. Как это

работает? Класс определяет поля предназначенные для хранения положения окна на экране. Экземпляр класса (объект) хранит определенные значения в этих полях. Таким образом каждый объект хранит свои значения в соответствующих полях и в общем случае для разных окон значения этих полей будут разными.

Методы

Класс не только создает объект окна, но и дает ему доступ к набору действий, которые он может выполнять. Одно из этих действий — закрытие окна. Когда вы нажимаете кнопку «закрыть» на окне, кнопка посылает сообщение «закрыть» объекту окна. Действия которые может выполнять объект называются методами. Как вы можете видеть, они очень похожи на функции и вам не составит большого труда научиться их использованию, если вы будете следовать нашим инструкциям.

Объекты в памяти

Когда класс создает для вас объект окна, он резервирует память (RAM) для сохранения позиции окна и некоторой другой информации. Тем ни менее, он не делает копию кода закрывающего окно. Это было бы потерей компьютерной памяти потому, что этот код одинаков для каждого окна. Код закрывающий окно должен быть представлен только единожды, но каждый объект окна имеет доступ к этом коду принадлежащему классу окна.

Код, которой вы увидите в этой главе, содержит строки, которые служат для резервирования и освобождения памяти. Как уже было сказано раньше, мы не будем обсуждать эту сложную тему на этом этапе. Извините.

Наше приложение с GUI

Сейчас мы напишем программу, содержащую две кнопки и текстовое поле. При нажатии одной кнопки её значение будет введено в текстовое поле. При нажатии на вторую кнопку в текстовое поле будет введено другое значение. Получится программа наподобие двухкнопочного калькулятора, который не может выполнять какие-либо арифметические действия. Конечно, в процессе обучения, вы сможете написать настоящий калькулятор. Эта программа всего лишь один из первых шагов.

Если одна из кнопок нашего приложения будет нажата, будет послано сообщение. Это сообщение содержит название метода, который будет выполняться. Сообщение отправляется, хорошо, но куда? В случае окна сообщение послали в тот объект окна, который был экземпляром класса окна. Нам нужен объект, способный к получению сообщения от каждой из этих двух кнопок и способный «сказать» текстовому полю отобразить значение.

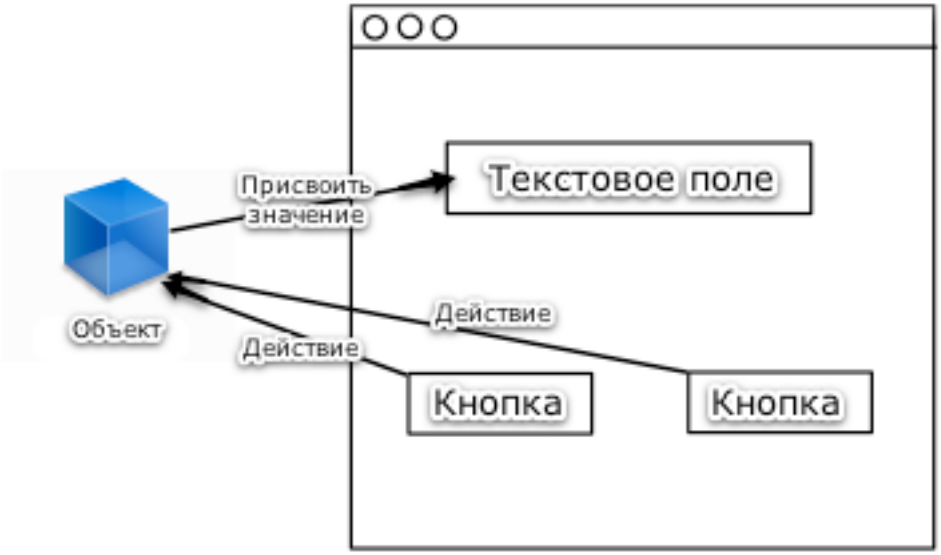
Набросок нашей будущей программы

Наш первый класс

Для начала создадим наш собственный класс и экземпляр этого класса. Этот объект будет принимать сообщение от кнопок. Также как и окно, наш экземпляр — это объект, но в отличии от объекта окна, мы не увидим наш экземпляр на экране во время выполнения программы. Он всего лишь находится в памяти.

Когда наш экземпляр получает сообщение от одной из двух кнопок, выполняется соответствующий метод. Код этого метода находится в классе (но не в конкретном экземпляре класса). Во время выполнения, этот метод заменит текст в текстовом поле.

Так как же метод одного класса знает как изменить текст в текстовом поле? На самом деле никак. Но это знает само текстовое поле. Таким образом, мы посылаем текстовому полю сообщение с просьбой это сделать. Что же это должно быть за сообщение? Конечно, нам необходимо указать имя получателя (например текстовое поле нашего окна). Также в сообщении необходимо указать, что получатель должен в итоге сделать. Мы определяем это указанием метода, который выполнит текстовое поле по получении сообщения. Конечно, нам нужно знать какие методы текстовое поле вообще способно выполнять и каким образом они вызываются. Также нам нужно сказать текстовому полю какое значение отобразить (в зависимости от нажатой кнопки). То есть, выражение отправки сообщения содержит не только имя объекта и имя метода, но и аргумент (значение), которое должен принимать метод объекта текстового поля.



Приведём набросок обмена сообщениями между объектами нашего приложения.

Здесь представлен общий формат того, как посылаются сообщения в Objective-C, как без аргумента [55.2], так и с ним [55.3]:

```
    //[55]
    [receiver message];
    [receiver messageWithArgument:theArgument];
```

Как вы можете видеть в каждом из этих блоков, структура, помещённая между квадратными скобками и завершающей точкой с запятой, является последним штрихом. Между скобками, объект-получатель упомянутый первым, следует за именем одного из его методов. Если вызываемый метод требует одно или несколько значений, они должны быть так же указаны [55.3].

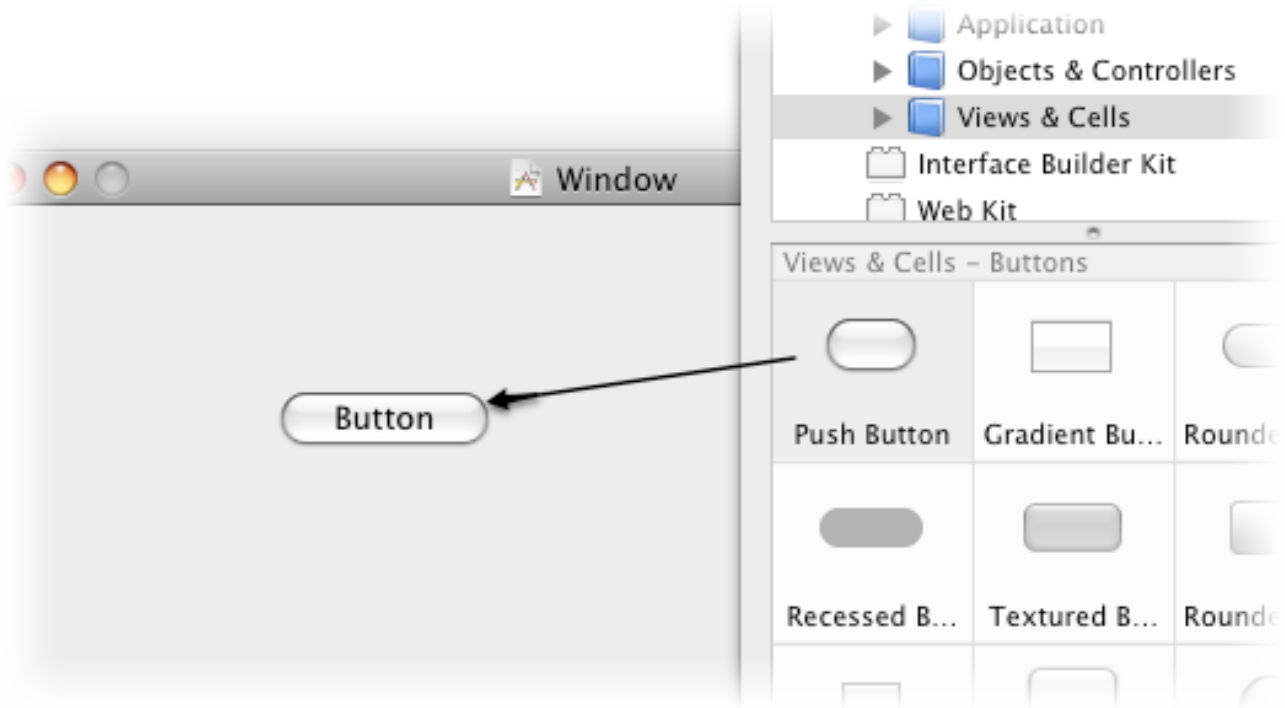
Создание проекта

Давайте взглянем, как это всё работает в действительности. Запустите Xcode для создания нового проекта. Выберите Cocoa Application под заголовком приложения, дайте название проекту (по существующему соглашению, названия приложений с GUI должны начинаться с заглавной буквы). В появившемся фрейме «Groups & Files» окна Xcode откройте каталог Resources. Дважды нажмите на MainMenu.xib (в более ранних версиях Xcode файл, хранящий ресурсы GUI, имел расширение nib — примечание редактора).

Создание графического интерфейса пользователя

Запустится другая программа — Interface Builder. Так как появится много окон, вы можете выбрать пункт Hide Others (Interface Builder ▸ Hide Others ⌥⌘N). Перед вами будут несколько окон. Первое, названое «Window», как раз то, которое будут видеть пользователи вашего приложения. Оно немного великовато, так что можете изменить его размер, как вам нравится. Справа от этого окна находится окно с названием «Library». Это хранилище для всех видов

объектов, которые вы можете использовать в своём GUI. Выберите элемент «Views & Cells» из списка в верхней части этого окна и перетащите две кнопки `Buttons` на ваше окно GUI — «Window». Аналогичным способом перетащите текстовую метку с текстом «Label».



Перетаскивание GUI объектов из окна палитры в окно вашего приложения.

Вдаваясь в подробности, при действии по перетаскиванию кнопки из палитры на окно вашего приложения, происходит создание объекта новой кнопки и помещение его в ваше окно. То же самое происходит для текстового поля и любого другого объекта, который вы можете перетащить из окна палитры.

Имейте ввиду, что если держать курсор над иконкой окна палитры, будет отображено имя, такое как `NSButton` или `NSTextView`. Это имена классов, созданных Apple. Позже в этой главе мы покажем как найти методы этих классов, требуемые нам для выполнения необходимых действий в нашей программе.

Выравнивайте объекты, которые вы перетащили на окно «Window» так, как вам нравится. Изменяйте их размеры, как душе угодно. Для изменения текста объекта кнопки дважды нажмите на нём мышью. Я предлагаю вам исследовать окно палитры как следует, после того как мы закончим этот проект, для того чтобы закрепить навыки добавки остальных объектов на ваше окно.

Обзор Interface Builder-а

Для изменения свойств объекта, выберите его и в инспекторе объектов отобразятся его свойства (открыть инспектор объектов — Tools ▸ Inspector ⇧⌘I). Рассмотрите их как следует. Например, выберите кнопку «Button» (вы можете увидеть как оно становиться выбранным в окне xib). Если в строке названия отобразилось «Button attributes», то вы можете изменить свойства данного объекта. Например указать новый заголовок у кнопки, указав новое значение в поле «Title». Вы увидите, что можно изменять внешний вид вашего приложения в широком диапазоне, без единой строчки написанного кода.

Немного о классах

Как мы обещали выше, мы создадим класс. Но перед этим, давайте внимательнее изучим как работают классы.

Для сохранения кучи сил, было бы замечательно создавать на основе того, что уже было создано и написано другими программистами, вместо того чтобы вручную разрабатывать с самого начала. Если вы, например, хотите создать окно с особыми свойствами (или возможностями), вам всего лишь требуется добавить код, реализующий эти возможности, и не нужно писать код, для обработки остального поведения окна, такого как закрытие или минимизация окна. Создавая на основе того что уже написано другими программистами, вы можете наследовать все реализованные ими типы поведения окна совершенно бесплатно. Этот факт отличает Objective-C от чистого C.

Как же это сделано? Итак, существует класс окна (NSWindow), и вы можете написать класс который наследуется от этого класса.

Возможно вы добавите какое-то определённое поведение в ваш собственный класс окна. Что же произойдет если ваше окно получит сообщение «Close»? Ведь вы не писали никакого кода для обработки этого сообщения, а так же не копировали код для этих целей в ваш класс. Попросту говоря, если класс вашего окна не содержит кода для определённого метода, сообщение автоматически передается для обработки классу, от которого было унаследовано ваше окно (т.е. его суперклассу). И если необходимо, этот процесс повторяется до тех пор, пока обработчик сообщения не будет найден (или не будет достигнут самый родительский класс всей иерархии наследования в данном случае).

Если метод не может быть найден, то посланное вами сообщение не будет обработано. Это словно поиск гаража для смены шин, когда даже его владелец не в силах оказать помощь вам. В таких случаях Objective-C укажет на ошибку.

Пользовательские классы

А что, если вы хотите использовать свою собственную обработку для метода, уже унаследованного от родительского класса? Это легко, вы имеете возможность переопределить необходимые методы. Например, вы можете написать код, который при нажатии кнопки закрытия окна, перемещает его из области видимости, ещё до действительного его закрытия. Ваш особый класс окна может использовать те же имена методов для закрытия окна, что определены Apple. Итак, когда ваше окно получает сообщение о закрытии, выполняется ваш переопределённый метод, вместо метода определённого Apple. Таким образом, окно может убираться из области видимости, до того как оно будет фактически закрыто.

Да, обработка закрытия окна уже реализована программистами Apple. Внутри нашего метода закрытия окна мы легко можем вызывать метод закрытия окна-родителя, хотя порядок его вызова слегка отличается, для того чтобы удостовериться, что наш собственный метод не будет вызываться рекурсивно.

```
    //[56]
    // Код для перемещения окна из области видимости.
    [super close]; // Вызов метода close родительского класса
```

Этот код слишком наворочен для этого вводного буклета и мы не ждем, что вы тут же вникните в его смысл, прочитав эти несколько строк.

Существует один класс, являющийся родительским для всех остальных, как царь горы, и называется он NSObject. В общем, все классы, которые вы когда-либо создадите или будете использовать, будут являться классами-потомками класса NSObject, прямо или косвенно. Например класс NSWindow является потомком от класса NSResponder, который в свою очередь потомок от NSObject. В классе NSObject определены общие методы для всех объектов (например генерация текстового описания объекта, опрос объекта способен ли он обработать передаваемое сообщение и т. д.).

Перед тем, как я усыплю вас кучей теории, давайте посмотрим, как создать класс.

Создание нашего класса

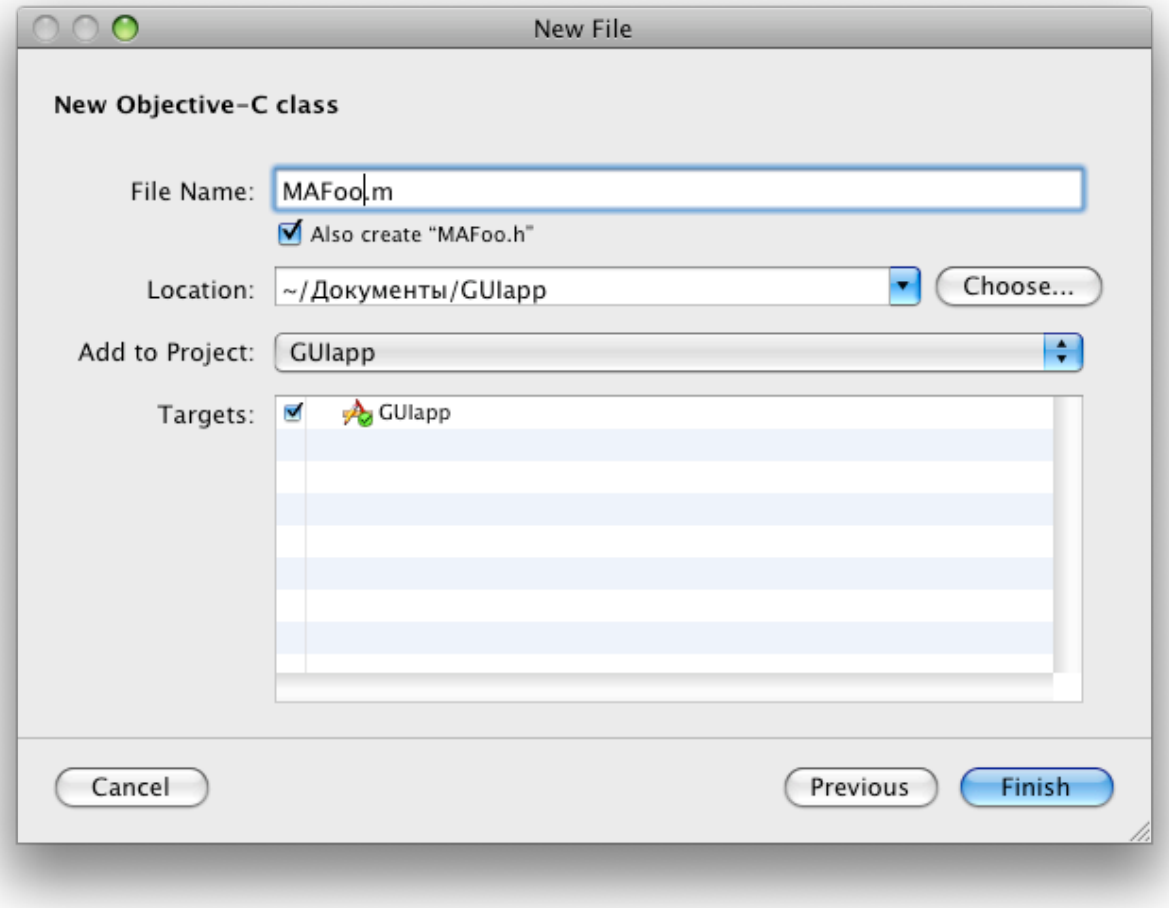
Перейдите на ваш проект в Xcode и создайте новый файл (File ▸ New File… ⌘N). Выберите Objective-C class из списка, затем нажмите кнопку «Next». Задайте имя класса. Я назвал его «MAFoo» (имя файла класса «MAFoo.m» — примечание редактора). Нажмите кнопку «Finish».

Первые две заглавные буквы «MAFoo» обозначают «My Application». Вы можете назвать класс так, как вам нравится. Как только вы начнете писать свои собственные приложения, мы рекомендуем вам выбрать две или три буквы, которые вы будете использовать для всех классов, чтобы избежать беспорядка с существующими названиями класса. Однако не используйте NS, поскольку это может смутить вас потом. NS используется для классов Apple. Оно обозначает NeXTStep, NeXTStep — операционная система, на которой был основана Mac OS X после того, как Apple, Inc. купила NeXT и вернула Стива Джобса в качестве бонуса.

CocoaDev вики содержит список других префиксов, которые нужно избегать. Вы должны проверить их при выборе собственного:

http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix

При создании нового класса вы должны дать ему название, которое передает полезную информацию об этом классе. Например, мы уже видели, что в Cocoa класс, используемый для представления окон называется NSWindow. Другим примером является класс, который используется для представления цветов, который называется NSColor. В нашем случае, MAFoo класс мы создаем только здесь, чтобы проиллюстрировать способ объектов общаться вместе в приложении. Именно поэтому мы дали ему общее имя без какого-либо специального смысла.



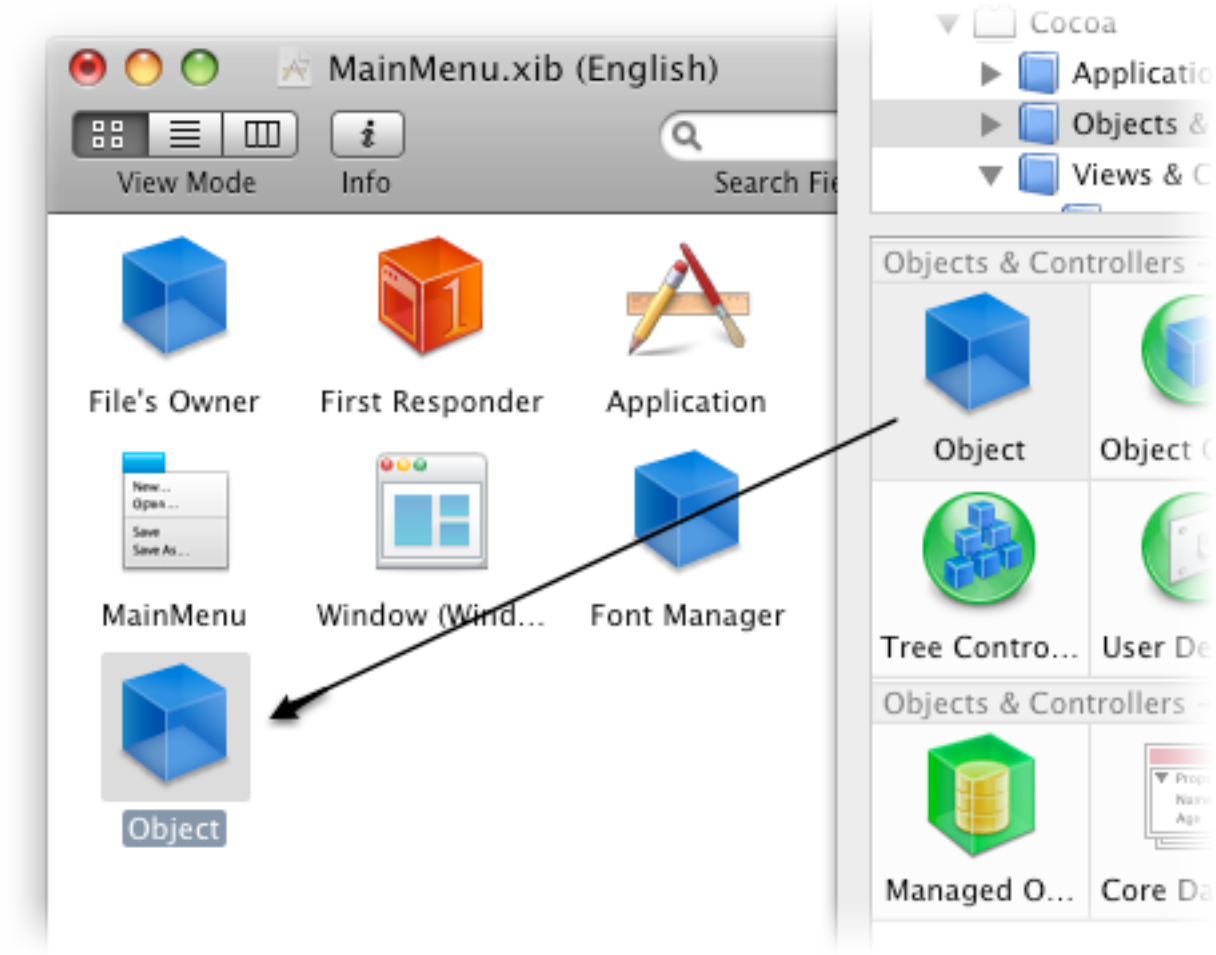
Создание класса «MAFoo»

Создание экземпляра класса в Interface Builder

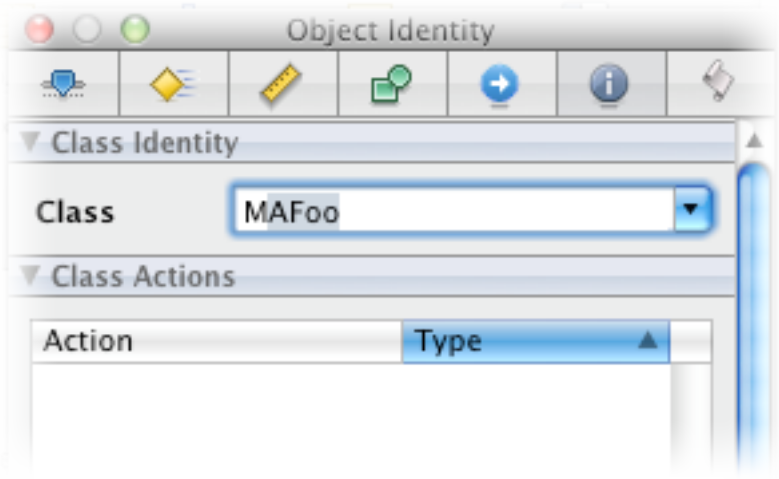
Вернитесь в Interface Builder, перейдите на палитру «Library» и выберите «Objects & Controllers» из верхнего меню. Теперь тяните Object (синий куб) от палитры до класса MainMenu.xib.

Затем выберите вкладку Identity в инспекторе объектов (⌘6) и следом отметьте MAFoo из всплывающего меню «Class».

Теперь мы связали MAFoo класс в Xcode с объектом в xib-файле. Это позволит общаться нашему коду и нашему интерфейсу.



Создание нового объекта



Идентификация экземпляра нашего объекта

Создание связей

Наш следующий шаг заключается в создании связей между кнопками от которых посылаются сообщения до нашего объекта MAFoo. Кроме того, мы собираемся создать соединение от объекта MAFoo к текстовому полю, потому что сообщение должно быть отправлено в текстовое поле. Объект не имеет способа отправлять сообщения к другому объекту, если связи с ним нет. Сделав связи между кнопкой и объектом MAFoo, мы показываем, что кнопка может посылать сообщения на объект MAFoo. С помощью этой связи кнопки будут иметь возможность отправлять сообщения на MAFoo. Аналогичным образом установление связи от нашего объекта к текстовому полю позволит отсылать ему сообщения.

Давайте еще раз пройдем то, что должно сделать приложение. Каждая из кнопок при нажатии может отсылать сообщение, соответствующее специфическому действию. Это сообщение содержит название метода класса MAFoo, который должен быть выполнен. Сообщение посылается в объект MAFoo — экземпляр класса MAFoo, который мы только что создали. Помните, что сами объектные экземпляры класса не содержат код, который выполняет действие, но классы делают это. Итак, это сообщение, посланное в объект MAFoo, вызывает метод класса MAFoo, чтобы сделать что-нибудь. В нашем случае, посылая сообщение в объект текстового поля. Как и любое сообщение, оно состоит из названия метода, который объект должен будет выполнить. В этом случае, у метода объекта текстового поля есть задача показа значения, и это значение нужно послать как часть сообщения (названное «параметром», помните?), наряду с названием метода, чтобы вызвать его.

Наш класс требует 2 действия (action), которые будут вызваны объектами кнопок. Класс нуждается в одном выходе (outlet), переменной для того, чтобы помнить, какому объекту (т.е. объекту текстового поля) будет послано сообщение.

Удостоверьтесь, что MAFoo выбран в окне MainFile.xib. Затем выберите вкладку Identity в инспекторе объектов (⌘6). В окне инспектора в секции Actions, нажмите кнопку добавить (нажмите на +, который располагается под списком), чтобы добавить действие (т.е., метод действия) к классу MAFoo. Замените название, заданное Interface Builder более понятным именем (например, вы можете ввести «setTo5:» потому что мы будем программировать этот метод, чтобы отобразить номер 5 в текстовом поле). Добавьте другой метод, и дайте ему название (например «reset:», потому что мы будем программировать его, чтобы отобразить число 0 в текстовом поле). Заметьте, что названия наших методов оканчиваются двоеточием (:). Больше об этом позже.

Теперь в окне инспектора выберите вкладку Outlet, добавьте выход (outlet) и дайте ему название (например, «textField»).

Добавляя действия (actions) и выходы (outlets) к классу MAFoo перед тем, как связать объекты, мы должны дать значащие названия нашим двум кнопкам. Т.к. первая будет «просить», чтобы наш экземпляр MAFoo показал число 5 в текстовом поле, мы назовем ее «Set to 5» (мы уже знаем, как изменить название кнопки: двойной щелчок на ее имени на экране, и затем вводим новое имя).

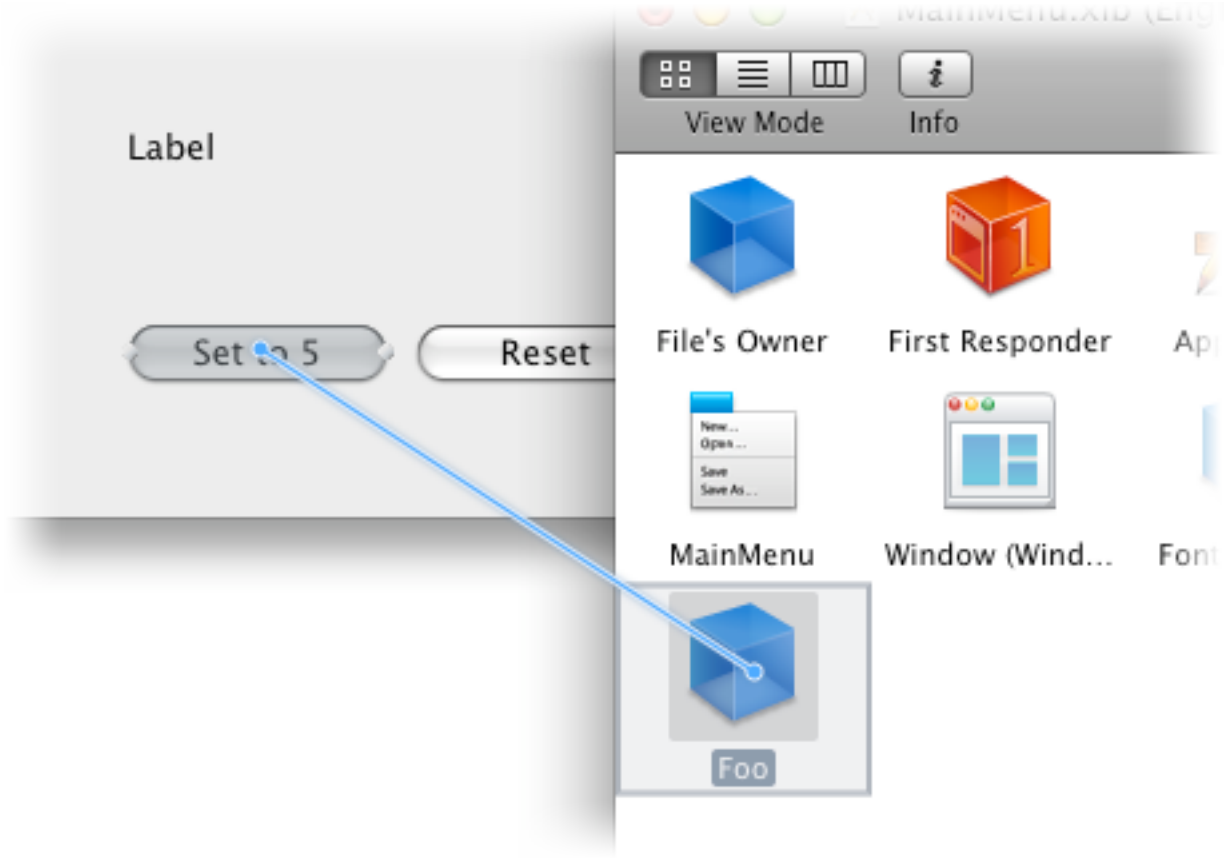
Кроме этого, вторую мы назовем «Reset». Заметим, что называть кнопку специфически не обязательно для работы программы. Это нужно только для того, чтобы наш пользовательский интерфейс был максимально информативным для конечного пользователя.

Теперь мы готовы создать нужные связи между:

- Кнопкой "Reset" и экземпляром MAFoo;
- Кнопкой "Set to 5" и экземпляром MAFoo;
- Экземпляром MAFoo и текстовым полем.

Для создания связи нажмите кнопку ctrl ^ на клавиатуре и используйте мышь для перетаскивания от кнопки «Set to 5» до экземпляра MAFoo в окне MainMenu.xib (не делайте этого наоборот!). Линия, представляющая соединение, будет появляться на экране, и меню будет всплывать на иконке представления объекта. Выберите «setTo5:» из списка.

> *Так же операцию связывания можно произвести просто зажав правую кнопку мыши вместо нажатия ctrl ^ (прим. редактора).*



Установление соединения

Теперь кнопка имеет ссылку на объект MAFoo, и будет направлять ему сообщение «setTo5:» всякий раз при нажатии.

Вы можете связать кнопку «Reset» с объектом MAFoo таким же способом.

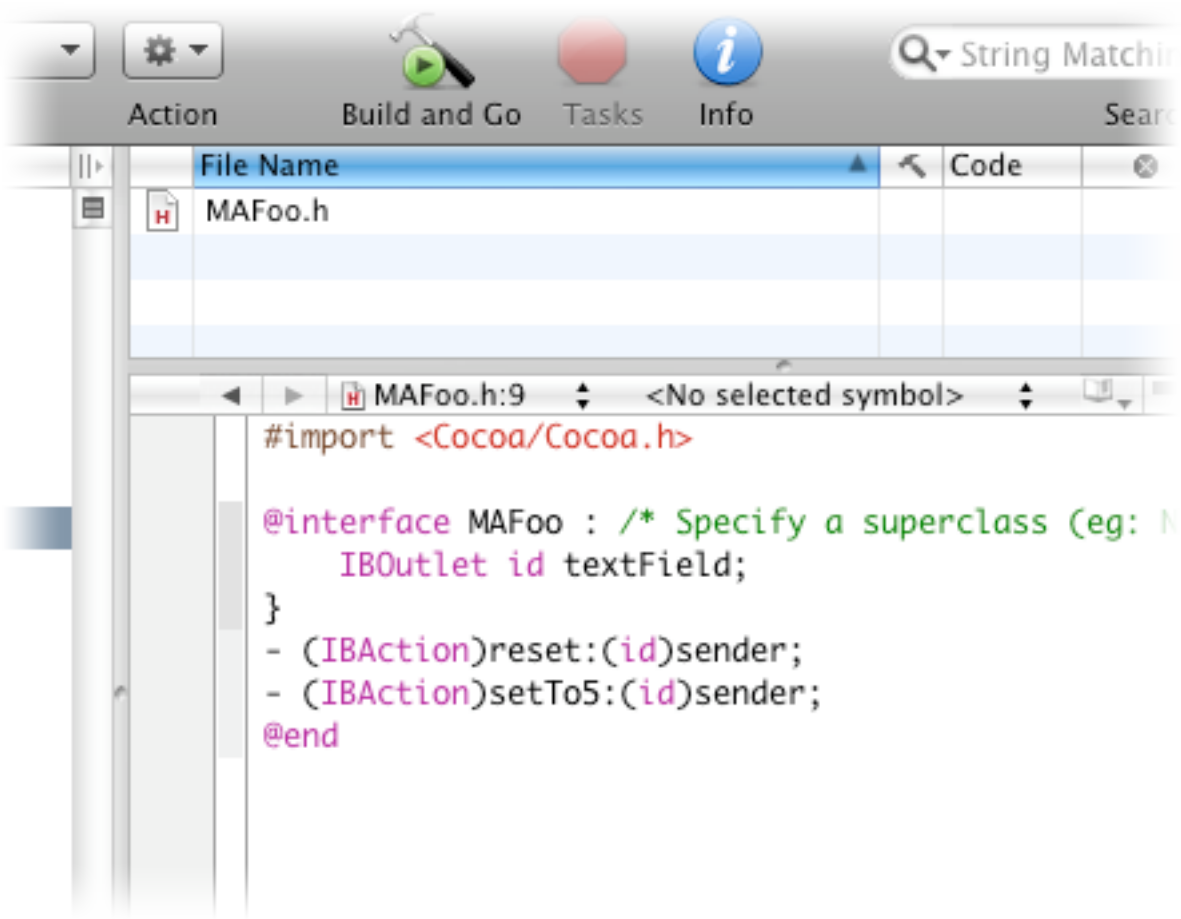
Для создания связи между объектом MAFoo и текстовым полем начните от MAFoo и с нажатой кнопкой ctrl ^ перетащите линию до этого поля. Выберите «textField» из меню чтобы присвоить связь.

О чем это все? Хорошо, как вы сейчас увидите, вы только что создали немного кода без написания единой строки.

Создание кода

Перейдите в меню File в Interface Builder'е и выберите «Write Class Files». Interface Builder спросит у вас, куда конкретно на диск вы хотите поместить сгенерированные файлы. Перейдите в каталог с вашим проектом и замените имеющийся класс MAFoo тем, что вы создали только что.

Теперь, если вы переключитесь обратно в Xcode, вы увидите сгенерированные файлы в окне вашего проекта, внутри группы Classes. Нажмите кнопку Editor и затем выберите файл MAFoo.h.



Сгенерированные файлы появились в вашем проекте Xcode.

Давайте вернемся на минутку к Главе 4, туда, где мы обсуждали функции. Помните ли вы заголовок функции [41.2]?

Это было своего рода предупреждение для компилятора, чтобы сказать ему, что можно ожидать. Один из этих двух файлов, которые мы только что создали, называется MAFoo.h, и это — файл заголовка: он содержит информацию о нашем классе. Например: вы признаете, что есть строка [57.5], содержащая NSObject, в которой говорится, что наш класс наследуется от NSObject.

```
//[57]
/* MAFoo */
#import <cocoa/cocoa.h> // [57.3]

@interface MAFoo : NSObject
{
  IBOutlet id textField; // [57.7]
}
- (IBAction)reset:(id)sender;
- (IBAction)setTo5:(id)sender;
@end
```

Вы видите, что есть один выход (IBOutlet) [57.7] в текстовое поле. Id указывает на объект. Префикс «IB» — это Interface Builder, его вы использовали для создания этого кода.

IBAction [57.9, 57.10] равнозначен void. Объекту, который посылает сообщение, не возвращается ничего: кнопки в нашей программе не получают ответа от объекта MAFoo в ответ на их сообщение.

Также вы можете видеть, что есть 2 действия (action) программы Inteface Builder. Это 2 метода нашего класса. Методы очень похожи на функции, которые мы уже знаем, но есть различия. Больше о них позже.

Ранее мы видели #import <Foundation/Foundation.h> вместо строки [57.3], Это было для приложений без GUI, #import <cocoa/cocoa.h> же используется для GUI приложений.

Давайте посмотрим второй файл, MAFoo.m. Опять мы получаем много кода.

```
//[58]
#import "MAFoo.h"
@implementation MAFoo

- (IBAction)reset:(id)sender // [58.5]
{
}
- (IBAction)setTo5:(id)sender
{
}
@end
```

Во-первых, импортирован файл заголовка MAFoo.h, таким образом компилятор знает, что ожидать. Могут быть узнаны два метода: Reset: и setTo5:. Это методы нашего класса. Они подобны функциям в том, что вы должны поместить свой код между фигурными скобками. В нашем приложении при нажатии кнопки посылается сообщение в объект MAFoo, прося выполнения одного из методов. Мы не должны написать код для этого. Создание связей между кнопками и объектом MAFoo в Interface Builder является всем, что нужно. Однако мы должны реализовать эти два метода, т.е. мы должны написать код, который выполняет их функции. В этом случае эти методы только посылают сообщение каждому из экземпляров MAFoo для объекта textField, и мы их описываем как [58b.7, 58b.11].

```
//[58b]
#import "MAFoo.h"
@implementation MAFoo

- (IBAction)reset:(id)sender
{
  [textField setIntValue:0]; // [58b.7]
}
- (IBAction)setTo5:(id)sender
{
  [textField setIntValue:5]; // [58b.11]
}
@end
```

Как вы можете видеть, мы посылаем сообщение объекту, на который ссылается выход (outlet) textField. Т.к. мы соединили этот выход с текстовым полем, используя Interface Builder, наше сообщение будет послано в нужный объект. Сообщение с названием `setIntValue:` вместе с целочисленным значением. Метод `setIntValue:` способен к показу целочисленного значения в объекте текстового поля. В следующей главе мы расскажем, как узнали о нем.

Готовимся зажигать

Теперь вы готовы скомпилировать и запустить свое приложение. Обычно нужно нажать кнопку Build and Go на панели инструментов Xcode. Xcode нужно несколько секунд, чтобы собрать и запустить программу. Наконец приложение появится на экране и вы сможете протестировать его.

Наше приложение работает.

Итак, вы только что создали (очень простое) приложение, для которого сами написали всего 2 строчки кода.

Глава 9: Поиск Методов

В предыдущей главе мы узнали о методах. Мы написали 2 метода сами, но так же использовали 1, предложенный Apple. `setIntValue:` — метод для показа целочисленного значения в текстовом поле. Как мы можем посмотреть доступные методы?

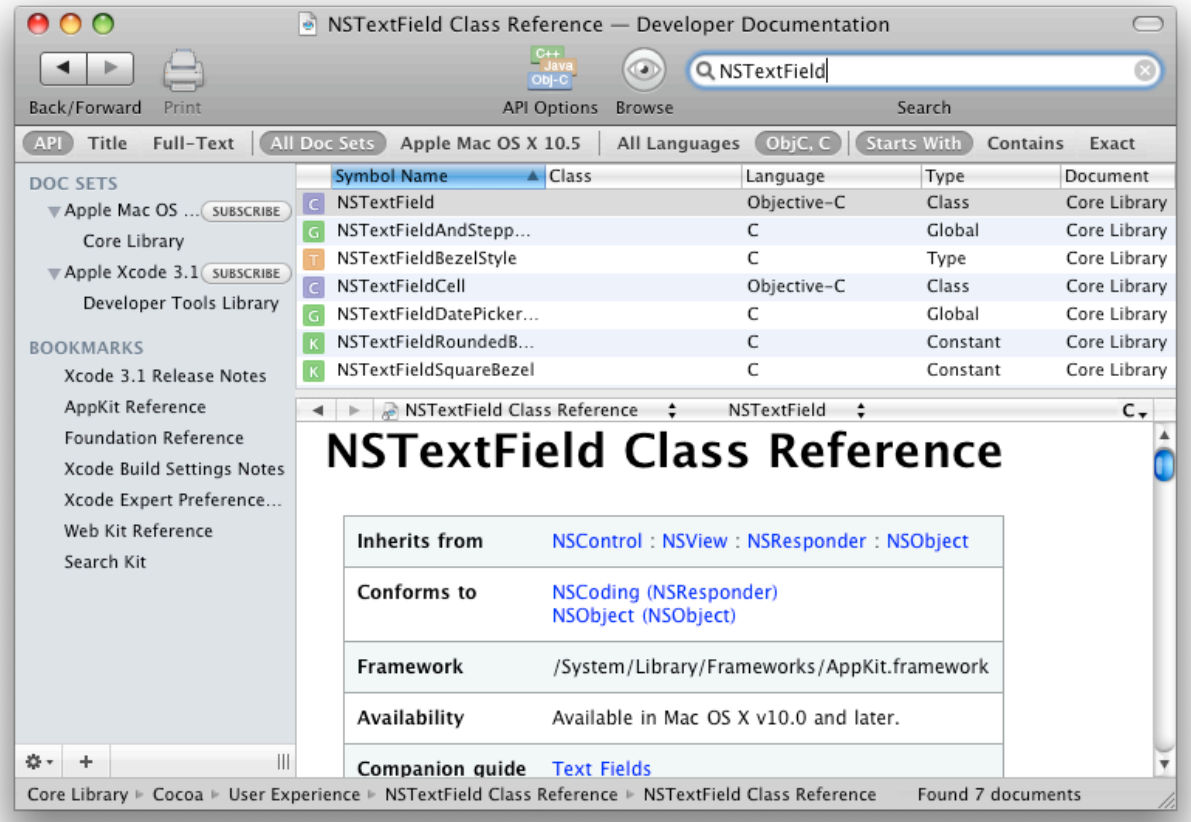
> *Помните, для каждого использованного вами метода, созданного Apple, вы не должны писать код самостоятельно. И скорее всего, они написаны без ошибок. Поэтому стоит всегда проверять, доступны ли подходящие методы перед написанием своего собственного.*

Если вы задержите курсор мыши над объектом в окне палитр в Interface Builder, то появится всплывающая ссылка (tooltip, вероятно). Если вы установите курсор мыши над значком кнопки, то увидите «NSButton». Над текстовым полем увидите «NSTextField». Каждое из этих названий — имя класса. Давайте проверим класс `NSTextField`, чтобы увидеть, какие методы доступны.

Вернитесь к Xcode и переключитесь на окно документации разработчика (Help ▸ Documentation ⌥⌘?). В фрейме слева выберите Apple Mac OS X 10.5, затем введите «NSTextField» в поле поиска (убедитесь что выбран режим поиска API). По ходу набора текста в поле поиска, список возможных вариантов существенно уменьшится и вскоре вы увидите появившийся сверху `NSTextField`.

Щёлкните по строке `NSTextField` (of type Class) чтобы получить информацию по классу `NSTextField`, которая отобразится в нижнем фрейме.

Первое, что вы должны учитывать, этот класс наследуется от целого ряда других классов. Последний из них вверху списка это базовый класс `NSObject`.



Навигация по документации Cocoa при помощи Xcode

Типы методов

Здесь-то мы и начнём наш поиск. Быстрый взгляд по заголовкам скажет нам, что мы не найдем метод, который требуется для отображения значения в текстовом поле объекта. Из-за особенностей принципов наследования, нам нужно рассмотреть непосредственный базовый класс для класса `NSTextField`, которым является `NSControl` (если мы и здесь ничего не найдем, то следует просмотреть базовый класс для `NSControl`, и так далее вверх по иерархии). Так как вся документация выполнена в виде HTML-документа, нам следует щёлкнуть мышью по слову `NSControl` (как показано выше в пункте Inherits). Это даст нам информацию по классу `NSControl`.

NSControl

Наследуется из `NSView` : `NSResponder` : `NSObject`

Как видите мы перенесли один класс вверх. В списке методов мы видим подзаголовок:

Установка значения контрола (Setting the Control's Value)

То что нам нужно, так это установить значение. Ниже заголовка мы обнаружим:

– setIntValue:

Выглядит многообещающе, так что мы посмотрим описание этого метода щёлкнув по ссылке `setIntValue:`.

setIntValue:

```
- (void)setIntValue:(int)anInt
```

Устанавливает значение ячейки назначения (или выбранной ячейки) в целочисленное anInt. Если ячейка в настоящее время редактируется, он прерывает редактирование до установки значения, если ячейки не наследуют NSActionCell он отмечает свойства ячейки как требующие обновления(NSActionCell выполняет свои собственные обновления ячеек)

В наших приложениях объект NSTextField является получателем, и мы должны передать ему целочисленное значение. Мы можем увидеть также такую форму записи метода:

```
- (void)setIntValue:(int)anInt
```

В Objective-C знак минуса обозначает начало объявления метода объекта (в противоположность объявлению метода класса, о котором мы поговорим позже). Void показывает то, что никакое значение не возвращается в вызывающий этот метод объект. Таким образом, если мы посылаем сообщение setIntValue объекту textField, наш MAFoo объект не получает назад никакого значения от объекта текстового поля. И это нормально. После двоеточия (int) говорит нам о том, что переменная anInt должна быть целочисленного типа. В нашем примере мы посылали значение 5 или 0, кои являются целочисленными, и все было замечательно. Иногда немного сложно понять какой же метод использовать в данной ситуации. Тут вы лучше себя будете чувствовать, если глубже ознакомитесь с документацией, так что дерзайте.

А что делать, если вам, например, захотелось прочитать значение из нашего текстового поля объекта textField? Вы помните тот факт, что переменные внутри функций защищены? То же справедливо и для методов. Однако, часто объекты имеют пару близких методов, называемых аксессоры, один для чтения значения, и другой для присваивания значения. Нам уже известно о последнем методе, это setIntValue, первый же выглядит так:

```
//[59]
- (int) intValue
```

Как видите, этот метод возвращает целое. Таким образом, если мы хотим прочитать целочисленное значение, ассоциированное с нашим текстовым полем, нам нужно послать сообщение вот так:

```
//[60]
int resultReceived = [textField intValue];
```

Ещё раз заострим внимание на том факте, что в функциях (собственно как и в методах) все имена переменных защищены. Это очень полезно, так как вам не нужно беспокоится о том, что присвоение значения переменной в одной части вашей программы повлияет на переменную с тем же именем в вашей функции. Однако, имена функций, все же, должны быть уникальными в пределах всей программы. Objective-C обеспечивает защиту на один шаг вперёд: имена методов должны быть уникальными только в пределах одного класса, но разные классы могут содержать одноимённые методы. Это большое подспорье для больших программ, потому что программисты могут писать классы независимо друг от друга, без боязни возникновения конфликтов между именами методов.

Более того, различные методы разных классов могут иметь одинаковые имена, этот принцип называется полиморфизм (от греческого много-форменность), и является одной тех особенностей, которая выделяет объектно-ориентированное программирование.

Полиморфизм позволяет вам писать куски кода без знания того, что за классами объектов вы манипулируете. Все это нужно для того, чтобы во время выполнения соответствующие объекты могли обрабатывать сообщения, посланные вами.

Используя преимущества полиморфизма, вы можете писать приложения, являющиеся весьма гибкими и расширяемыми во время разработки. В качестве примера, если мы создали GUI-приложение, затем заменили объект класса текстового поля объектом другого класса, который может обрабатывать сообщение `setIntValue`, наше приложение все ещё будет работоспособно без необходимости что-то менять в коде или даже без перекомпиляции.

Глава 10: awakeFromNib

> *В ранних версиях Interface Builder использовался файл с расширением nib. Начиная с версии 3.0 используется файл с расширением xib. Данная глава была адаптированна под новую версию Interface Builder. (прим. редактора)*

Apple выполнили кучу работы за вас, позволив вам сделать создание приложений ещё более лёгким. В вашем маленьком приложении вам не нужно беспокоиться об отрисовке окна и кнопок на экране и ещё много о чём.

Большинство из этого стало возможным благодаря двум фреймворкам. Это фреймворк Foundation Kit, который мы импортировали в примере [41] главы 4, он выполняет работу по обеспечению большинства сервисов, не связанных с графическим интерфейсом пользователя. Другой фреймворк называется Application Kit, он имеет дело с объектами, которые вы видите на экране, а так же с механизмами взаимодействия с пользователем. Оба фреймворка хорошо документированы.

Теперь же вернемся к нашему GUI-приложению. Допустим, неплохо бы, чтобы наше приложение отображало определенное значение в объекте текстового поля сразу же после запуска и начального отображения окна.

Вся информация для окна хранится в xib-файле (nib служит для NeXT Interface Builder'a). Это очень хорошая подсказка того, что требуемый нам метод может являться частью Application Kit. Давайте-ка узнаем как получить информацию об этом фреймворке.

Вернитесь к Xcode и переключитесь на окно документации разработчика (Help ▸ Documentation ⌥⌘?). В окне документации убедитесь, что поиск производится по заголовкам (на панеле поиска выбран пункт «Title»).

Затем введите «Application Kit» в поле поиска и нажмите Return ↩.

XCode предоставит несколько результатов (поиска). Среди них документ «Application Kit Framework Reference».

Внутри документации вы найдете перечень сервисов, которые предоставляет этот фреймворк. В разделе Протоколов (Protocol Reference) есть ссылка «NSNibAwaking». Если вы пройдете по ней, вы найдете документацию по классу NSNibAwaking.

Руководство по протоколу NSNibAwaking
(неофициальный протокол)

Framework

```
/System/Library/Frameworks/AppKit.framework
```

Объявлен в

```
AppKit/NSNibLoading.h
```

Сопровождающий документ

Загрузка ресурсов (Resource Programming Guide)

Описание протокола

Этот неофициальный протокол состоит из одного метода: awakeFromNib. Классы могут использовать этот метод для инициализации информации состояния после того как объекты были загружены из архива Interface Builder (nib-файла).

Если мы используем этот метод, он будет вызван после загрузки наших объектов из своего nib-файла. Поэтому мы можем использовать его для достижения нашей цели: отображения значения в текстовом поле во время запуска.

> *Я не хочу сказать, что нахождение правильного метода является всегда тривиальной задачей. Часто требуется некоторое время полистать документацию и с умом использовать ключевые слова для поисков, чтобы найти метод так нужен. По этой причине, для вас очень важно ознакомиться с документацией обоих фреймворков, так чтобы вы знали какие классы и методы вам доступны. Может быть сейчас это вам ещё не требуется, но оно поможет вам выяснить как заставить свою программу делать то что вы хотите.*

Хорошо, мы нашли метод, все что осталось, так это добавить его в наш файл MAFoo.m [61.13].

```
//[61]
#import "MAFoo.h"

@implementation MAFoo
- (IBAction)reset:(id)sender
{
  [textField setIntValue:0];
}
- (IBAction)setTo5:(id)sender
{
  [textField setIntValue:5];
}
- (void)awakeFromNib // [61.13]
{
  [textField setIntValue:0];
}
@end
```

Когда окно открывается, метод awakeFromNib будет вызван автоматически. Как результат, текстовое поле покажет ноль когда вы посмотрите на вновь открытое окно.

Глава 11: Указатели

> *Эта глава содержит сложные понятия представляющие основу языка C. Эти понятия могут отпугнуть начинающих программистов. Но не бойтесь, если вы не поймёте всё сразу! Понимание принципа работы указателей очень полезно, но к счастью не сильно важно на начальном этапе изучения Objectice-C.*

Когда вы декларируете переменную ваш Mac соединит эту переменную с областью в памяти, для хранения ее значения.

Для примера рассмотрим следующие инструкции:

```
//[62]
int x = 4;
```

При запуске, ваш Mac найдет немного незанятого места в памяти и отметит, что в место теперь записана переменная x (конечно мы могли и должны были использовать более подходящее имя для нашей переменной). Посмотрите на инструкцию [62] опять. Указание типа переменной (в данном случае int) говорит компьютеру сколько памяти требуется для хранения переменной x. Если вы укажете тип переменной `long` или `double`, нужно будет выделить больше места.

Инструкция присвоения `x = 4` присваивает значение 4 в зарезервированной памяти. Конечно, компьютер запоминает где в памяти было сохранено значение переменой x. Другими словами запоминая адрес переменной x. Поэтому, каждый раз, когда вы используете x в вашей программе, компьютер «смотрит» в нужном месте (по правильному адресу) и может найти актуальное значение переменной x.

Указатель — это просто переменная, хранящая **адрес** другой переменной.

Ссылки на переменные

Адрес переменной можно получить добавлением символа & перед переменной. К примеру, чтобы получить адрес переменной x, напишите &x.

Когда компьютер вычисляет выражение x он возвращает значение переменной x (в нашем примере он вернет 4). В противоположность этому, когда компьютер вычисляет выражение &x, он возвращает адрес переменной x, но не её значение. Адрес это номер который обозначает определённое место в памяти компьютера (что-то типа того как номер комнаты обозначает вполне определённую комнату в отеле)

Использование указателей

Указатель можно объявить так:

```
//[63]
int *y;
```

Эта команда определяет переменную y, которая содержит адрес переменной целочисленного типа (на данный момент она не содержит адрес переменной, т.к. не произведена инициализация — прим. редактора). Повторяю: она не содержит переменную типа int, но она содержит адрес в памяти, где лежит эта переменная. Для того чтобы присвоить переменной y адрес переменной x вам нужно написать следующий код:

```
//[64]
y = &x;
```

Теперь переменная y указывает на адрес переменной x. Используя переменную y вы можете отыскать переменную x. Как это сделать.

Вы можете узнать значение на которое указывает переменная y путем добавления звездочки перед указателем. Например, выражение *y возвратит значение 4. Оно дает такой же результат, как если бы обратились непосредственное к переменной x. Выполнение команды:

```
*y = 5;
```

равносильно выполнению:

```
x = 5;
```

Использование указателей удобно, потому что вы можете захотеть обращаться не к значению переменной, а к ее адресу. Например, вы можете захотеть создать функцию, которая увеличивает значение переменной на 1. Что ж, можно это сделать вот так:

```
//[65]
void increment(int x)
{
  x = x + 1;
}
```

Но у вас ничего не получится. Если вызвать эту функцию из программы, вы не получите тот результат, который ожидаете.

```
//[66]
int myValue = 6;
increment(myValue);
NSLog(@"%d:\n", myValue);
```

Результатом выполнения этого кода будет 6. Почему? Разве вы не увеличили значение переменной myValue вызвав функцию инкремента? Нет. Смотрите, функция [65] берет только значение переменной myValue, увеличивает его на единицу и... прекращает выполнение. Функция работает только со значением переменной, которую вы указали в качестве аргумента, а не с переменной, которая хранит это значение. Даже если вы изменили значение x, вы изменили только значение переменной, которое получила функция.

Любое такое изменение будет потеряно, когда будет осуществлен возврат из функции. Кроме того x даже не обязательно присваивать переменную. Если вы напишите `increment(5);` какой результат получите?

Если вы хотите написать функцию инкремента, которая действительно работает, то есть принимает в качестве аргумента переменную и изменяет значение этой переменной вам нужно передать в функцию адрес этой переменной. Итак, используйте указатель как аргумент:

```
//[67]
void increment(int *y)
{
  *y = *y + 1;
}
```

затем вы можете сделать, например, такой вызов функции:

```
//[68]
int myValue = 6;
increment(&myValue); // передать адрес
NSLog(@"%d:\n", myValue); // теперь myValue равно 7
```

Глава 12: Строки

Выше мы рассмотрели несколько основных типов данных: `int`, `long`, `float`, `double`, `BOOL`. Также, в последней главе, ознакомились с указателями. А так же немного обсудили строки (`strings`) в связи с функцией `NSLog()`.

Эта функция позволяет выводить строку на экран, заменяя код начинающийся %-буква, например `%d`, значением.

```
//[69]
float piValue = 3.14159;
NSLog(@"Три примера вывода строки на экран.\n");
NSLog(@"π равно %10.4f.\n", piValue);
NSLog(@"У кубика %d граней.\n", 6);
```

Мы не обсуждали строки как тип данных по веской причине. В отличие от типов `int` или `float`, объекты типа `string` являются настоящими объектами. Они создаются с использованием класса `NSString` или класса `NSMutableString`. Давайте обсудим эти классы. Начнем с `NSString`.

NSString

Опять указатели

```
//[70]
NSString *favoriteComputer;
favoriteComputer = @"Mac!";
NSLog(favoriteComputer);
```

Возможно, вам понятен смысл второй строчки кода, первая строка вносит еще большее объяснение. Помните, что когда мы объявляем указатель, мы должны сказать какого типа он будет. Ниже выражение из Главы 11 [71].

```
//[71]
int *y;
```

Здесь мы говорим компилятору, что указатель y содержит адрес в памяти где хранится целочисленное значение.

В [70.2] мы сказали компилятору, что указатель `favoriteComputer` содержит адрес расположения объекта типа `NSString` в памяти. Мы использовали указатель для хранения нашей строки так как в Objective-C объектами никогда не манипулируют напрямую, но только через указатели на эти объекты.

Не волнуйтесь сильно, если не поняли все это до конца, это не критично. То, что действительно важно, так это постоянно ссылаться на сущность типа `NSString` или `NSMutableString` (или любого другого объекта), используя нотацию «*».

Символ @

Ну почему же этот забавный символ @ (читается как «эт» — прим. переводчика), все время выделяется? Ну, язык Objective-C является расширением C подобных языков и имеет свое

собственное обозначение для работы со строками. Для того, чтобы можно было различить новые типы строк, являющихся полноценными объектами, Objective-C использует символ @.

Новый вид строк
Какие же улучшения в типе строк, по сравнению с языком C? Строки в Objective-C поддерживают юникод, в отличие от ASCII в C. Строки юникод могут содержать символы любых алфавитов мира, такие как китайские иероглифы или кириллицу.

Конечно, можно объявить и инициализировать переменную указатель в одной строке [72].

```
//[72]
NSString *favoriteActress = @"Юлия";
```

Переменная указатель `favoriteActress` указывает на область в которой хранится объект представляющий строку «Юлия».

Как только вы проинициализировали переменную, например `favoriteComputer`, вы можете присвоить ей любое другое значение, но не можете изменить саму строку [73.9] так как она является экземпляром класса `NSString`. Подробнее об этом через пару минут.

```
//[73]
#import <foundation/foundation.h>

int main (int argc, const char *argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  NSString *favoriteComputer;
  favoriteComputer = @"iBook"; // [73.9]
  favoriteComputer = @"MacBook Pro";
  NSLog(@"%@", favoriteComputer);
  [pool release];
  return 0;
}
```

При запуске программа напишет:

```
MacBook Pro
```

NSMutableString

Строка класса `NSString` называется неизменяемой, так как она не может быть модифицирована.

Что хорошего в строке, которую вы не можете модифицировать? Итак, строка, которая не может быть модифицирована, является более простой для операционной системы с точки зрения обработки, так что ваша программа может выполняться быстрее. На самом деле когда вы используете Objective-C для написания своих программ, вы поймёте, что в большинстве случаев вам не требуется модифицировать свои строки.

Конечно же, иногда нужны изменяемые строки. Поэтому существует другой класс, и строковые объекты, созданные на основе него модифицируемы. Это класс `NSMutableString`. Позднее мы обсудим его в этой главе.

Давайте сначала убедимся, что вы поняли, что такое объекты типа `string`. Т.к. они являются объектами, мы можем посылать им сообщения. Например, мы можем послать длину сообщения в объект `string`.

```
//[74]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  int theLength;
  NSString * foo;
  foo = @"Юлия!";
  theLength = [foo length]; // [74.10]
  NSLog(@"Количество символов: %d.", theLength);
  [pool release];
  return 0;
}
```

Когда запустим, программа выведет:

```
Количество символов: 4
```

Программисты часто используют в качестве имен переменных `foo` *(с англ. нечто, прим. переводчика) и* `bar` *для объяснения каких-либо понятий. Конечно же это плохие имена переменных, т.к. они не описывают предназначение, наподобие имени переменной* `x`. *Мы обращаем на это внимание, чтобы вы не были озадачены, когда вы увидите такие названия переменных на форумах в интернете.*

В строке [74.10] мы посылаем объекту `foo` длину сообщения. Метод `length` определен в классе `NSString` следующим образом:

```
- (unsigned int)length
```

Он возвращает количество символов юникод.

Вы также можете изменить символы строки в верхнем регистре [75]. Для этого отправьте в объект строки соответствующее сообщение, например `uppercaseString`, документацию по нему вы можете найти сами (смотрите методы, имеющиеся в `NSString` класс). После получения данного сообщения, объект строки создает и возвращает новый объект строки, имеющей то же содержание, причем каждый символ изменен на верхний регистр.

```
//[75]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSString *foo, *bar;
  foo = @"Юлия!";
  bar = [foo uppercaseString];
  NSLog(@"%@ изменено на %@.", foo, bar);
  [pool release];
  return 0;
}
```

После запуска программа выведет следующее:

```
Юлия! изменено на ЮЛИЯ!
```

Иногда вы можете захотеть изменить содержание имеющихся строки вместо создания новой. В таком случае вам прийдется использовать объект класса `NSMutableString` для представления вашей строки. `NSMutableString` предусматривает несколько методов, которые позволяют изменять содержимое строки. Например, метод `appendString:` дописывает строку в качестве аргумента передаваемого в получателя.

```
//[76]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableString *foo; // [76.7]
  foo = [@"Юлия!" mutableCopy]; // [76.8]
  [foo appendString:@" Я счастлив."];
  NSLog(@"Результат: %@.", foo);
  [pool release];
  return 0;
}
```

Во время выполнения программа напечатает:

```
Результат: Юлия! Я счастлив.
```

В строке [76.8], метод `mutableCopy` (который обеспечивает `NSString` класс) создает и возвращает непостоянную строку с тем же содержанием, как приемник. То есть, после исполнения этой линии [76,8], `foo` указывает на объект непостоянной строки , который содержит строку «Юлия!».

Еще об указателях.

Ранее в этой главе мы отмечали, что в Objective-C, объекты никогда не используются напрямую, но всегда через указатели на них. И вот почему, например, мы используем указатель нотации в строке [76.7] выше. На самом деле, когда мы используем слово «объект»

в Objective-C, мы, как правило, имеем ввиду «указатель на объект». Но поскольку мы всегда использовали объекты через указатели, мы используем слово «объект» как ярлык. Тот факт, что объекты всегда используются через указатели имеет важные последствия, вы должны понять: несколько переменных могут быть ссылкой один и тот же объект одновременно. Например, после исполнения линии [76,8], переменная `foo` ссылается на объект, представляющий строку «Юлия!».

Объекты всегда управляются указателями. Теперь предположим, что мы присваиваем значение `foo` переменной `bar`, вот так:

```
bar = foo;
```

В результате, `foo` и `bar` теперь указывают на один и тот же объект. Множество переменных могут ссылаться на один объект.

В такой ситуации, послав сообщение в объект используя `foo`, как получателя (`[foo dosomething];`) имеем тот же эффект, как в случае отправки сообщения, используя `bar` (`[bar dosomething];`):

```
//[77]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableString *foo = [@"Юлия!" mutableCopy];
  NSMutableString *bar = foo;
  NSLog(@"foo указывает на строку: %@.", foo);
  NSLog(@"bar указывает на строку: %@.", bar);
  NSLog(@"\n");
  [foo appendString:@" Я счастлив."];
  NSLog(@"foo указывает на строку: %@.", foo);
  NSLog(@"bar указывает на строку: %@.", bar);
  [pool release];
  return 0;
}
```

После исполнения программа напечатает:

```
foo указывает на строку: Юлия!
bar указывает на строку: Юлия!
foo указывает на строку: Юлия! Я счастлив.
bar указывает на строку: Юлия! Я счастлив.
```

Возможность иметь ссылки на один объект одновременно из разных мест программы — неотъемлемая особенность объектно-ориентированных языков. Более того, мы уже пользовались этим в предыдущих главах. Например, в Главе 8, мы ссылались на наш объект `MAFoo` из двух разных кнопок.

Глава 13: Массивы

Иногда вам необходимо работать с наборами данных. Например, у вас есть список строк. Было бы довольно обременительно оперировать с ним, используя переменную для каждой из строк. Конечно же есть более подходящее решение — это массивы.

Массив — это упорядоченный список объектов (если быть более точным, список указателей на объекты). Вы можете добавлять объекты в массив, удалять их, запрашивать объекты по индексу. Также вы можете узнать сколько элементов содержит массив.

Когда вы считаете, обычно начинаете с единицы. Однако, в массивах, первый элемент является нулевым, второй элемент имеет индекс 1 и так далее.

Мы приведем пример кода позже в этой главе. Этот код наглядно покажет почему лучше считать с нуля.

Функциональность массивов заложена в двух классах: `NSArray` и `NSMutableArray`. Также как и со строками, существует постоянная и переменная (изменчивая) версии. В этой главе мы будем обсуждать переменную версию.

> *Эти массивы специфичны для Objective-C и Cocoa. Существует другой, облегчённый тип массива в языке C (который именно поэтому является частью Objective-C), но мы не будем его здесь обсуждать. Это просто напоминание о том, что попозже вы можете почитать про массивы языка C в дополнительных источниках, чтобы удостовериться в том, что они не имеют непосредственного отношения к* `NSArrays` *или* `NSMutableArrays`.

Метод класса

Один из способов создать массив является использование следующего выражения:

```
[NSMutableArray array];
```

Этот код создает и возвращает новый массив. Но... минуточку... этот код немного странный, не так ли?

Действительно, в данном случае мы использовали имя класса `NSMutableArray` в качестве приёмника сообщения. Но ведь до сих пор мы посылали сообщения экземплярам, но не классам, верно?

Так вот, мы узнали кое-что новое: в Objective-C мы можем посылать сообщения в том числе и классам (потому что классы — это тоже объекты, экземпляры того, что мы называем мета-классами; впрочем, эта тема выходит за рамки нашей книги).

Следует заметить, что этот объект автоматически помечается как `autoreleased` при создании, т.е. добавляется в `NSAutoreleasePool` и будет уничтожен методом класса, который его создал. Вызов этого метода класса эквивалентен следующему:

```
NSMutableArray *array = [[[NSMutableArray alloc] init] autorelease];
```

В случае, если вы желаете, чтобы массив существовал дольше, чем отводится элементам авторелиз-пула, вы должны послать сообщение `retain`.

В документации Cocoa методы классов обозначены префиксом «+» вместо «–», который мы видим обычно перед именами методов экземпляров (см. примеры в Главе 8 [58.5]). Например, в документации мы видим такое описание метода array:

array

```
+ (id)array
```

Создает и возвращает пустой массив. Этот метод используется изменяемыми дочерними классами класса NSArray. Так же посмотрите: + arrayWithObject:, + arrayWithObjects:

Вернемся к написанию кода. Следующая программа создает пустой массив, сохраняет в нем три строки, а затем выводит количество элементов в массиве.

```
//[78]
#import <foundation/foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableArray *myArray = [NSMutableArray array];
  [myArray addObject:@"первая строка"];
  [myArray addObject:@"вторая строка"];
  [myArray addObject:@"третья строка"];
  int count = [myArray count];
  NSLog(@"В массиве %d элемента", count);
  [pool release];
  return 0;
}
```

После запуска программа выведет:

```
В массиве 3 элемента
```

Следующая программа похожа на предыдущую за одним исключением — она выведет строку с индексом 0 из массива. Для получения этой строки используется метод objectAtIndex [79.11].

```
//[79]
#import <foundation/foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableArray *myArray = [NSMutableArray array];
  [myArray addObject:@"первая строка"];
  [myArray addObject:@"вторая строка"];
  [myArray addObject:@"третья строка"];
  NSString *element = [myArray objectAtIndex:0]; // [79.11]
  NSLog(@"Элемент массива по индексу 0: %@", element);
  [pool release];
  return 0;
}
```

После запуска программа выведет:

```
Элемент массива по индексу 0: первая строка
```

Часто нужно пройтись по массиву, чтобы сделать что-либо с каждым его элементом. Чтобы сделать это можно использовать конструкцию цикла, как это сделано в примере, который выводит каждый элемент массива вместе с его индексом.

```
//[80]
#import <foundation/foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableArray *myArray = [NSMutableArray array];
  [myArray addObject:@"первая строка"];
  [myArray addObject:@"вторая строка"];
  [myArray addObject:@"третья строка"];

  int i;
  int count;
  for (i = 0, count = [myArray count]; i < count; i = i + 1)
  {
    NSString *element = [myArray objectAtIndex:i];
    NSLog(@"Элемент массива по индексу %d: %@", i, element);
  }
  [pool release];
  return 0;
}
```

После запуска программа выведет:

```
Элемент массива по индексу 0: первая строка
Элемент массива по индексу 1: вторая строка
Элемент массива по индексу 2: третья строка
```

Обратите внимание, что массивы не обязаны содержать только строки. Они могут хранить любые объекты.

Классы `NSArray` и `NSMutableArray` предоставляют много других методов, рекомендуем посмотреть документацию о этих классах, чтобы получить дополнительную информацию о массивах. В конце этой части поговорим о методе, который позволит заменить объект с указанным индексом другим объектом. Он называется `replaceObjectAtIndex:withObject:`.

До сих пор мы рассматривали методы, которые принимают самое большее один аргумент. Этот отличается, по этой причине мы его сейчас и рассмотрим: он принимает два аргумента. Это происходит потому, что его название состоит из двух двоеточий. В Objective-C методы могут иметь любое число аргументов. Вот как вы можете использовать этот метод:

```
//[81]
[myArray replaceObjectAtIndex:1 withObject:@"Привет"];
```

После выполнения этого метода, объект под индексом 1 является строкой «Привет». Естественно, этот метод нужно вызывать только с уже существующим индексом. То есть, по

индексу, который мы передаем методу, уже должен находиться объект. Это необходимо, чтобы метод мог заменить этот объект на тот, что мы ему передаем.

Как вы можете видеть, методы в Objective-C, как фразы с пустотами в них (с двоеточим впереди). Когда вы вызываете метод вам нужно заполнить пустоты фактическими значениями, создавая значимые «фразы». Этот способ, определения и вызова метода пришли к нам из языка программирования Smalltalk и является одним из величайших преимуществ Objective-C, так как это делает код очень выразительным. Когда вы создаете свой собственный метод, вы должны стремиться называть их таким образом, чтобы они образовывали выразительные фразы, во время вызова. Это помогает сделать код Objective-C удобным для чтения, что имеет очень важное значение в простоте развития ваших программ.

Глава 14: Методы доступа и свойства

Мы видели, что объект может быть видимым, как окно или текстовое поле; или невидимым, как массив или контролер, который реагирует на действия с пользовательским интерфейсом. Итак, что же такое объект?

В сущности, объект имеет определенные значения (переменные или свойства) и выполняет определенные действия (методы). Объект и содержит, и преобразует данные. Объект можно рассматривать как небольшой компьютер сам по себе, который посылает и отвечает на сообщения. Ваша программа представляет собой сеть из этих небольших компьютеров, работающих вместе для получения желаемого результата.

Состав объекта

Работой программиста Cocoa является создание классов, которые содержат целый ряд других объектов (таких как, например, строки, массивы и словари) для установки значений. Некоторые из этих объектов будут созданы, использованы и отброшены в одном методе. Другие, возможно, необходимо сохранять на все время существования объекта. Эти объекты известны как переменные или свойства класса. Класс может также определять методы, которые работают на этих переменных.

Этот метод известен как объект композиции. Такие объекты, как правило, наследуются непосредственно от `NSObject`.

Например, контролер класса `calculator` может содержать такие переменные, как массив объектов кнопок и текстовое поле для результата. Он также может включать методы умножения, сложения, вычитания, деления чисел и отображения результата в GUI.

Переменные объявляются в заголовке интерфейса для данного класса. На примере нашего калькулятора контроллер класса может выглядеть так:

```
//[82]
@interface MyCalculatorController : NSObject
{
  //Поля данных
  NSArray * buttons;
  NSTextField * resultField;
}
//Методы
- (NSNumber *)mutiply:(NSNumber *)value;
- (NSNumber *)add:(NSNumber *)value;
- (NSNumber *)subtract:(NSNumber *)value;
- (NSNumber *)divide:(NSNumber *)value;
@end
```

Инкапсуляция

Одной из целей ООП является инкапсуляция: создание каждого класса самостоятельным и повторно используемым. И, как вы помните из Главы 3, переменные защищены от внешних петель (loop), функций и методов. Это означает, что другие объекты не могут получить доступ к переменным внутри объекта, они доступны только своим методам.

Очевидно, что время от времени другие объекты должны будут изменять данные, содержащиеся в объекте. Как?

Методы доступны вне объекта. Напомним, что все мы должны сделать, это отправить сообщение на наш объект для вызова этого метода. Таким образом чтобы сделать переменную доступной, нужно создать пару методов для доступа и изменения к этой переменной. Эти методы в совокупности называются акксессорами (см. http://ru.wikipedia.org/wiki/Свойство_(программирование)).

В Главе 8 мы обнаружили метод `setIntValue:` для `NSTextField` Этот метод является партнером метода `intValue`. Эти методы являются акксессорами для `NSTextField`.

Аксессоры

Итак, как же это выглядит в коде? Рассмотрим следующий пример.

```
//[83]
@interface MyDog : NSObject
{
  NSString * _name;   //[83.4]
}
- (NSString *)name;
- (void)setName:(NSString *)value;
@end
```

Этот интерфейс определяет объект `MyDog`. `MyDog` имеет одну переменную: строка, названная `_name` [83.4]. Для того чтобы иметь возможность читать или изменить `_name`, мы определили два акксессора: `name` и `setName:`.

Пока все хорошо. Реализация выглядит подобно этому:

```
//[84]
@implementation MyDog
- (NSString *)name
{
  return _name;  //[84.5]
}
- (void)setName:(NSString *)value
{
  _name = value;  //[84.9]
}
@end
```

В первом методе [84.5] мы просто возвращаем переменную. Во втором методе [84.9] мы устанавливаем ее в полученное значение. Заметьте, что я уже упростил эту процедуру для

ясности; обычно вам необходимо управлять памятью в пределах этих методов. В приведенном ниже примере показан более реалистичный набор средств доступа:

```
//[85]
@implementation MyDog
- (NSString *)name
{
  return [[_name retain] autorelease];
}
- (void)setName:(NSString *)value
{
  if (_name != value)
  {
    [_name release];
    _name = [value copy];
  }
}
@end
```

Я не буду вдаваться в подробности по поводу дополнительного кода (см. Главу 15), но вы можете видеть, что он выглядит также, как и [84], лишь с некоторыми операциями копирования, сохранения и высвобождения, обернутыми вокруг него. Разные типы значений требуют различный код управления памятью.

> *Обратите внимание, что на практике рекомендуется не использовать подчеркивания перед именем переменной, я использовал его здесь для ясности. В своем коде вы могли бы просто назвать переменную «name». Поскольку методы и переменные имеют раздельные пространства имен, никаких конфликтов будет не возникать.*

Свойства

Mac OS X Leopard и Objective-C 2.0 внедряют новые особенности языка для контакта с общим шаблоном программирования более экономно. Новой особенностью, о которой мы говорим, является суммирование свойств. Акксессоры настолько распространены, что эта долгожданная поддержка языкового уровня может привести к значительно меньшему количеству кода. И меньше кода означает меньше кода для отладки.

Так чем свойства отличаются от аксессоров? По существу свойство создает акксессор напрямую, используя наиболее эффективное и правильное управление памятью. Иными словами, они пишут акксессоры для вас, но в фоновом режиме, чтобы вы никогда не увидели код.

Используя наш пример [83] выше, в Objective-C 2.0 мы могли бы вместо этого написать:

```
//[86]
@interface MyDog : NSObject
{
  NSString * name;
}
@property (copy) NSString *name;
@end
```

И наша реализация будет выглядеть так:

```
    //[87]
    @implementation MyDog
    @synthesize name;
    @end
```

Это логически эквивалентно [85]. Как вы видите, это еще немного упростило наш код. Если ваш класс имеет много полей данных, требующий акксессоров, то вы можете себе представить, насколько легче стала ваша жизнь!

Глава 15: Работа с памятью

Во многих главах я извинялся, что не объяснил пару утверждений на примерах. Это касается работы с памятью. Ваша программа не является единственной программой на вашем Mac, и оперативно запоминающее устройство (ОЗУ или память) является ценным ресурсом. Если вашей программе больше не нужна часть памяти, вы должны освободить ее (память), отдать в распоряжение системы. Когда ваша мама сказала вам, что вы должны быть вежливы и жить в гармонии с обществом, она учила вас как программировать! Даже если ваша программа единственная сейчас работает, заполнение памяти может привести к тому, что ваш Mac начнет тормозить.

Сборка мусора

Средства управления памятью, используемые в Cocoa и описываемые далее в этой главе, известны под обобщенным названием механизма подсчета ссылок. Подробное описание этого механизма вы найдете в специализированных книгах и статьях.

Mac OS X 10.5 Leopard добавляет в Objective-C 2.0 новый вид управления памятью: сборку мусора (Cocoa Garbage Collection). Сборщик мусора управляет памятью автоматически, избавляя программиста от необходимости явно сохранять и освобождать Cocoa-объекты.

Средство сборки мусора работает со всеми Cocoa объектами, унаследованными от объекта `NSObject` или `NSProxy`. Оно позволяет программисту писать меньше кода, в отличие от более старых версий Objective-C. Больше теоретических выкладок не будет – дальше практика. Забудьте обо всем, что читали в этой главе!

Включение сборки мусора

По умолчанию в `Xcode` сборка мусора выключена. Чтобы ее включить выберите целевое приложение из списка исходных кодов и откройте Inspector (Tools ▸ Inspector ⇧⌘I). Поставьте галочку на опции «Enable Objective-C Garbage Collection». Заметьте, что любой фреймворк, на который вы ссылаетесь в вашем проекте, обязательно должен быть так же с включенной опцией сборки мусора.

Подсчёт ссылок: жизненный цикл объекта

Этот раздел специально для тех, кому интересны механизмы управления памятью до Mac OS X Leopard.

Когда вы создаете в программе какой-либо объект, этот объект занимает некоторый объем памяти. Когда объект перестает быть нужным, вам необходимо освободить занимаемую память. Однако, перед тем как удалить объект, нужно определить, используется ли еще этот объект где-либо еще, что не так то просто.

Например, во время выполнения программы, на ваш объект может ссылаться множество других объектов, и по этой причине, он не может быть удален до тех пор, пока он используется другими объектами. Использование объекта, который уже удален из памяти

может быть причиной непредсказуемого поведения программы или даже завершения с ошибкой.

Число ссылок на объект

Для того, чтобы помочь вам освобождать память из под объектов, которые вам больше не нужны, Cocoa создает у каждого объекта счетчик, который называется «число ссылок на объект» (retain count). Когда вы сохраняете ссылку на объект в программе, вы должны позволить объекту знать об этом путем увеличения счетчика на единицу. Когда вы убираете ссылку на объект, вы должны, соответственно, уменьшить значение счетчика. Когда значение счетчика становиться равным нулю, объект «понимает», что на него ссылок больше нет и он может быть удален из памяти.

Например, предположим, что ваше приложение — это музыкальный плеер и у вас есть объекты, представляющие собой песни и плейлисты. Предположим, что на один объект «песня» ссылается три объекта «плейлист». Если на него (объект «песня») больше никакой другой объект не ссылается, ваша песня имеет число ссылающихся объектов равное трем.

Объект знает сколько раз на него ссылаются благодаря счетчику.

Сохранить и Освободить

Для того, чтобы увеличить счетчик ссылающихся объектов, вам нужно всего лишь послать сообщение:

```
    [anObject retain];
```

Чтобы уменьшить значение, нужно послать сообщение об освобождении объекта:

```
    [anObject release];
```

Автоматическое освобождение

Также Cocoa предлагает механизм, называемый «Autorelease pool», который позволяет посылать отложенное сообщение об освобождении объекта. Т.е. это сообщение будет послано не в момент отправки, а через некоторое время. Чтобы использовать данную функциональность, вы должны зарегистрировать объект в пуле автоматически освобождаемых объектов (Autorelease pool), послав сообщение следующим образом:

```
    [anObject autorelease];
```

«Autorelease pool» позаботится о направлении отложенного сообщение на ваш объект. Выражения, работающие с «Autorelease pool», которые мы наблюдали ранее в наших программах, это инструкции для системы, чтобы правильно создать «Autorelease pool» настройки.

Глава 16: Источники информации

Скоромная цель этой книги состоит в том, чтобы научить вас основам Objective-C в среде Xcode. Если вы прочли эту книгу дважды, и сделали примеры с вашими собственными вариациями, то вы готовы учиться, как писать «Killer apps», которые вам хочется создать. Эта книга дает вам достаточно знаний для быстрого погружения. Дочитав до этого места, вы будете готовы использовать другие ресурсы. Одна важная рекомендация, прежде чем приступать к написанию кода: Не начинайте прямо сейчас! Проверьте существующие решения, возможно что Apple, уже сделали работу за вас, или при создали классы, которые требуют меньше усилий для получения того, что вам нужно. Кроме того, кто-то другой, мог уже сделать то, что вам нужно, и сделал исходный код доступным. Таким образом вы сэкономите время, просмотрев документы и поискав в Интернете. Ваш первый визит должен быть на сайт Apple в раздел для разработчиков, который располагается по адресу: http://developer.apple.com

Настоятельно рекомендуем следующие источники:

- http://osx.hyperjeff.net/reference/CocoaArticles.php
- http://www.cocoadev.com
- http://www.cocoadevcentral.com
- http://www.cocoabuilder.com
- http://www.stepwise.com

Перечисленные выше сайты содержат большое количество ссылок на другие полезные сайты и источники информации. Также советуем вам подписаться на список рассылок cocoa-dev. Сделать это вы можете на http://lists.apple.com/mailman/listinfo/cocoa-dev. Кстати, в списках рассылок вы можете задать и ваши вопросы. Но, не смотря на то, что другие пользователи будут рады помочь вам, будьте учтивы и попробуйте сперва найти ответ в архивах (http://www.cocoabuilder.com). Некоторые советы и рекомендации по размещению вопросов в списках рассылки, вы можете прочитать в статье «How To Ask Questions The Smart Way», расположенной по адресу http://www.catb.org/~esr/faqs/smart-questions.html.

Существует несколько очень хороших книг о Cocoa. «Programming in Objective-C» Стефана Кохана (Stephen Kochan) рассчитана на новичков. Некоторые книги предполагают у читателя наличие знаний, почерпнутых в книге Кохана.

Мы наслаждались книгой «Cocoa Programming for Mac OS X» Aaron Hillegass из Big Nerd Ranch, где он обучает Xcode. Мы также наслаждались Cocoa with Objective-C, написанной James Duncan Davidson и Apple, опубликованной O'Reilly.

Наконец, теплые слова предостережения. Вы программируете для Mac. Прежде чем вы выпустите свою программу, убедитесь, что в ней нет ошибок и она выглядит великолепно и придерживается рекомендаций построения интерфейса пользователя Apple. Как только вы сделаете это, не стесняйтесь, делая свою программу доступной! Комментарии от других помогут вам совершенствовать и развивать свою програ мму и уделять внимание качеству.

Мы надеемся вам понравилась эта книга и вы продолжите программировать с помощью Xcode.

Bert, Alex, Philippe.