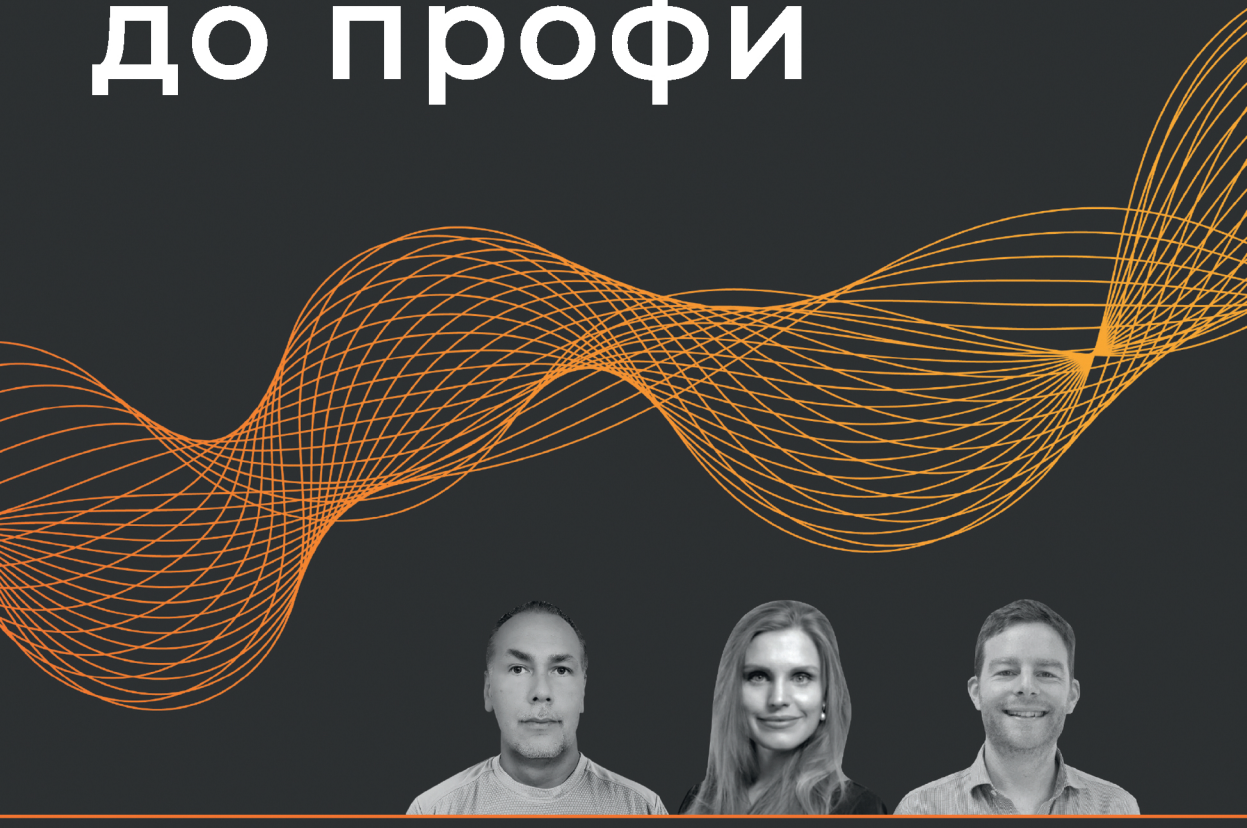
EXPERT INSIGHT

JavaScript с нуля до профи



Лоренс Ларс Свекис
Майке ван Путтен
Роб Персиваль

Packt

JavaScript с нуля до профи

Лоренс Ларс Свекис

Майке ван Путтен

Роб Персиваль

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.988.02-018.1
УДК 004.738.5
С24

Свекис Лоренс Ларс, Путтен Майке ван, Персиваль Роб

С24 JavaScript с нуля до профи. — СПб.: Питер, 2023. — 480 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2269-1

Книга демонстрирует возможности JavaScript для разработки веб-приложений, сочетая теорию с упражнениями и интересными проектами. Она показывает, как простые методы JavaScript могут применяться для создания веб-приложений, начиная от динамических веб-сайтов и заканчивая простыми браузерными играми.

В «JavaScript с нуля до профи» основное внимание уделяется ключевым концепциям программирования и манипуляциям с объектной моделью документа для решения распространенных проблем в профессиональных веб-приложениях. К ним относятся проверка данных, управление внешним видом веб-страниц и работа с асинхронным и многопоточным кодом.

Обучайтесь на основе проектов, дополняющих теоретические блоки и серии примеров кода, которые могут быть использованы в качестве модулей различных приложений, таких как валидаторы входных данных, игры и простые анимации. Обучение дополнено ускоренным курсом по HTML и CSS, чтобы проиллюстрировать, как компоненты JavaScript вписываются в полноценное веб-приложение.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.1
УДК 004.738.5

ISBN 978-1800562523 англ.

ISBN 978-5-4461-2269-1

Краткое содержание

Оглавление

Об авторах

Лоренс Ларс Свекис является экспертом в области инновационных технологий, имеет широкий спектр теоретических знаний и реальный опыт в области веб-разработки. Начиная с 1999 года участвует в различных веб-проектах, как крупных, так и небольших. Преподает с 2015 года и обожает воплощать идеи в онлайн-жизнь. Для Лоренса учить и помогать людям — потрясающая возможность, поскольку ему нравится делиться знаниями с обществом.

Алексис и Себастьян, большое спасибо за вашу поддержку.

Майке ван Путтен известна страстью к обучению и разработке программного обеспечения, стремлением сопровождать людей на их пути к новым высотам в карьере. К числу ее любимых языков относятся JavaScript, Java и Python. Как разработчик участвует в проектах по созданию программного обеспечения и как тренер — в различных сферах, начиная с IT для «чайников» и заканчивая продвинутыми темами для опытных специалистов. Любит создавать образовательный онлайн-контент на различных платформах, предназначенный для широкой аудитории.

Роб Персиваль — уважаемый веб-разработчик и преподаватель Udemy с аудиторией более 1,7 миллиона учеников. Более 500 000 из них приобрели его «Полный курс веб-разработчика 2.0», а также курсы разработчиков Android и iOS.

О научном редакторе

Крис Минник — активный автор, блогер, тренер, спикер и веб-разработчик. Его компания WatzThis? занимается поиском лучших способов обучения новичков навыкам обращения с компьютером и программированию.

Крис более 25 лет трудится full-stack-разработчиком и более десяти лет — преподавателем. Обучает веб-разработке, ReactJS и продвинутому JavaScript во многих крупнейших мировых компаниях, а также в публичных библиотеках, коворкингах и на личных встречах.

Крис Минник — автор и соавтор более десятка технических изданий для взрослых и детей, включая *React JS Foundations*, *HTML and CSS for Dummies* («HTML и CSS для чайников»), *Coding with JavaScript for Dummies* («JavaScript для чайников»), *JavaScript for Kids*, *Adventures in Coding* и *Writing Computer Code*.

Предисловие

JavaScript — удивительный мультифункциональный язык, широко используемый, кроме всего прочего, в веб-разработке. Любое действие на веб-странице — это работа JavaScript. Фактически все современные браузеры понимают JavaScript — а в скором времени поймете его и вы.

В данной книге есть все, что вам нужно знать для разработки приложений на JavaScript и проектирования веб-страниц с его использованием. После ее прочтения вы сможете создавать интерактивные веб-страницы, динамические приложения и многое другое.

Для кого эта книга

Для комфортного знакомства с книгой не требуется никакого опыта в JavaScript. Конечно, упражнения дадутся немного легче, если вы хотя бы немного умеете программировать. Знакомство с основами HTML и CSS будет вашим преимуществом. Если вы начинающий программист, для нас большая честь поприветствовать вас в мире программирования. Вначале он может показаться сложным, но мы проведем вас через все трудности.

Структура издания

Глава 1 «Начало работы с JavaScript» знакомит с базовыми сведениями по JavaScript, которые вы должны знать, чтобы понять оставшуюся часть книги.

Глава 2 «Основы JavaScript» описывает основные элементы кода: переменные, типы данных и операторы.

Глава 3 «Множественные значения JavaScript» затрагивает вопросы хранения множественных величин в одной переменной с использованием массивов и объектов.

В *главе 4 «Логические операторы»* начнется настоящее веселье: мы будем использовать логические операторы, чтобы они принимали решения за нас!

Глава 5 «Циклы» описывает ситуации, когда необходимо повторить фрагмент кода (для чего циклы и используются). Мы рассмотрим различные типы циклов, в том числе циклы `for` и `while`.

Глава 6 «Функции» вводит очень полезные объекты для работы с повторяющимися фрагментами кода: функции! Они позволяют вызывать определенный блок кода в любом месте нашего скрипта для выполнения какой-то задачи, благодаря чему мы не будем дублировать написанное, соблюдая таким образом один из фундаментальных принципов чистого кода.

В *главе 7 «Классы»* мы продолжим строить блоки JavaScript, которые помогут лучше структурировать приложение. Зная к тому моменту, как создавать объекты, вы научитесь работать с шаблонами объектов, которые, когда понадобится конкретный тип объекта, можно использовать повторно.

Глава 8 «Встроенные методы JavaScript» знакомит с некоторым встроенным функционалом. Конечно, можно писать функции и самостоятельно; но на практике куда чаще используются встроенные функции JavaScript, особенно в таких общих задачах, как проверка, имеем ли мы дело с числом или нет.

Глава 9 «Объектная модель документа» рассматривает такие понятия, как объектная модель браузера и объектная модель документа (DOM). Благодаря им вы сильно обогатите свои методы использования JavaScript. Вы узнаете, что такое DOM, как с ней взаимодействует JavaScript и как благодаря этому можно изменять веб-сайты.

Глава 10 «Управление динамическими элементами с помощью DOM» показывает, как динамически управлять элементами DOM и создавать таким образом современные пользовательские интерфейсы. Вы сможете изменить сайт в ответ на поведение пользователей, например на нажатие кнопки.

Глава 11 «Интерактивный контент и прослушиватели событий» поднимет качество вашего отклика на желания пользователей на новый уровень. Например, вы научитесь реагировать на такие события, как выход курсора за пределы поля ввода и движение мыши.

Глава 12 «Средний уровень JavaScript» касается задач, для решения которых понадобится средний уровень знания JavaScript. Это, например, регулярные выражения, рекурсия и отладка, необходимые для повышения производительности программного кода.

Глава 13 «Параллелизм» познакомит с поточным и асинхронным программированием, которое позволит вашему коду выполнять несколько задач одновременно и действительно быть гибким.

Глава 14 «HTML, Canvas и JavaScript» посвящена HTML5 и JavaScript. Предыдущие главы дают довольно много информации о HTML и JavaScript; здесь же мы сосредоточимся именно на HTML5, в частности на таких его особенностях, как элемент `canvas`.

Глава 15 «Дальнейшие шаги» расскажет, куда можно двигаться после изучения всех фундаментальных возможностей JavaScript и приобретения навыка создания изящных программ с помощью JavaScript. Мы познакомимся с такими знаменитыми библиотеками и фреймворками для JavaScript, как Angular, React и Vue, а также рассмотрим Node.js и узнаем, как бэкенд может быть реализован в JavaScript.

Как извлечь максимальную пользу из книги

Предыдущий опыт программирования для вас будет полезен, но он точно не обязателен. Чтобы начать работу с данной книгой, достаточно компьютера с текстовым редактором (Notepad или TextEdit, не Word!) и браузером. Выполняйте упражнения и проекты, постоянно экспериментируйте — так вы будете уверены, что разобрались в теме и сможете идти дальше.

Файлы с примерами программного кода

Файлы с примерами для книги размещены в GitHub по адресу https://github.com/PacktPublishing/JavaScript-from-Beginner-to-Professional. Изучите их!

Цветные изображения

Мы также предоставляем PDF-файл с цветными оригинальными изображениями скриншотов/диаграмм, использованных в книге. Вы можете найти файл по адресу https://static.packt-cdn.com/downloads/9781800562523_ColorImages.pdf.

Условные обозначения

Моноширинным шрифтом написаны фрагменты кода в тексте, имена таблиц баз данных, имена папок и файлов, расширения файлов, имена путей, макеты URL, пользовательский ввод, имена пользователей Twitter. Например: «Нам также необходимо сообщить браузеру, с каким типом документа мы работаем, объявив `<!DOCTYPE>`».

Фрагмент кода выглядит следующим образом:

```
<html>
  <script type="text/javascript">
    alert("Hi there!");
  </script>
</html>
```

Любой ввод или вывод командной строки указывается так:

```
console.log("Hello world!")
```

Курсивом выделяется новый термин или важное слово. Рубленым шрифтом выделены слова, отображаемые на экране (выбранные пункты в меню или диалоговых окнах также будут выделены таким шрифтом). Например: «Если вы щелкнете правой кнопкой мыши и выберете пункт Inspect (Проверка) в системах macOS, на экране появится изображение, представленное на одном из следующих снимков экрана».

Предупреждения или важные сообщения выглядят так.

Советы и рекомендации выглядят так.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Начало работы с JavaScript

Итак, вы решили начать изучать JavaScript — отличный выбор! JavaScript — язык программирования, который может применяться и в серверной, и в клиентской части приложения. Серверная часть приложения — это внутренняя логика, которая обычно выполняется на компьютерах в центрах обработки данных и взаимодействует с базой данных, в то время как клиентская часть запускается на устройстве пользователя, часто с использованием браузера для JavaScript.

Вполне вероятно, что вы уже пользовались функциями, написанными на JavaScript, — особенно если работали в таких браузерах, как Chrome, Firefox, Safari или Edge. JavaScript чрезвычайно распространен. После открытия страницы вам предлагается принять cookie-файлы. Как только вы нажимаете OK, всплывающее окно исчезает — это результат работы JavaScript. Когда вы перемещаетесь по разделам сайта и открываете подменю, также работает JavaScript. Или когда вы фильтруете товары в каталоге интернет-магазина. А как насчет чатов, которые выскакивают в считаные секунды пребывания на сайте? Что ж, вы правы — это также JavaScript!

Практически любое взаимодействие, которое мы осуществляем с веб-страницами, происходит благодаря JavaScript. Кнопки, которые вы нажимаете, поздравительные открытки, которые создаете, вычисления, которые проводите. Все, чему мало статичной страницы, требует JavaScript.

В этой главе мы затронем следующие темы.

- Почему надо изучать JavaScript.
- Как настроить среду разработки.
- Как браузер понимает JavaScript.
- Как использовать консоль браузера.
- Как добавить JavaScript на веб-страницу.
- Как написать код JavaScript.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Почему надо знать JavaScript

Существует множество аргументов в пользу изучения данного языка. JavaScript (берущий начало в 1995 году) считается одним из наиболее распространенных языков программирования. Причина этому — JavaScript поддерживается и распознается браузерами. Если на вашем компьютере установлены браузер и текстовый редактор — вы уже владеете всем необходимым для работы с JavaScript. Конечно, есть и более продвинутые инструменты — мы рассмотрим их позже в текущей главе.

JavaScript — отличный язык программирования для новичков. Тем не менее самые продвинутые разработчики программного обеспечения также должны его знать хотя бы немного, ведь однажды они неизбежно столкнутся с ним. JavaScript — ваш прекрасный выбор по многим причинам. Прежде всего, вы сможете создавать действительно классные приложения с его помощью быстрее, чем можете себе представить. К тому моменту, как вы доберетесь до главы 5, вам уже будут по зубам довольно сложные скрипты для взаимодействия с пользователями. А к концу книги вы спокойно будете писать динамические веб-страницы для всевозможных задач.

JavaScript используется для разработки приложений и скриптов множества типов. На нем можно писать код для браузеров, а также программировать логический, невидимый нам, слой кода в приложении (такой как связь с базой данных). JavaScript применим в играх, сценариях автоматизации и других продуктах. Он используется в различных стилях программирования — способах структурирования и написания программного кода. Как вы его примените, зависит от назначения разрабатываемого скрипта. JavaScript полезен в различных парадигмах программирования: объектно-ориентированного, функционального и процедурного (если вы никогда раньше не занимались разработкой, то можете не до конца понять перечисленные концепции, но на данном этапе в этом нет особой необходимости).

Обладая основами JavaScript, можно уверенно использовать огромное число библиотек и фреймворков. Они поднимут ваши навыки на новый уровень, облегчат работу и помогут делать больше за меньшее время. Примеры распространенных и эффективных библиотек: React, Vue.js, jQuery, Angular и Node.js (не переживайте, если для вас это пока просто названия: мы кратко рассмотрим их в самом конце книги).

Как мы уже говорили, JavaScript — крайне популярный язык программирования, поэтому в процессе его изучения новичок не столкнется с такими проблемами, для которых не найдется готового решения в интернете. Сообщество JavaScript огромно. Известный форум Stack Overflow хранит множество решений по всем видам проблем кода, а также огромный раздел о JavaScript. Наверняка вы часто будете заглядывать на эту страницу в поисках советов и рекомендаций.

Если JavaScript — ваш первый язык программирования, то вас ждет много приятного. Разработчики программного обеспечения в целом любят помогать. В интернете есть форумы и учебные пособия с ответами практически на любые вопросы. И хоть вам, как начинающему, может быть непросто их понять, продолжайте учиться — и вскоре все станет ясно.

Настройка среды разработки

Есть множество способов настройки среды программирования JavaScript. Скорее всего, ваш компьютер уже оснащен минимальным набором необходимого для работы с JavaScript. Рекомендуем немного упростить свою жизнь и использовать встроенную среду разработки (IDE).

Встроенная среда разработки

Встроенная среда разработки (IDE) — это специальное приложение для написания, запуска и отладки кода. Вы можете открыть его, как любую другую программу. Например, чтобы создать текстовый документ, необходимо запустить редактор, выбрать файл и начать писать. Так же и в программировании: вы открываете IDE и пишете код. Если нужно запустить код, просто нажмите соответствующую кнопку в IDE. Для JavaScript может потребоваться открыть браузер вручную.

Однако IDE способна на большее. Обычно в ней есть функция подсветки синтаксиса: определенные элементы кода выделяются определенным цветом, что позволяет быстрее находить ошибки. Другая полезная особенность — это автодополнение кода: редактор сам предлагает варианты заполнения, доступные в текущем месте кода. Большинство IDE содержат специальные плагины, которые помогают сделать работу интуитивно понятной и добавить дополнительные функции, например горячую перезагрузку (hot reload) в браузере.

Существует множество IDE, и все они отличаются наборами предлагаемых функций. В книге мы используем Visual Studio Code, но это личное предпочтение. Для своей работы вы можете взять и другие IDE: Atom, Sublime Text и WebStorm. Среды разработки продолжают появляться, поэтому, скорее всего, самой популярной на момент прочтения данной книги в этом списке нет. С актуальными предложениями вы можете познакомиться, запустив быстрый поиск в интернете по запросу `JavaScript IDE`. При выборе IDE прежде всего уделите внимание следующим моментам: убедитесь, что среда поддерживает подсветку синтаксиса, отладку и автодополнение кода JavaScript.

Браузер

Вам также понадобится браузер. Большинство браузеров отлично подходят для задач JavaScript, но лучше не использовать Internet Explorer, который не поддерживает его обновленные функции. Два хороших варианта: Chrome и Firefox. Они прекрасно работают с актуальным функционалом JavaScript, к тому же располагают полезными плагинами.

Дополнительные инструменты

В своей работе не проходите мимо дополнительных возможностей, используемых при программировании, — в том числе плагинов браузеров, которые помогут вам с отладкой или упростят просмотр страницы. На начальном этапе вам все это не понадобится; тем не менее уже сейчас берите на заметку инструменты, которые ценят другие разработчики.

Онлайн-редактор

Это хорошее решение в случае, если у вас нет компьютера (возможно, только планшет) или вы не можете ничего на него устанавливать. В частности, для таких ситуаций есть отличные онлайн-редакторы. Мы не даем конкретных названий: онлайн-редакторы быстро развиваются, и предложенный нами список, вероятно, устареет к моменту выхода книги. Сделайте в интернете запрос `online JavaScript IDE` — система выдаст вам множество результатов, где вы сможете начать программировать на JavaScript простым нажатием кнопки.

Как браузер понимает JavaScript

JavaScript является интерпретируемым языком программирования: это значит, компьютер распознает его в процессе работы кода. Некоторые языки перед запуском кода требуют обработки (данный процесс называется компиляцией) — но для JavaScript в этом нет необходимости. Компьютер интерпретирует JavaScript на лету. При этом «движок», понимающий JavaScript, называется интерпретатором.

Веб-страница — это не только JavaScript. Она создается с помощью трех языков: HTML, CSS и JavaScript.

HTML определяет то, что отображается на странице: ее содержимое хранится в нем. Если на странице есть абзац, в HTML-коде это будет прописано. Видим

заголовок — значит, в HTML он тоже есть. И так далее. HTML состоит из элементов, которые называются тегами. Они описывают конкретный компонент страницы. Вот небольшой пример кода страницы с текстом `Hello world!`:

```
<html>
  <body>
    Hello world!
  </body>
</html>
```

Не переживайте, если не работали с HTML ранее: в главе 9 есть ускоренный курс этого языка.

CSS — это шаблон веб-страницы. Размер шрифта, семейство шрифтов и расположение текста на странице задаются с его помощью. Так, например, синий цвет текста — результат работы CSS.

JavaScript — последний кусочек пазла: он определяет, что может делать страница и как она будет взаимодействовать с пользователем или серверной частью.

Работая с JavaScript, вы рано или поздно столкнетесь с термином *ECMAScript*. Это спецификация, или стандартизация, для языка JavaScript. Текущий стандарт — *ECMAScript 6* (также называемый *ES6*). Браузеры используют его для поддержки JavaScript (в дополнение к таким возможностям, как *объектная модель документа (DOM)*, — ее мы рассмотрим позже). Существует множество незначительно различающихся реализаций JavaScript, и ECMAScript можно считать базовой спецификацией, про которую с уверенностью можно сказать, что в конкретную реализацию она будет включена.

Использование консоли браузера

У браузера есть встроенные возможности для просмотра кода на текущей странице — возможно, вы уже имели с ними дело. Если, находясь в браузере, нажать клавишу F12 на компьютере с Windows или щелкнуть правой кнопкой мыши на пункте Inspect (Проверка) в системе macOS, экран приобретет вид, представленный на одном из следующих снимков экрана.

На вашем компьютере итог может несколько отличаться, но, как правило, на macOS результат нажатия Inspect выглядит так, как показано на рис. 1.1.

На снимке экрана вверху видно несколько вкладок. Рассмотрим элементы, которые содержат в себе HTML и CSS (помните такие?). Если вы щелкнете на вкладке

консоли, в нижней части панели найдется место, где можно непосредственно ввести код. Там могут появляться различные предупреждения и сообщения об ошибках — если страница в целом работает, не беспокойтесь о них: подобные уведомления не редкость.

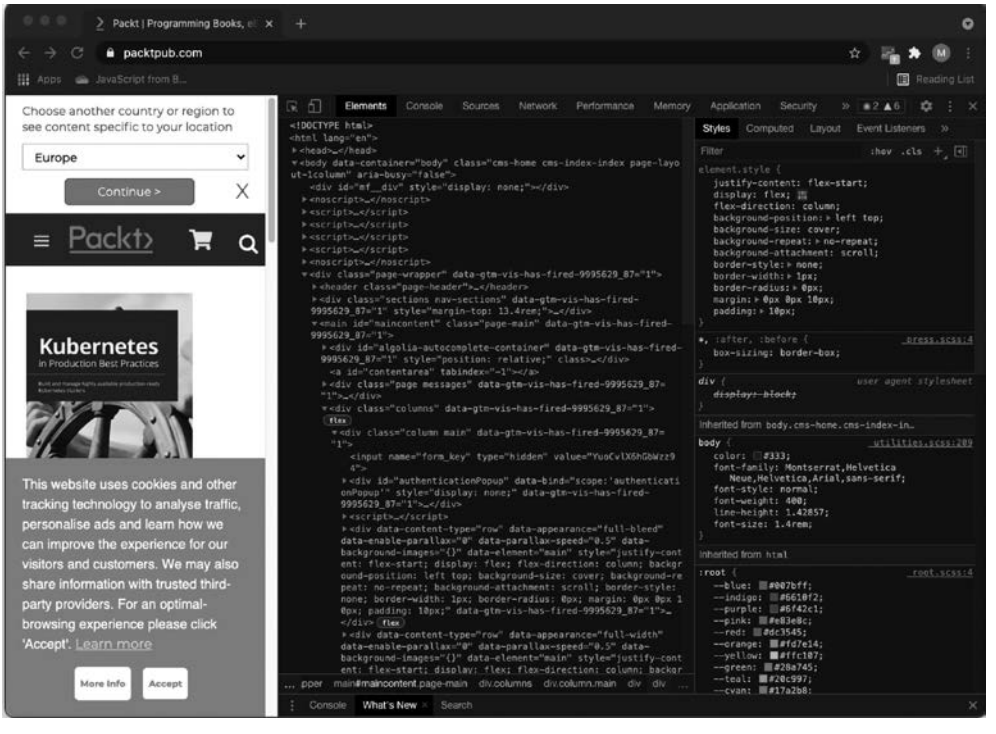


Рис. 1.1. Консоль браузера на сайте Packt

Разработчики используют консоль для логирования происходящего и отладки. Отладка — это поиск проблем в случаях, когда приложение не отображает желаемый результат. Консоль дает некоторое представление о том, что происходит, когда вы вводите осмысленные сообщения. Итак, первая команда, которую мы выучим:

```
console.log("Hello world!");
```

Щелкните на вкладке консоли и введите этот код. После нажатия Enter на экране отобразится вывод вашего кода. Он должен выглядеть как на рис. 1.2.

В книге мы будем часто использовать `console.log()`, чтобы проверять фрагменты кода и просматривать результаты. Существуют и другие консольные

методы. Например, `console.table()` представляет введенные данные в виде таблицы. Еще один консольный метод, `console.error()`, выводит сообщение об ошибке.

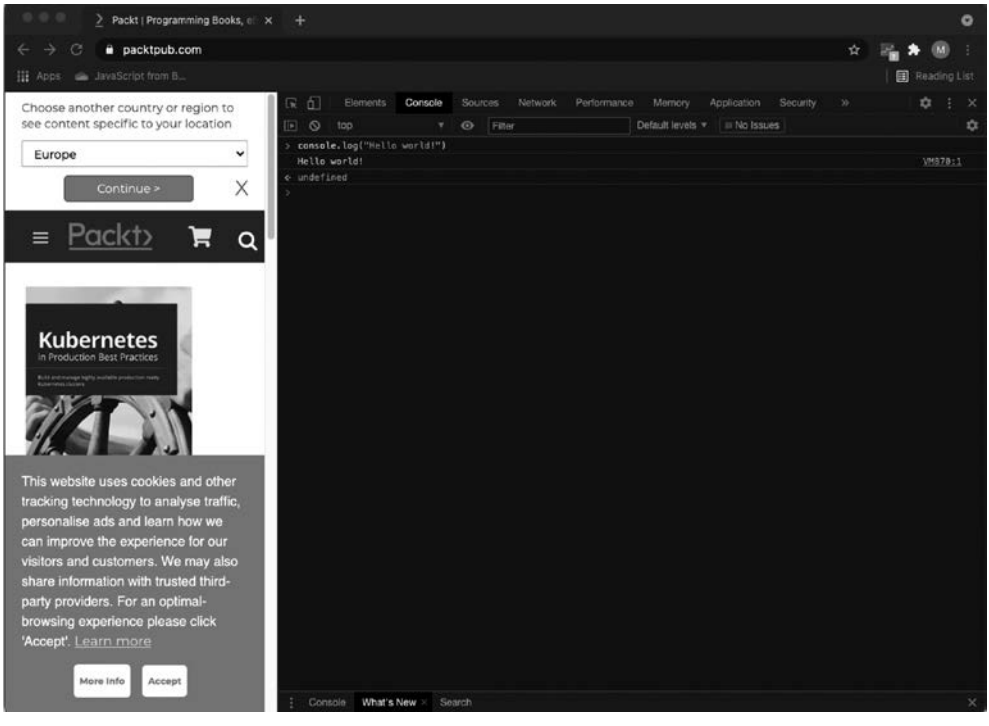


Рис. 1.2. JavaScript в консоли браузера

Практическое занятие 1.1

Работа с консолью

1. Откройте консоль браузера, напишите `4 + 10` и нажмите Enter. Что было представлено в качестве ответа?
2. Введите команду `console.log()`, поместив в круглые скобки какое-нибудь значение. Попробуйте вписать свое имя в кавычках (кавычки нужны для указания факта, что перед нами текстовая строка; мы рассмотрим это подробнее в следующей главе).

Добавление JavaScript на веб-страницу

Существует два способа добавления JavaScript-кода на веб-страницу. Один из них — ввести код прямо в HTML между парными тегами `<script>`. Первый `<script>` объявляет, что в рамках скрипта нужно выполнить следующий код. Далее располагается содержимое скрипта. Затем мы закрываем скрипт тем же тегом `<script>`, только в начале текста между скобками добавляем слеш — `</script>`. Второй способ — подключить файл JavaScript к HTML-файлу, используя соответствующий тег в заголовке страницы HTML.

Непосредственно в HTML

Перед вами пример простой веб-страницы, на которой всплывает окно с надписью `Hi there!`:

```
<html>
  <script type="text/javascript">
    alert("Hi there!");
  </script>
</html>
```

Если сохранить этот код как файл с расширением `.html` и открыть его в браузере, результат будет похожим на представленный на рис. 1.3. Сохраним файл как `Hi.html`.

Рис. 1.3. Код JavaScript реализовал всплывающее окно с текстом Hi there!

Команда `alert` создает всплывающее окно с сообщением. Текст сообщения оформляется в скобках.

Сейчас наш контент располагается непосредственно между тегами `<html>`. Это не лучшее решение. Нужно создать еще два элемента — `<head>` и `<body>`. В `head` мы укажем метаданные; также в данном теге внешние файлы подключаются к HTML-файлу. В `body` поместится содержимое веб-страницы.

Кроме этого, необходимо сообщить браузеру, с каким типом документа мы работаем, объявив `<!DOCTYPE>`. Поскольку мы начали писать код JavaScript в HTML, наполнение тега будет следующим: `<!DOCTYPE html>`. Пример:

```
<!DOCTYPE html>
<html>

<head>
  <title>This goes in the tab of your browser</title>
</head>

<body>
The content of the webpage
  <script>
    console.log("Hi there!");
  </script>
</body>
</html>
```

В результате веб-страница должна отобразить следующее: `The content of the webpage`. Если вы посмотрите в консоль браузера, то обнаружите там сюрприз: исполненный код JavaScript и строку `Hi there!`.

Практическое занятие 1.2

JavaScript на HTML-странице

1. Откройте редактор кода и создайте HTML-файл.
2. Пропишите HTML-теги `doctype`, `html`, `head` и `body` и добавьте скрипт.
3. Между тегами скрипта напишите какой-нибудь JavaScript-код. Можете использовать `console.log("hello world!")`.

Присоединение стороннего файла к веб-странице

Еще один способ добавить JavaScript-код на страницу — подключить к файлу HTML сторонний файл. Данный вариант считается лучшим, поскольку код страницы в целом будет лучше организован и не будет слишком длинным. В дополнение к этим преимуществам вы сможете использовать JavaScript при создании других страниц на сайте, не копируя исходный код. Скажем, все десять страниц вашего сайта содержат один и тот же код JavaScript, в который вам нужно внести изменения. Если вы настроите страницу так, как показано в примере ниже, вам придется изменить только один файл.

Сначала нам понадобится отдельный файл JavaScript (расширение таких файлов — `.js`). Мы назовем его `ch1_alert.js`. Содержимое файла следующее:

```
alert("Saying hi from a different file!");
```

Теперь создадим отдельный файл HTML (с расширением `.html`). И наполним его:

```
<html>
  <script type="text/javascript" src="ch1_alert.js"></script>
</html>
```

Убедитесь, что файлы находятся в одной папке, или укажите точный путь к JavaScript-файлу в HTML-коде. Код чувствителен к регистру, поэтому имена должны полностью совпадать.

Указывая путь к JavaScript-файлу, можно использовать относительный или абсолютный путь. Прежде всего рассмотрим последний, более простой для объяснения. В компьютере есть коренной каталог. Для систем Linux и macOS это обычно `/`, а для Windows — `C:/`. Путь к файлу, который начинается в коренном каталоге, называется абсолютным. Прописывать его проще всего, и на вашем компьютере данный способ наверняка сработает. Но есть одна загвоздка: если папка вашего вебсайта позже будет перемещена с компьютера на сервер, абсолютный путь больше не будет актуальным.

Второй, более безопасный способ — это относительный путь. Вы указываете, как добраться к JavaScript-файлу из файла HTML, в котором вы сейчас находитесь. Если оба файла располагаются в одной папке, просто указывается требуемое название. Допустим, файл JavaScript сохранен в папке example, которая находится внутри папки с HTML-файлом — в таком случае вы пишете `example/nameOfTheFile.js`. Если JavaScript-файл лежит на одну папку выше, по сути располагаясь рядом с HTML-файлом вид записи будет следующим: `../nameOfTheFile.js`.

Когда вы откроете HTML-файл, вы должны увидеть следующее (рис. 1.4).

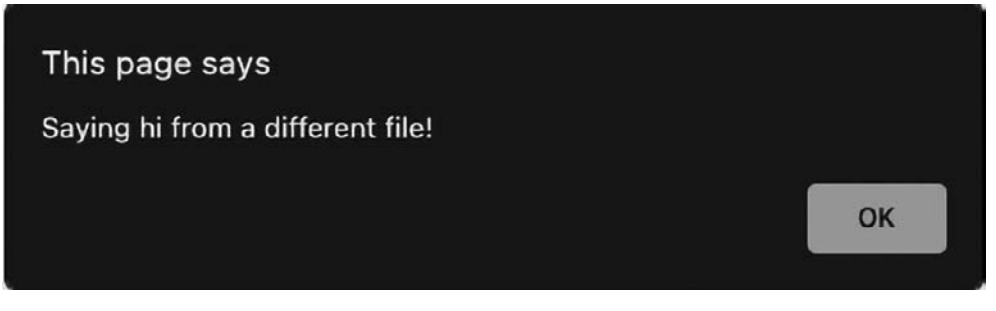


Рис. 1.4. Всплывающее окно, созданное JavaScript в стороннем файле

Практическое занятие 1.3

Подключение файла JavaScript

1. Создайте отдельный файл `app` с расширением `.js`.
2. Введите код JavaScript в файл `.js`.
3. Создайте подключение к отдельному файлу `.js` в коде HTML-файла, созданного на практическом занятии 1.2.
4. Откройте HTML-файл в браузере, чтобы удостовериться, что код JavaScript работает корректно.

Написание кода JavaScript

Итак, вы познакомились с контекстом и наверняка задаетесь вопросом: хорошо, но как же писать код JavaScript? Рассмотрим несколько важных моментов, которые необходимо держать в уме: как отформатировать код, как правильно использовать отступы и точки с запятой, а также как добавлять комментарии.

Форматирование кода

Код необходимо правильно отформатировать. Если у вас объемный файл, со множеством строк, но при работе с ним вы не придерживались нескольких базовых правил, то написанное вами будет крайне проблематично понять. Так каковы же основные правила форматирования? Два самых важных — это регулирование отступов и расстановка точек с запятой. Существуют также правила именования, но их мы рассмотрим отдельно позже.

Отступы и пробелы

Код, который вы создаете, обычно относится к определенному блоку (где написанное помещается в фигурные скобки (`{ вот так }`)) или к родительскому оператору. Поэтому в рамках одного блока строки даются с отступом, чтобы можно было легко различить, что является частью блока и когда начинается новый блок. В примере ниже вам не нужно понимать сам код — мы лишь рассмотрим, как велика разница между оформлением с отступами и без них.

Без переноса на новую строку:

```
let status = "new"; let scared = true; if (status === "new") { console.
log("Welcome to JavaScript!"); if (scared) { console.log("Don't worry
```

```
you will be fine!"); } else { console.log("You're brave! You are going
to do great!"); } } else { console.log("Welcome back, I knew you'd like
it!"); }
```

С переносом на новую строку, но без отступов:

```
let status = "new";
let scared = true;
if (status === "new") {
console.log("Welcome to JavaScript!");
if (scared) {
console.log("Don't worry you will be fine!");
} else {
console.log("You're brave! You are going to do great!");
}
} else {
console.log("Welcome back, I knew you'd like it!");
}
```

С переносом на новую строку и с отступами:

```
let status = "new";
let scared = true;
if (status === "new") {
  console.log("Welcome to JavaScript!");
  if (scared) {
    console.log("Don't worry you will be fine!");
  } else {
    console.log("You're brave! You are going to do great!");
  }
} else {
  console.log("Welcome back, I knew you'd like it!");
}
```

Благодаря отступам легко определить границы конкретного блока, в нашем случае они видны по оператору `if`, совпадающему по уровню с `}`. В примере без отступов, чтобы определить, где заканчивается блок оператора `if`, вам придется считать скобки. На работоспособность кода это не влияет, но все же стоит использовать отступы. Вы позже сами скажете себе спасибо.

Точки с запятой

Каждый оператор должен заканчиваться точкой с запятой. JavaScript очень снисходителен и, если вы забудете поставить нужный знак в конце конкретного кода, скорее всего, поймет вас правильно, но лучше выработайте полезную привычку разбираться со знаками сразу. Когда вы объявляете блок кода, например оператор `if` или цикл, ставить точку с запятой в конце строки не надо, она обязательна только для отдельных операторов.

Комментарии к коду

С помощью знака комментария вы можете дать интерпретатору команду игнорировать часть кода. Такая часть не будет выполняться. Комментарии очень полезны, особенно в случаях, если:

а) вы не хотите, чтобы какой-то код исполнялся во время запуска скрипта. Просто вынесите этот код в комментарий — и интерпретатор его не увидит;

б) вам нужно добавить к коду контекст (метаданные): например, имя автора или описание функции;

в) вы хотите пояснить фрагменты кода или обосновать выбор решения — в таких случаях комментарии даже необходимы.

Комментарии бывают однострочными и многострочными. Рассмотрим пример:

```
// Я однострочный комментарий
// console.log("однострочный комментарий не логируется");

/* Я многострочный комментарий. Все, что находится между символами слеш-
звездочка и звездочка-слеш, не будет выполнено.
console.log("Я не логируюсь, потому что я комментарий");
*/
```

В этом фрагменте кода представлены оба типа комментариев.

Первый — однострочный. Его можно использовать также в строке кода — и он будет действовать до конца строки. Все, что идет после `//`, будет проигнорировано.

Второй — многострочный. Он начинается с `/*`, а заканчивается так: `*/`.

Практическое занятие 1.4

Добавление комментариев

1. Добавьте новый оператор в код JavaScript, объявив значение переменной, — как это сделать, вы узнаете в следующей главе, а пока просто используйте команду:

   ```
   let a = 10;
   ```

2. В конце оператора добавьте комментарий о том, что `10` — это значение.
3. Выведите значение с помощью `console.log()`. Добавьте комментарий, поясняющий, что эта команда делает.
4. В конце кода JavaScript используйте многострочный комментарий. В реальном производственном сценарии вы можете использовать это пространство для краткого описания цели файла.

Функция prompt()

Рассмотрим еще одну полезную команду — функцию `prompt()`. По большей части она работает как команда `alert`, но имеет окно ввода информации для пользователя. Очень скоро мы узнаем, как хранить переменные, и, как только вы это изучите, вы сможете сохранить результат этой функции и что-то с ним сделать. А пока замените `alert()` на `prompt()` в файле `Hi.html`, например, так:

```
prompt("Hi! How are you?");
```

Далее обновите HTML-файл. Вы получите всплывающее окно с полем ввода, в котором сможете ввести текст, как показано на рис. 1.5.

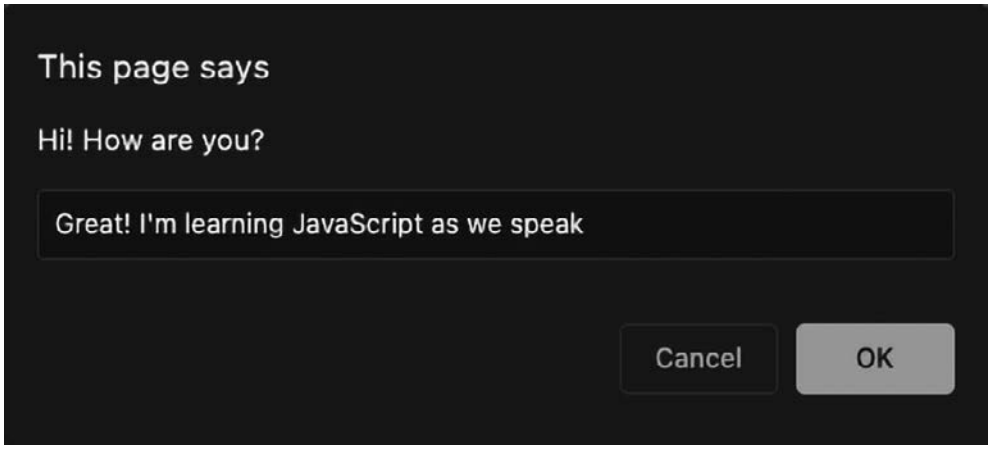


Рис. 1.5. Страница с запросом на применение ввода

Значение, которое вы (или любой другой пользователь) введете, будет возвращено в скрипт и сможет быть использовано в коде! Таким образом, функция `promt()` — отличный способ получить от пользователя данные и с их помощью модифицировать ваш код.

Случайные числа

Развлечения ради в первых главах предлагаем вам создать генератор случайных чисел на JavaScript — абсолютно нормально, если вы сейчас не совсем понимаете, о чем мы. Давайте познакомимся с командой для создания случайных чисел:

```
Math.random();
```

Можно реализовать ее в консоли, логируя результат:

```
console.log(Math.random());
```

Результатом будет десятичная дробь в пределах от 0 до 1. Если вам нужно случайное число в пределах от 0 до 100, умножьте значение на 100 следующим образом:

```
console.log(Math.random() * 100);
```

Не переживайте, мы рассмотрим математические операторы в главе 2.

Десятичная дробь вам не подходит, можно использовать функцию `Math.floor` — она округлит результат до ближайшего целого числа.

```
console.log(Math.floor(Math.random() * 100));
```

Не беспокойтесь, если пока вам еще не все понятно, — мы хорошенько изучим все эти вещи по ходу книги. Встроенные методы JavaScript мы рассмотрим в главе 8.

Проект текущей главы

Создание HTML-файла и привязка JavaScript-файла

Создайте HTML-файл и отдельный JavaScript-файл. Сделайте привязку JavaScript-файла к HTML-файлу.

1. В файле JavaScript выведите свое имя на экран и добавьте к своему коду несколько строк комментария.
2. Попробуйте закомментировать консольное сообщение в файле JavaScript так, чтобы в консоли ничего не отображалось.

Вопросы для самопроверки

1. Какой HTML-синтаксис присоединяет к веб-странице внешний файл JavaScript?
2. Можно ли запустить файл JavaScript с расширением JS в браузере?
3. Как написать многострочный комментарий в JavaScript?
4. Назовите лучший способ исключить код из процесса запуска, при том условии, что он может вам еще понадобиться после отладки.

Резюме

Отличная работа! Начало работе с JavaScript положено! В этой главе мы рассмотрели множество полезной информации, которую необходимо усвоить перед тем, как программировать что-либо на JavaScript. Вы узнали, что этот язык применим во многих сферах, среди которых одна из самых популярных — сеть Интернет. Браузеры работают с JavaScript, потому что в них встроен специальный интерпретатор, запускающий код JavaScript. Мы рассмотрели различные способы написания кода JavaScript. Вы узнали, что для написания и запуска кода нужны IDE.

Добавление кода JavaScript на веб-страницу можно реализовать разными способами. Мы разобрались, как сделать его частью скрипта и как подключить внешний файл JavaScript к странице. Закончили главу сведениями о том, как создать правильную и разборчивую структуру кода, дополненную комментариями с необходимой информацией. Вы также увидели, как вывести данные на экран методом `console.log()` и с помощью `prompt()` запросить информацию от пользователя. И наконец, изучили возможность генерировать случайные числа благодаря функции `Math.random()`.

Далее мы рассмотрим основные типы данных JavaScript, а также операторы для работы с ними.

2 Основы JavaScript

В этой главе мы будем иметь дело с некоторыми значимыми строительными блоками JavaScript: с переменными и операторами. Начнем с переменных: вы узнаете, что это такое, какие типы данных существуют. Эти базовые объекты необходимы для хранения и работы со значениями переменных в скриптах, реализации их как динамических величин.

Как только вы разберетесь с переменными, вы будете готовы работать с операторами. На этом этапе будут обсуждаться арифметические, присваивающие, условные и логические операторы. Операторы необходимы, чтобы изменять переменные или сообщать нам что-то об этих переменных. С их помощью мы можем выполнять базовые расчеты на основе различных данных — например, введенных пользователем.

Мы рассмотрим следующие темы:

- переменные;
- примитивы;
- анализ и модификацию типов данных;
- операторы.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Переменные

Переменные — первый строительный блок, который встретится вам при изучении большинства языков программирования. Значения переменных в вашем коде могут быть различными при каждом его запуске. Вот пример с двумя переменными:

```
firstname = "Maaike";
x = 2;
```

Во время выполнения кода им может быть присвоено новое значение:

```
firstname = "Edward";
x = 7;
```

Без переменных фрагмент кода будет давать один и тот же результат при каждом запуске. В некоторых случаях нужно, чтобы именно так и было, но благодаря переменным код можно сделать намного более полезным, заставляя его каждый раз делать что-то новое.

Объявление переменных

Когда вы создаете новую переменную, ее необходимо объявить. Для этого требуются специальные операторы: `let`, `var` и `const`. Вкратце рассмотрим их использование. Когда вы вызываете переменную во второй раз, чтобы присвоить ей новое значение, вам необходимо только ее имя:

```
let firstname = "Maria";
firstname = "Jacky";
```

В примерах нашим переменным в коде мы будем давать значения вручную. Это называется «хардкодинг» (жесткое программирование), поскольку значение переменной определяется в вашем скрипте, а не динамически поступает с какого-либо ввода. Это то, что редко применяется в реальном коде: чаще всего мы получаем значение из внешнего источника, такого как заполненное пользователем поле на веб-сайте, база данных или другой код, вызывающий ваш код. Использование переменных, поступающих из внешних источников, вместо жестко прописанных позволяет скриптам адаптироваться к новой информации без необходимости переписывать код.

Вы видите, насколько мощным строительным блоком являются переменные. Какое-то время в примерах мы будем жестко кодировать переменные, поэтому скрипты не будут меняться до тех пор, пока программист не изменит код. Однако скоро вы узнаете, как заставить переменные принимать значения из внешних источников.

let, var и const

Определение переменной состоит из трех частей: определяющего слова (`let`, `var` или `const`), названия и значения. Начнем с изучения разницы между `let`, `var` и `const`. Здесь представлены несколько примеров переменных, использующих различные ключевые слова:

```
let nr1 = 12;
var nr2 = 8;
const PI = 3.14159;
```

`let` и `var` используются для переменных, которым где-то в программе может быть присвоено новое значение. Различия между `let` и `var` не так просты. Они связаны с областью их применения.

Если вы поймете сказанное ниже — это здорово, но совершенно не страшно, если этого не случится. Вы разберетесь в данном вопросе достаточно скоро, продолжая читать книгу.

Переменная `var` относится к *глобальным переменным*; `let` — *локальная переменная*. Глобальность `var` означает, что вы можете использовать переменную, ранее определенную как `var`, в любом месте кода. Область же действия `let` ограничивается только конкретным блоком кода: вы можете использовать переменные, определенные с помощью `let`, только там, где они и были определены. Помните, что блок кода всегда будет начинаться с `{` и заканчиваться `}` — именно так вы можете распознать его границы.

Оператор `const` используется для переменных, которым присваивается значение только один раз, например `π`, которое априори не меняется. Если вы попытаетесь переназначить переменную `const`, система выдаст сообщение об ошибке:

```
const someConstant = 3;
someConstant = 4;
```

В результате вы получите:

```
Uncaught TypeError: Assignment to constant variable.
```

Почти во всех примерах мы будем использовать `let` — просто поверьте, что на практике в большинстве случаев вы будете сталкиваться именно с ним.

Именование переменных

Мы добрались до присвоения имен переменным, и тут действуют свои правила.

- Переменные должны начинаться со строчной, а не с заглавной буквы и быть описательными. Если значение содержит данные о возрасте, не стоит присваивать переменной имя `x`, лучше использовать `age`. Так, когда вы позже прочтете свой скрипт, вы сможете легко понять, что вы реализовали.
- Имена переменных не могут содержать пробелы, только нижние подчеркивания. Если вы используете пробел, JavaScript не распознает написанное как одну переменную.

Мы будем использовать «горбатый регистр» — стиль, при котором для описания переменной можно взять несколько слов. Начнем со строчной буквы, а затем каждое новое слово будем писать с заглавной, например: ageOfBuyer.

Какими бы ни были правила там, где вы работаете, главное — последовательность. Если все имена будут оформлены аналогично, код будет выглядеть более чистым и читабельным, что значительно упростит последующее внесение изменений.

Значение вашей переменной может быть любым. Давайте начнем с самой простой формы, которую могут принимать переменные, — примитива.

Примитивы

Переменным присваивается значение. И эти значения могут быть разных типов. JavaScript — язык свободных типов. Это означает, что JavaScript определяет тип данных на основе значения. Тип не обязательно должен быть назван явно. Например, если вы присвоили переменной значение `5`, JavaScript автоматически определит тип данных как численный.

Существует различие между примитивными типами данных и другими, более сложными. В этом разделе мы рассмотрим примитив — относительно простую структуру данных. В JavaScript есть семь типов примитивов: `String`, `Number`, `BigInt`, `Boolean`, `Symbol`, `undefined` и `null`. Далее познакомимся с каждым из них более подробно.

Тип данных String

Строковый тип данных используется для передачи текстовых значений. Это последовательность символов. Есть несколько способов объявить переменную строкового типа:

- двойные кавычки;
- одинарные кавычки;
- обратные апострофы: специальные шаблонные строки, в которых вы можете напрямую использовать переменные.

Одинарные и двойные кавычки используются так:

```
let singleString = 'Hi there!';
let doubleString = "How are you?";
```

Вы можете использовать тот вариант, который вам больше нравится, если только не работаете с кодом, где оформление уже выбрали до вас. Опять же последовательность в этом деле является ключевым фактором.

Основное различие между одинарными и двойными кавычками заключается в следующем. Вы можете использовать одинарную кавычку в качестве апострофа в строках, заключенных в двойные кавычки — и наоборот. Если вы, объявляя строчную переменную, начинаете писать код с одинарной кавычки, то конец строки

определится вторым таким же знаком, даже если он будет в середине слова. Так, например, следующее написание приведет к ошибке, потому что строка будет заканчиваться апострофом внутри слова `let's`:

```
let funActivity = 'Let's learn JavaScript';
```

Слово `Let` будет определено как строка, но после него набор символов не будет интерпретирован. Но если вы объявите строку, заключив ее в двойные кавычки, строка не завершится, когда достигнет апострофа, потому что программа будет искать другую двойную кавычку. Следовательно, эта альтернатива будет работать лучше:

```
let funActivity = "Let's learn JavaScript";
```

Аналогичная история с двойными кавычками. Следующий код не будет работать:

```
let question = "Do you want to learn JavaScript? "Yes!"";
```

Компилятор не будет различать двойные кавычки, используемые в разных контекстах, и выдаст ошибку.

В строке, использующей обратные апострофы, вы можете ссылаться на переменные — значение переменной будет подставлено в строку. Как во фрагменте кода, представленном ниже:

```
let language = "JavaScript";
let message = `Let's learn ${language}`;
console.log(message);
```

Итак, переменные придется указывать с помощью довольно странного синтаксиса. Не пугайтесь! В этих строках они заключены между `${nameOfVariable}`. Подобный сложный синтаксис используется для того, чтобы избежать громоздкости, которая бы появилась в коде, оформи мы его привычным образом. В нашем случае вывод на экран будет выглядеть так:

```
Let's learn JavaScript
```

Как видите, переменная `language` заменяется ее значением: JavaScript.

Escape-символы

Допустим, мы хотим использовать в нашей строке двойные кавычки, одинарные кавычки и обратные апострофы. С теми ингредиентами, которые у нас есть сейчас, сделать подобное невозможно. Но для этой проблемы есть элегантное решение — специальный символ, с помощью которого можно сказать JavaScript: «Не читай следующий символ как обычно». Речь об escape-символе, обратном слеше.

Обратный слеш может использоваться для того, чтобы интерпретатор не видел одинарные или двойные кавычки и не заканчивал строку слишком рано:

```
let str = "Hello, what's your name? Is it \"Mike\"?";
console.log(str);
let str2 = 'Hello, what\'s your name? Is it "Mike"?';
console.log(str2);
```

В консоли выводится:

```
Hello, what's your name? Is it "Mike"?
Hello, what's your name? Is it "Mike"?
```

Как видим, оба типа кавычек внутри строк были прочитаны без ошибок. Обратный слеш перед кавычками придает им другое значение — в данном случае, что строка должна завершаться знаком препинания, а не индикатором.

У escape-символа есть много назначений. Вы можете использовать его для создания разрыва строки с помощью `\n` или для включения символа обратного слеша в текст с помощью `\\`:

```
let str3 = "New \nline.";
let str4 = "I'm containing a backslash: \\!";
console.log(str3);
console.log(str4);
```

Вывод будет таким:

```
New
line.
I'm containing a backslash: \!
```

Есть еще несколько вариантов использования escape-символов, но мы пока их оставим. Вернемся к примитивам в целом, обратив внимание на числовой тип данных.

Тип данных Number

Во многих языках существует очень четкая разница между различными типами чисел. Разработчики JavaScript решили использовать один тип данных для них всех: `Number` (если точнее, это 64-битное число с плавающей точкой). Этот тип может хранить довольно большие числа (как со знаком, так и без), десятичные дроби и многое другое.

Существуют различные виды чисел, которые он может представлять. Прежде всего, целые числа, например 4 или 89. Но тип данных `Number` также может использоваться для представления десятичных, экспоненциальных, восьмеричных, шестнадцатеричных и двоичных чисел. Следующий пример кода говорит сам за себя:

```
let intNr = 1;
let decNr = 1.5;
```

```
let expNr = 1.4e15;
let octNr = 0o10; //в десятичной системе будет 8
let hexNr = 0x3E8; //в десятичной системе будет 1000
let binNr = 0b101; //в десятичной системе будет 5
```

Вам не стоит переживать насчет последних трех позиций, если вы с ними не знакомы, это просто разные способы представления чисел, с которыми вы можете столкнуться в более продвинутых задачах информационных технологий. Главное здесь то, что все вышеперечисленные числа относятся к типу данных `Number`. Итак, целые числа — это числа, подобные этим:

```
let intNr2 = 3434;
let intNr3 = -111;
```

Числа с плавающей точкой тоже относятся к типу `Number`:

```
let decNr2 = 45.78;
```

И двоичные числа:

```
let binNr2 = 0b100; // десятичная версия будет 4
```

Итак, мы только что рассмотрели тип данных `Number` — он действительно используется очень часто. Но в особых случаях вам может понадобиться большее число.

Тип данных BigInt

Диапазон значений типа данных `Number` находится между 253–1 и –(253–1). `BigInt` вступает в дело, когда вам требуются числа больше (или меньше) этого интервала. Тип данных `BigInt` можно узнать по окончанию `n`:

```
let bigNr = 90071992547409920n;
```

Рассмотрим, что происходит, когда мы начинаем вычисления между ранее заданным целым числом типа `Number`, `intNr`, и значением типа `BigInt`, `bigNr`:

```
let result = bigNr + intNr;
```

Результат будет таким:

```
Uncaught TypeError: Cannot mix BigInt and other types, use explicit conversions
```

Ошибка типов данных — `TypeError`! Совершенно ясно: что-то пошло не так. Для выполнения операций нельзя смешивать тип данных `BigInt` с типом данных `Number`. Это то, что следует запомнить на будущее, когда вы будете работать с `BigInt`, — вы можете использовать `BigInt` только с другими `BigInt`.

Тип данных Boolean

Логический тип данных `Boolean` может принимать два значения: `true` (истина) и `false` (ложь). И ничего больше. Этот тип данных часто используется в коде, особенно в логических выражениях:

```
let bool1 = false;
let bool2 = true;
```

В этом примере представлены возможные параметры логического типа `Boolean`. Он используется в ситуациях, когда вы хотите определить значение `true` или `false` (которое может означать «вкл./выкл.» или «да/нет») для какой-то ситуации. Например, вы хотите обозначить, будет ли элемент удален:

```
let objectIsDeleted = false;
```

Или подсветка индикатора будет включена или выключена:

```
let lightIsOn = true;
```

Соответственно, в этих записях переменные указывают на то, что объект не удален и что горит соответствующий индикатор.

Тип данных Symbol

Символьный тип `Symbol` совершенно новый и представлен в ES6 (мы упоминали стандарт ECMAScript 6 в главе 1 «Начало работы с JavaScript»). Он может устанавливаться в случаях, когда важно подчеркнуть, что переменные не равны, даже если их значение и тип одинаковы (в этом случае они обе примут тип символа). Сравните следующие объявления строк с объявлениями символов (и те и другие имеют одинаковое значение):

```
let str1 = "JavaScript is fun!";
let str2 = "JavaScript is fun!";
console.log("These two strings are the same:", str1 === str2);

let sym1 = Symbol("JavaScript is fun!");
let sym2 = Symbol("JavaScript is fun!");
console.log("These two Symbols are the same:", sym1 === sym2);
```

Результат будет следующим:

```
These two strings are the same: true
These two Symbols are the same: false
```

В первой части кода JavaScript приходит к выводу, что строки одинаковые, у них один тип и значение. Однако во второй части кода каждый символ определяется как уникальный. Поэтому, хотя содержимое символов одинаковое, они не совпадают и при сравнении выводится значение `false`. Эти символьные типы данных могут быть очень удобны в качестве свойств объектов, которые мы рассмотрим в главе 3.

Значение undefined

JavaScript крайне специфичный язык. В нем есть специальный тип данных для переменной, которой не было присвоено значение. И он называется `undefined`:

```
let unassigned;
console.log(unassigned);
```

Результат будет таким:

```
Undefined
```

Мы также можем целенаправленно назначать `undefined` — важно знать, что такое возможно. Но еще важнее знать, что присвоение типа `undefined` вручную — плохая практика:

```
let terribleThingToDo = undefined;
```

Да, так можно, но лучше так не делайте. По целому ряду причин — например, в случаях, когда требуется проверить, совпадают ли две переменные. Если одна переменная не определена (undefined), а вашей собственной переменной вы вручную присвоили значение `undefined`, переменные будут считаться равными. Но вы же хотите знать, действительно ли значения равны, а не только то, что они не определены. Так, кличка чьего-то домашнего животного и фамилия хозяина могут считаться равными, тогда как на самом деле они являются просто пустыми значениями.

Значение null

Проблема, с которой мы столкнулись выше, может быть решена с помощью примитивного типа `null`. `null` — это специальное значение, указывающее на то, что переменная пуста или имеет неизвестное значение. Данное значение чувствительно к регистру — для написания `null` используются только строчные буквы:

```
let empty = null;
```

Чтобы решить проблему, с которой мы столкнулись при объявлении переменной как неопределенной, следует установить для нее значение `null` — и проблем не возникнет. Это одна из причин, по которой лучше присваивать переменной значение `null`, если вы хотите указать, что она изначально пуста и неизвестна:

```
let terribleThingToDo = undefined;
let lastName;
console.log("Same undefined:", lastName === terribleThingToDo);

let betterOption = null;
console.log("Same null:", lastName === betterOption);
```

Результат будет следующим:

```
Same undefined: true
Same null: false
```

Как видим, автоматически не определенная переменная, `lastName`, и намеренно не определенная переменная, `terribleThingToDo`, считаются равными — это проблематично. С другой стороны, `lastName` и `betterOption`, которой было присвоено значение `null`, не равны.

Анализ и модификация типов данных

Мы рассмотрели примитивные типы данных. Существует несколько встроенных методов JavaScript, которые помогут справиться с распространенными проблемами, возникающими при работе с примитивами. Встроенные методы — это части логической схемы, которые можно использовать сразу и не мучиться над их созданием вручную.

Мы уже встречали один из встроенных методов: console.log().

Существует множество встроенных методов. В этой главе вы познакомитесь лишь с некоторыми из них — теми, с которыми вы в своей практике столкнетесь прежде всего.

Определение типа переменной

Иногда трудно определить тип данных, с которым вы имеете дело, особенно в случае `null` и `undefined`. Рассмотрим оператор `typeof`. Он возвращает тип переменной. С его помощью вы можете получить эту информацию. Введите `typeof`, затем либо пробел, за которым следует рассматриваемая переменная, либо эту переменную в скобках:

```
testVariable = 1;
variableTypeTest1 = typeof testVariable;
variableTypeTest2 = typeof(testVariable);
console.log(variableTypeTest1);
console.log(variableTypeTest2);
```

Как вы могли предположить, оба метода выведут `number`. Скобки не требуются, потому что технически `typeof` является оператором, а не методом, в отличие от `console.log`. Хотя на практике вы наверняка заметите, что использование скобок облегчает чтение вашего кода. Рассмотрим следующий пример:

```
let str = "Hello";
let nr = 7;
let bigNr = 12345678901234n;
let bool = true;
let sym = Symbol("unique");
let undef = undefined;
let unknown = null;

console.log("str", typeof str);
console.log("nr", typeof nr);
console.log("bigNr", typeof bigNr);
console.log("bool", typeof bool);
console.log("sym", typeof sym);
console.log("undef", typeof undef);
console.log("unknown", typeof unknown);
```

Здесь, в той же команде вывода `console.log()`, мы вводим имя каждой переменной (в виде строки, объявленной в двойных кавычках), затем ее тип (используя `typeof`). В результате получим:

```
str string
nr number
bigNr bigint
bool boolean
sym symbol
undef undefined
unknown object
```

Последний тип — это и есть тип `null`. В выводе можно наблюдать, что `typeof null` возвращает значение `object`, в то время как на самом деле `null` является примитивом, а не объектом. Это ошибка, которая существует в JavaScript с незапамятных времен и которую пока нет возможности исправить из-за проблем с обратной совместимостью. Не беспокойтесь о ней: на выполнение программ она не повлияет, но имейте эту ошибку в виду, поскольку в ближайшее время она никуда не денется и может повлиять на адекватную работу приложений.

Преобразование типов данных

В JavaScript можно изменить тип данных переменных. Иногда JavaScript делает это автоматически. Как вы считаете, каким будет результат запуска следующего фрагмента кода?

```
let nr1 = 2;
let nr2 = "2";
console.log(nr1 * nr2);
```

Мы попытались умножить переменную типа `Number` на переменную типа `String`. JavaScript не выдает ошибку (как сделали бы многие другие языки), а пытается сначала преобразовать значение переменной `String` в `Number`. Если это можно сделать, команда выполняется без каких-либо проблем, как если бы были объявлены две переменные типа `Number`. В таком случае метод `console.log()` выдает результат `4` на экране.

Но это достаточно опасный ход! Угадайте, что делает данный фрагмент кода:

```
let nr1 = 2;
let nr2 = "2";
console.log(nr1 + nr2);
```

Результатом будет `22`. Знак плюс можно использовать для объединения строк. Поэтому вместо преобразования `string` в `number` код преобразует `number` в `string`. И соединяет две строки — `2` и `2`, выдавая в результате строку `22`. К счастью, нам не нужно полагаться на поведение JavaScript при преобразовании типов данных. Существуют встроенные функции, которые мы можем для этого использовать.

Рассмотрим три метода преобразования: `String()`, `Number()` и `Boolean()`. Первый преобразует переменную в тип `String`. Он, в общем-то, принимает любое значение, включая `undefined` и `null`, и заключает его в кавычки.

Второй пытается преобразовать переменную в тип `Number`. Если это невозможно сделать логически, значением переменной станет `NaN` (not a number — «не число»). `Boolean()` преобразует переменную в тип `Boolean`. Данный метод будет работать для всего, кроме `null`, `undefined`, `0` (число), пустой строки и `NaN`. Посмотрим на эти методы в действии:

```
let nrToStr = 6;
nrToStr = String(nrToStr);
console.log(nrToStr, typeof nrToStr);

let strToNr = "12";
strToNr = Number(strToNr);
console.log(strToNr, typeof strToNr);

let strToBool = "any string will return true";
strToBool = Boolean(strToBool);
console.log(strToBool, typeof strToBool);
```

Результат будет таким:

```
6 string
12 number
true boolean
```

Выглядит довольно просто, но не все так очевидно. Например, вы все еще можете получить не то, на что рассчитывали:

```
let nullToNr = null;
nullToNr = Number(nullToNr);
console.log("null", nullToNr, typeof nullToNr);
let strToNr = "";
strToNr = Number(strToNr);
console.log("empty string", strToNr, typeof strToNr);
```

В консоли появится следующий результат:

```
null 0 number
empty string 0 number
```

Как видим, пустая строка и `null` в итоге дают `0`. Это выбор, сделанный создателями JavaScript, о котором вам нужно знать — данная опция может пригодиться в тех случаях, когда вы захотите преобразовать строку в `0`, в то время как она пуста или равна нулю.

Далее введем фрагмент:

```
let strToNr2 = "hello";
strToNr2 = Number(strToNr2);
console.log(strToNr2, typeof strToNr2);
```

В консоли отобразится следующая запись:

```
NaN number
```

Следовательно, все, что нельзя интерпретировать как число, просто удалив кавычки, будет оцениваться как `NaN` (не число).

Продолжим таким кодом:

```
let strToBool2 = "false";
strToBool2 = Boolean(strToBool2);
console.log(strToBool2, typeof strToBool2);

let strToBool = "";
strToBool = Boolean(strToBool);
console.log(strToBool, typeof strToBool);
```

Результат будет следующим:

```
true boolean
false boolean
```

Этот пример показывает, что практически любая строка возвращает значение `true`, когда преобразуется в тип `Boolean`, даже строка со значением `false`! Только пустая строка, `null` и `undefined` отправят логическое значение `false`.

Давайте еще немного поэкспериментируем. Как думаете, каким будет результат?

```
let nr1 = 2;
let nr2 = "2";
console.log(nr1 + Number(nr2));
```

В консоли вы увидите `4`! Строка преобразуется в число до выполнения операции «плюс» — соответственно, это математическая операция, а не объединение строк. В следующих разделах данной главы мы рассмотрим операторы более подробно.

Практическое занятие 2.1

Каковы типы переменных, перечисленных ниже? Проверяйте с помощью оператора `typeof` и выводите результаты на экран.

```
let str1 = 'Laurence';
let str2 = "Svekis";
let val1 = undefined;
let val2 = null;
let myNum = 1000;
```

Операторы

После знакомства с довольно большим количеством типов данных и некоторыми способами их преобразования пришло время для следующего строительного блока: операторов. Они будут полезны всякий раз, когда мы захотим работать с переменными, изменять их, выполнять над ними вычисления и сравнивать. Они называются операторами, потому что мы используем их, чтобы *оперировать* переменными.

Арифметические операторы

Арифметические операторы могут использоваться для выполнения операций с числами. Большинство из них покажутся вам очень естественными: это базовая математика, с которой вы уже сталкивались раньше.

Сложение

Сложение в JavaScript выполняется очень просто, мы уже встречались с ним. Для этой операции используется оператор `+`.

```
let nr1 = 12;
let nr2 = 14;
let result1 = nr1 + nr2;
```

Этот оператор также может быть очень полезен для объединения строк. Обратите внимание на пробел после `Hello` — он нужен в конечном результате:

```
let str1 = "Hello ";
let str2 = "addition";
let result2 = str1 + str2;
```

В итоге `result1` и `result2` будут выведены так:

```
26
Hello addition
```

Как видите, сложение чисел и строк приводит к разным результатам. Если мы сложим две разные строки, это объединит их в одну строку.

Практическое занятие 2.2

Создайте переменную для своего имени, еще одну — для вашего возраста, и еще одну — для строки, определяющей, умеете ли вы кодировать в JavaScript или нет.

Выведите на экран следующее суждение, где `name`, `age` и `true/false` будут переменными:

```
Hello, my name is Maaike, I am 29 years old and I can code
JavaScript: true.
```

Вычитание

Вычитание также работает ожидаемым образом. Для этого действия используется оператор `-`. Как думаете, что хранится в переменной во втором примере?

```
let nr1 = 20;
let nr2 = 4;
let str1 = "Hi";
```

```
let nr3 = 3;
let result1 = nr1 - nr2;
let result2 = str1 - nr3;
console.log(result1, result2);
```

Вывод будет следующим:

```
16 NaN
```

Первый результат — число `16`. А второй результат намного более интересен. `NaN` не ошибка: это просто заключение, что результат вычитания числа из строки не является числом. Спасибо, что не падаешь, JavaScript!

Умножение

Мы можем умножать два числовых значения с помощью оператора `*`. В отличие от некоторых других языков в JavaScript данное действие не получится совершить с числом и строкой.

Результатом умножения числового и нечислового значений является `NaN`:

```
let nr1 = 15;
let nr2 = 10;
let str1 = "Hi";
let nr3 = 3;
let result1 = nr1 * nr2;
let result2 = str1 * nr3;
console.log(result1, result2);
```

Вывод:

```
150 NaN
```

Деление

Еще один простой оператор — это деление. Мы можем разделить одно число на другое, используя оператор `/`:

```
let nr1 = 30;
let nr2 = 5;
let result1 = nr1 / nr2;
console.log(result1);
```

Вывод будет следующим:

```
6
```

Возведение в степень

Возведение в степень означает возведение определенного базового числа в степень экспоненты, например x^y (произносится как «возведение x в степень y»). Это значит, что x будет умножено само на себя y раз. Чтобы возвести число в степень в JavaScript, используем ** для вызова оператора:

```
let nr1 = 2;
let nr2 = 3;
let result1 = nr1 ** nr2;
console.log(result1);
```

В консоли вы увидите:

```
8
```

Результат этой операции равен 2 в степени 3 ($2 \times 2 \times 2$), что составляет `8`. Чтобы не превращать книгу в урок математики, добавим только, что, используя дробные показатели, с помощью этой же операции мы можем найти корень числа: например, квадратный корень из значения — это то же самое, что возведение его в степень 0,5.

Оператор деления по модулю

Данный оператор обычно требует пояснений. Модуль — это операция, определяющая остаток после полного деления одного числа на другое. Количество раз, которое одно число может поместиться в другое, здесь не имеет значения — результатом будет остаток. Для вызова оператора используется символ `%`. Вот несколько примеров:

```
let nr1 = 10;
let nr2 = 3;
let result1 = nr1 % nr2;
console.log(`${nr1} % ${nr2} = ${result1}`);

let nr3 = 8;
let nr4 = 2;
let result2 = nr3 % nr4;
console.log(`${nr3} % ${nr4} = ${result2}`);

let nr5 = 15;
let nr6 = 4;
let result3 = nr5 % nr6;
console.log(`${nr5} % ${nr6} = ${result3}`);
```

И вывод:

```
10 % 3 = 1
8 % 2 = 0
15 % 4 = 3
```

Первый пример — `10 % 3`, где 3 умещается 3 раза в числе 10 с остатком 1. Второй — `8 % 2`. Здесь результат 0, потому что 2 повторяется 4 раза в числе 8 и ничего не остается. Последний — `15 % 4`, где 4 умещается 3 раза в числе 15, после чего получается остаток 3.

По сути, это те действия, которые вы бы совершили у себя в голове, если бы вас попросили добавить 125 минут к текущему времени. Вероятно, вы бы сделали две вещи: целочисленное деление, чтобы определить, сколько целых часов содержится в 125 минутах, а затем 125 модуль 60 (в терминах JavaScript, `125 % 60`), чтобы сделать вывод, что вам придется добавить еще 5 минут. Скажем, наше текущее время 09:59. Скорее всего, вы начнете с добавления 2 часов и дойдете до 11:59, а затем добавите 5 минут, выполнив еще одну операцию по модулю с 59 и 5, добавив еще 1 час к общей сумме и оставив 4 минуты: 12:04.

Унарные операторы: инкремент и декремент

Последние два оператора нашего раздела арифметических операторов, если вы новичок в программировании (или работали с другим языком), вероятно, вам еще не знакомы. Это инкремент и декремент. Здесь нам пригодится термин *«операнд»*. Операнд — то, к чему применяется оператор. Если мы пишем `x + y`, то x и y — это операнды.

Для данных операторов потребуется только один операнд — поэтому они также называются *унарными* операторами. Если перед нами `x++`, то эту запись можно представить в виде $x = x + 1$. То же самое справедливо для оператора декремента: `x--` принимает вид $x = x - 1$:

```
let nr1 = 4;
nr1++;
console.log(nr1);

let nr2 = 4;
nr2--;
console.log(nr2);
```

Вывод будет следующим:

```
5
3
```

Операторы prefix и postfix

Инкремент может располагаться после операнда (`x++`), в таком случае он будет называться *постфиксным унарным оператором*. Он также может располагаться до операнда (`++x`) — *префиксный унарный оператор*. Однако он выполняет кое-что

другое — следующие строки могут быть сложными, так что не волнуйтесь, если вам нужно прочитать их несколько раз и внимательно ознакомиться с примерами.

Сначала отправляется переменная, потом выполняется постфикс и только после этого — операция. В следующем примере `nr` повышается на 1 *после* логирования. Поэтому в первый раз переменная логируется без изменений, потому что обновление еще не произошло. Обновление произойдет во второй раз:

```
let nr = 2;
console.log(nr++);
console.log(nr);
```

Вывод будет следующим:

```
2
3
```

Префикс выполняется *перед* отправкой переменной, поэтому чаще всего используется именно он. Взгляните на следующий пример.

```
let nr = 2;
console.log(++nr);
```

В результате вы получите:

```
3
```

Если вы можете определить, какие результаты следующих фрагментов кода логируются, то прекрасно, вы действительно разобрались в теме:

```
let nr1 = 4;
let nr2 = 5;
let nr3 = 2;
console.log(nr1++ + ++nr2 * nr3++);
```

Результат будет равен `16`. Сначала будет выполнено умножение в соответствии с основным математическим порядком операций. Для умножения используется число 6 (префикс поднял значение 5 до умножения) и 2 (постфикс поднял значение 2 только после выполнения операции, а значит, на текущие вычисления это не повлияло). Получается 12. У переменной `nr1` постфиксный оператор, поэтому он будет исполнен после операции сложения. Таким образом, результатом сложения 12 и 4 будет 16.

Сочетание операторов

Можно создавать сочетания операторов, и работать они будут так же, как и в математике. Они будут исполняться точно в том же порядке, и совершенно не обязательно слева направо. Это происходит благодаря явлению, называемому *приоритетом оператора*.

Есть еще одна деталь, которую следует принять во внимание, — группировка. Вы можете группировать операторы с помощью (и). Операции, заключенные в круглые скобки, имеют наивысший приоритет. После этого порядок операций определяется в зависимости от их типа (сначала с наивысшим приоритетом). Если операции имеют одинаковый приоритет, они выполняются слева направо.

Название	**Символ**	**Пример**
Группирование	`(...)`	`(x + y)`
Возведение в степень	`**`	`x ** y`
Префиксы инкремент и декремент	`--, ++`	`--x, ++y`
Умножение, деление, модуляция	`*, /, %`	`x * y, x / y, x % y`
Сложение и вычитание	`+, -`	`x + y, x - y`

Практическое занятие 2.3

Напишите фрагмент кода для вычисления гипотенузы треугольника с использованием теоремы Пифагора при заданных значениях двух других сторон. Теорема гласит, что отношение между сторонами прямоугольного треугольника выражается так: $a^2 + b^2 = c^2$.

Теорема Пифагора применима только к прямоугольным треугольникам. Стороны, соединенные под углом 90 градусов, называются катетами (в формуле обозначены как a и b). Самая длинная сторона, не прилегающая к прямому углу, называется гипотенузой, обозначена буквой c.

Чтобы получить значение для `a` и `b`, можно использовать `prompt()`. Напишите код, который будет запрашивать от пользователя длину катетов `a` и `b`. Затем возведите значения `a` и `b` в квадрат. После сложите результаты и извлеките квадратный корень. Полученное значение выведите на экран.

Операторы присваивания

Мы уже видели один назначающий оператор, когда присваивали значения переменным. Для операции присваивания чаще всего используется знак `=`. Но есть несколько других способов. Каждый двоичный арифметический оператор имеет соответствующий оператор присваивания — так можно сделать написанный код короче. Например, `x += 5` означает `x = x + 5`, а `x ** = 3` означает `x = x ** 3` (x в степени 3).

В этом примере мы объявим переменную x и зададим ей начальное значение 2:

```
let x = 2;
x += 2;
```

После операции присваивания значение x стало 4, потому что x += 2 — это то же самое, что $x = x + 2$.

В следующей операции присваивания мы вычтем 2:

```
x -= 2;
```

После этой операции x снова примет значение 2 ($x = x - 2$). Далее мы умножим значение на 6:

```
x *= 6;
```

Когда эта строка кода будет исполнена, значение x с 2 изменится на 12 (x = x * 6). В следующей строке мы используем оператор присваивания, чтобы продемонстрировать деление:

```
x /= 3;
```

После деления x на 3 получим значение 4. Далее мы используем оператор присваивания для возведения в степень:

```
x **= 2;
```

Значение x теперь равно 16, потому что предыдущее значение 4, возведенное в степень 2, равняется 16 (4×4). Последний оператор присваивания продемонстрирует модуляцию:

```
x %= 3;
```

После выполнения этого оператора значение x стало равно 1, потому что 3 помещается 5 раз в числе 16 и остается 1.

Практическое занятие 2.4

Создайте три численные переменные: *a*, *b* и *c*. Обновите эти переменные с помощью следующих действий, используя операторы присваивания:

- сложите *b* и *a*;
- разделите *a* на *c*;
- замените значение модуляцией *c* и *b*;
- выведите все три числа на экран.

Операторы сравнения

Сравнительные операторы отличаются от тех, которые мы уже рассмотрели. Результат их действия всегда принимает значение типа Boolean: `true` или `false`.

Равно

Существует несколько операторов равенства, которые определяют, равны ли два значения. Они бывают двух видов: только равное значение или равные значение и тип данных. Первый возвращает значение `true`, когда значения равны, даже если тип отличается, в то время как второй возвращает значение `true` только тогда, когда значение и тип данных совпадают:

```
let x = 5;
let y = "5";
console.log(x == y);
```

Оператор double equals (двойной знак равно) означает, что сравниваться будут только значения, но не типы данных. Если обе переменные имеют значение `5`, в консоли выводится `true`. Такой тип равенства обычно называют свободным равенством.

Оператор triple equals (три знака равно) означает, что он будет оценивать как значение, так и тип данных, чтобы определить, равны ли обе переменные или нет. Чтобы этот оператор отправил значение `true`, они должны быть тождественно равны, но в нашем случае это не так, поэтому оператор вернет значение `false`:

```
console.log(x === y);
```

Также данный оператор называется строгим равенством. Тройное равенство используется в большинстве случаев, когда нужно проверить равенство, так как только с его помощью вы можете быть уверены, что обе переменные действительно равны.

Не равно

Не равно очень похоже на равно, только оно делает все наоборот — возвращает `true`, когда две переменные не равны, и `false`, когда равны. Для обозначения этого оператора используется восклицательный знак перед знаком равно:

```
let x = 5;
let y = "5";
console.log(x != y);
```

Этот код вернет в консоль значение `false`. Если вам интересно, как все работает, взгляните еще раз на операторы double и triple equals: у операторов неравнозначности принцип действия аналогичный. Когда в операторе есть только один знак равенства, он не проводит строгое сравнение и возвращает результат `false`, если приходит к выводу, что переменные равны. Оператор, у которого два знака равенства, проверяет строгое несоответствие:

```
console.log(x !== y);
```

Пока у x и y типы данных разные, операнды не равны, поэтому выводится значение true.

Больше и меньше

Оператор «больше» возвращает значение true, если левая часть уравнения больше правой. Для обозначения этого оператора используется знак >. Есть также оператор «больше либо равно», >=, который возвращает true, если левая часть больше или равна правой части.

```
let x = 5;
let y = 6;
console.log(y > x);
```

В консоли будет зафиксировано значение true, потому что y больше, чем x.

```
console.log(x > y);
```

Пока значение x будет меньше, чем значение y, это выражение будет возвращать значение false.

```
console.log(y > y);
```

Значение y не больше y, поэтому результатом будет false.

```
console.log(y >= y);
```

Этот оператор проверяет, будет ли y больше или равен y; пока это условие соблюдается, значение будет true.

Логично, что есть также операторы «меньше» (<) и «меньше либо равно» (<=). Они очень похожи на предыдущие.

```
console.log(y < x);
```

Первый оператор будет выдавать результат false, пока y больше x.

```
console.log(x < y);
```

Второй выдает true, потому что x меньше y.

```
console.log(y < y);
```

Значение y не меньше y, поэтому результатом будет false.

```
console.log(y <= y);
```

Этот оператор проверяет, будет ли y меньше или равен y. Переменные y равны, поэтому результатом будет true.

Логические операторы

Логические операторы используются всякий раз, когда нужно проверить два условия в одном или отменить условие. Можно использовать *«И»*, *«ИЛИ»* и *«НЕ»*.

«И» (&&)

Рассмотрим первый оператор «И». Если вы хотите проверить, что `x` действительно больше `y` и `y` больше `z`, вы должны суметь объединить эти два выражения. Это можно реализовать с помощью оператора `&&`. Он вернет значение `true`, только если оба выражения правдивы:

```
let x = 1;
let y = 2;
let z = 3;
```

Используя эти переменные, мы рассмотрим логические операторы:

```
console.log(x < y && y < z);
```

Это выражение выдаст значение `true`. Его можно расшифровать как «если `x` меньше `y` и `y` меньше `z`, результат будет `true`». Выражение верно, поэтому мы и получим `true`. Следующий пример выдаст результат `false`:

```
console.log(x > y && y < z);
```

Поскольку значение `x` не превышает `y`, одна часть выражения не соответствует действительности, это приведет к значению `false`.

«ИЛИ» (||)

Если вы хотите получить значение `true`, даже если всего одно из выражений истинно, стоит использовать «ИЛИ». Оператор «ИЛИ» выглядит так — `||`. Данные прямые линии используются для проверки того, является ли одно из двух представленных выражений истинным — в таком случае все выражение оценивается как `true`. Рассмотрим оператор «ИЛИ» в действии:

```
console.log(x > y || y < z);
```

Выражение приведет к результату `true`, в то время как при использовании `&&` оно принимало значение `false`. Это связано с тем, что только одна из двух частей должна быть истинной, чтобы все выражение оценивалось как `true`. В данном случае `y` действительно меньше `z`.

Если обе части выражения неверны, то будет возвращено значение `false`, как в следующем случае:

```
console.log(x > y || y > z);
```

«НЕ» (!)

В некоторых случаях вам потребуется отрицать логическую переменную. Это действие изменит текущее значение на противоположное. Отрицание можно реализовать с помощью восклицательного знака, который означает «не»:

```
let x = false;
console.log(!x);
```

Результат в консоли будет регистрироваться как `true`, поскольку оператор просто изменит логическое значение на противоположное. Вы также можете отрицать выражение, которое вычисляется как логическое значение, но вам придется сначала его сгруппировать, дабы убедиться, что оно вычисляется в первую очередь.

```
let x = 1;
let y = 2;
console.log(!(x < y));
```

Значение `x` меньше значения `y`, поэтому выражение считается `true`. Но оно отрицается восклицательным знаком — поэтому на экран будет выведен результат `false`.

Проекты текущей главы

Конвертер миль в километры

Создайте переменную, содержащую значение в милях, преобразуйте ее в километры и запишите значение в километрах в следующем формате:

```
Расстояние 130 миль. равно 209,2142 км
```

Вычислитель индекса массы тела (ИМТ)

Установите значения для роста в дюймах и веса в фунтах, затем преобразуйте значения в сантиметры и килограммы:

- 1 дюйм равен 2,54 см;
- 2,2046 фунта равны 1 кг.

Выведите результаты. Затем рассчитайте и запишите ИМТ: он равен весу (в килограммах), деленному на квадрат роста (в метрах). Выведите результаты на экран.

Вопросы для самопроверки

1. К какому типу данных относится следующая переменная?

```
const c = "5";
```

2. Какой тип данных представляет переменная ниже?

```
const c = 91;
```

3. Какая из строк обычно считается лучшей, строка 1 или строка 2?

```
let empty1 = undefined; //строка 1
let empty2 = null; //строка 2
```

4. Какой результат после выполнения следующего кода будет выведен на экран?

```
let a = "Hello";
a = "world";
console.log(a);
```

5. Какой результат выводится в консоли?

```
let a = "world";
let b = `Hello ${a}!`;
console.log(b);
```

6. Каково значение `a`?

```
let a = "Hello";
a = prompt("world");
console.log(a);
```

7. Какое значение `b` будет выведено на экран?

```
let a = 5;
let b = 70;
let c = "5";
b++;
console.log(b);
```

8. Каково значение `result`?

```
let result = 3 + 4 * 2 / 8;
```

9. Каковы значения `total` и `total2`?

```
let firstNum = 5;
let secondNum = 10;
firstNum++;
```

```
secondNum--;
let total = ++firstNum + secondNum;
console.log(total);
let total2 = 500 + 100 / 5 + total--;
console.log(total2);
```

10. Какой результат вы увидите на экране?

```
const a = 5;
const b = 10;
console.log(a > 0 && b > 0);
console.log(a == 5 && b == 4);
console.log(true || false);
console.log(a == 3 || b == 10);
console.log(a == 3 || b == 7);
```

Резюме

В этой главе мы рассмотрели первых два строительных блока программирования: переменные и операторы. Переменные — это поля, у которых есть имена и значения. Мы объявляем переменные с помощью определяющих слов: `let`, `var` или `const`. Переменные позволяют нам делать скрипты динамичными, сохранять значения, чтобы позже получать к ним доступ и изменять их. Мы обсудили некоторые примитивные типы данных, включая `String`, `Number`, `Boolean` и `Symbol`, а также более абстрактные типы, такие как `undefined` и `null`. Вы узнали, как определить тип переменной с помощью оператора `typeof`. Рассмотрели, как преобразовать тип данных с помощью встроенных методов JavaScript: `Number()`, `String()` и `Boolean()`.

Затем мы перешли к обсуждению операторов. Операторы позволяют нам работать с переменными. Их можно использовать для выполнения вычислений, сравнения переменных и многого другого. В рассмотренный нами перечень операторов входили арифметические операторы, операторы присваивания, операторы сравнения и логические операторы.

Теперь вы готовы к знакомству с более сложными типами данных, такими как массивы и объекты. Мы перейдем к их рассмотрению в следующей главе.

3 Множественные значения JavaScript

В предыдущей главе вы познакомились с основными типами данных. Настало время изучить немного более сложную тему: массивы (arrays) и объекты (objects). Ранее вы встречались с переменными, которые содержали только одно значение. Объекты и массивы могут содержать несколько значений — благодаря этому можно писать более сложный программный код.

Объекты можно представить как набор свойств и методов. Свойства — это наши переменные. Это могут быть простые структуры, такие как числа и строки, но также и более сложные. Методы совершают действия. Они содержат определенное количество строк кода, который будет выполняться при вызове метода. Более подробно методы будут изучены позже в этой книге, а пока сосредоточимся на свойствах. Примером объекта может быть что-то реальное, например собака. У нее есть такие свойства, как кличка, вес, цвет и порода.

Мы также рассмотрим массивы. Массив — это такой тип объекта, который позволяет хранить множество значений. Массивы немного похожи на списки. Например, вы идете в продуктовый магазин с перечнем, который может включать следующие пункты: яблоки, яйца и хлеб. Данный список — это одна переменная, содержащая несколько значений.

Глава затрагивает следующие темы:

- массивы и их свойства;
- методы работы с массивами;
- многомерные массивы;
- объекты JavaScript;
- работу с объектами и массивами.

Давайте начнем с массивов.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Массивы и их свойства

Массивы — это перечни значений. Значения могут быть всех типов, один массив может содержать значения разных типов. Зачастую очень удобно хранить множество значений в одной переменной, например список студентов или продуктов, результаты теста. Как только вы начнете писать скрипты, вам придется очень часто создавать массивы. Скажем, когда вы захотите отслеживать все вводимые данные или когда понадобится список опций для представления пользователю.

Создание массивов

Возможно, мы вас убедили, что массивы — это здорово, поэтому рассмотрим, как их создавать. Есть правильный и неправильный способ. Мы представим вам оба. Как считаете, какая из следующих записей лучшая?

```
arr1 = new Array("purple", "green", "yellow");
arr2 = ["black", "orange", "pink"];
```

Если вы выбрали второй вариант, с использованием квадратных скобок, — вы правы. Это самый читаемый способ записи для создания массива. С другой стороны, первый вариант может дать неожиданные результаты. Взгляните на обе строки кода. Как думаете, каким будет итог?

```
arr3 = new Array(10);
arr4 = [10];
```

Вероятно, вы чувствуете, что здесь что-то не так. Массив с одним значением `10` получится не во всех вариантах. Нужный результат даст только вторая запись, `arr4`. Первый вариант даст массив с десятью неопределенными значениями. Если логировать значения следующим образом:

```
console.log(arr3);
console.log(arr4);
```

то вы увидите:

```
[ <10 empty items> ]
[ 10 ]
```

Спасибо, JavaScript! Это было крайне полезно. Делаем вывод: если только вашей целью не является получение массива с неопределенными значениями, используйте квадратные скобки!

Как вы помните, у нас могут быть смешанные массивы, ведь массивы позволяют хранить переменные любого типа. Значения массива не будут преобразованы в один тип данных или что-то в этом роде. JavaScript просто хранит все переменные с их собственным типом данных и значением в массиве:

```
let arr = ["hi there", 5, true];
console.log(typeof arr[0]);
console.log(typeof arr[1]);
console.log(typeof arr[2]);
```

Данный код вернет в консоль:

```
string
number
boolean
```

Еще одна интересная штука о массивах, которую мы здесь рассмотрим, — это результат определения массива с помощью `const`. Вы можете изменить значения констант массива, но не можете изменить сам массив. Фрагмент кода для демонстрации:

```
const arr = ["hi there"];
arr[0] = "new value";
console.log(arr[0]);

arr = ["nope, now you are overwriting the array"];
```

Значение первого элемента массива увеличивается, но вы не можете назначить новое значение всему массиву. Итак, вы увидите:

```
new value
TypeError: Assignment to constant variable.
```

Доступ к элементам

Прекрасный массив, который мы только что создали, стал бы намного мощнее, если бы мы могли получить доступ к его элементам. Мы можем сделать это, обратившись к индексу массива. Мы его сами не указывали, но нам это и не было нужно. JavaScript присваивает индекс каждому значению в массиве данных. Первое значение имеет индекс 0, второе — 1, третье — 2 и т. д. Если необходимо вызвать определенное значение по его положению в массиве, можно указать имя массива, добавить квадратные скобки в конце и вписать в них индекс, к которому мы хотим получить доступ. Например:

```
cars = ["Toyota", "Renault", "Volkswagen"];
console.log(cars[0]);
```

Этот оператор зафиксирует значение `Toyota` и выведет его на экран, потому что мы вызвали позицию 0 в массиве. В ней содержится первое значение списка.

```
console.log(cars[1]);
```

Вызов позиции с индексом 1 вернет нам значение второго элемента массива — `Renault`. Это и будет выведено на экран.

```
console.log(cars[2]);
```

Третьему элементу присвоен индекс 2, поэтому будет выведено его следующее значение — `Volkswagen`. Как думаете, что произойдет, если мы используем отрицательное значение индекса или значение индекса, которое превышает количество существующих значений?

```
console.log(cars[3]);
console.log(cars[-1]);
```

Мы не присваивали значение отрицательному или несуществующему индексу, поэтому, когда мы его запрашиваем, значение не определяется. Следовательно, мы получим результат `undefined`. JavaScript в этом случае не выбросит ошибку.

Перезапись элементов

Элементы массива можно перезаписать. Это можно осуществить следующим образом: запросите доступ к конкретному элементу массива, указав его индекс, и присвойте ему новое значение:

```
cars[0] = "Tesla";
console.log(cars[0]);
```

В консоли появится `Tesla`, потому что мы переписали старое значение `Toyota`. Если вывести все данные массива:

```
console.log(cars);
```

результат будет следующим:

```
[ 'Tesla', 'Renault', 'Volkswagen' ]
```

Что произойдет, если вы попытаетесь перезаписать данные элемента, которого не существует?

```
cars[3] = "Kia";
```

Или элемента с отрицательным значением индекса?

```
cars[-1] = "Fiat";
```

Давайте попытаемся вывести данные массива на экран:

```
console.log(cars[3]);
console.log(cars[-1]);
```

Получим следующий вывод:

```
Kia
Fiat
```

Ха! Они внезапно стали существовать. Вы могли такое предположить? Почему так произошло, обсудим в следующем разделе. А пока просто держите в голове, что этот способ добавления значений в массив — неправильный. Мы объясним, как сделать лучше, в разделе «Методы работы с массивами» далее.

Встроенное свойство длины

Массивы обладают очень полезным свойством: длиной. Она возвращает количество значений, которые содержит массив:

```
colors = ["black", "orange", "pink"]
booleans = [true, false, false, true];
emptyArray = [];

console.log("Length of colors:", colors.length);
console.log("Length of booleans:", booleans.length);
console.log("Length of empty array:", emptyArray.length);
```

Первый вызов `console.log` вернет значение `3`, указывая, что массив цветов содержит три значения. Второй выдаст значение `4`, а последний — это пустой массив со значением длины `0`:

```
Length of colors: 3
Length of booleans: 4
Length of empty array: 0
```

Имейте в виду, что длина на единицу превышает максимальный индекс (потому что индекс массива начинается с 0). Но при определении длины мы оцениваем количество элементов — и видим четыре отдельных элемента. Поэтому максимальный индекс — 3 при длине массива 4. Следовательно, позиционное значение последнего элемента в массиве будет на единицу меньше, чем общее количество элементов.

Остановитесь на минутку и попытайтесь понять, как вы можете использовать длину для доступа к последнему элементу массива:

```
lastElement = colors[colors.length - 1];
```

Вы получите наивысший индекс, вычитая 1 из длины, потому что, как вы знаете, массивы индексируются, начиная с нулевого значения. Таким образом, позиционное значение последнего элемента в массиве будет на единицу меньше, чем общее количество элементов.

Кажется, все довольно просто. Помните, мы брали несуществующую индексную позицию в предыдущем разделе? Посмотрим, что произойдет в данном примере:

```
numbers = [12, 24, 36];
numbers[5] = 48;
console.log(numbers.length);
```

Длина массива учитывает только целые числа, начиная с 0 и заканчивая самым высоким значением индекса заполненной ячейки. Если в середине последовательности есть элементы, которым не присвоено значение, они все равно будут подсчитаны. В таком случае значение длины становится равным `6`. Если логируете массив, вы увидите почему:

```
console.log("numbers", numbers);
```

Результат будет следующим:

```
numbers [ 12, 24, 36, <2 empty items>, 48 ]
```

Поскольку мы добавили значение `48` в позицию с индексом 5, массив создал два пустых элемента с индексами 3 и 4. А теперь давайте рассмотрим методы работы с массивами и определим правильный способ добавления значений в массив.

Практическое занятие 3.1

1. Создайте массив — список покупок, состоящий из трех единиц: `Milk`, `Bread` и `Apples`.
2. Проверьте длину списка в консоли.
3. Обновите данные второй ячейки и замените `Bread` на `Bananas`.
4. Выведите весь список на экран.

Методы работы с массивами

Только что мы рассмотрели встроенное свойство массивов — длину. У нас также есть несколько встроенных методов. Методы — это функции для определенного объекта. Они не удерживают значение (например, свойства), а выполняют действия. Более подробно мы рассмотрим функции в главе 6. А пока нужно знать только то, что вы можете вызывать методы и функции и, когда вы это делаете, выполняется некоторый код, записанный внутри этих функций.

Ранее мы заметили, что можем добавлять элементы, используя новые индексы. Это неправильный путь: так легко совершить ошибку и случайно перезаписать

определенное значение или пропустить определенный индекс. Правильнее всего будет добавить элементы с помощью специального метода. Таким же способом мы можем удалять элементы в массиве и сортировать их.

Добавление и замена элементов

Мы можем добавлять элементы с помощью метода `push()`:

```
favoriteFruits = ["grapefruit", "orange", "lemon"];
favoriteFruits.push("tangerine");
```

Значение добавляется в конец массива. Метод `push` возвращает новую длину массива (в данном случае `4`). Вы можете сохранить ее в переменной типа этой:

```
let lengthOfFavoriteFruits = favoriteFruits.push("lime");
```

В переменной `lengthOfFavoriteFruits` сохранилось значение `5`. Если мы логируем `favoriteFruits`:

```
console.log(favoriteFruits);
```

новый массив будет выглядеть следующим образом:

```
[ 'grapefruit', 'orange', 'lemon', 'tangerine', 'lime' ]
```

Совсем не трудно, правда? Но что, если вы захотите добавить элементы с определенным индексом? В таком случае необходимо использовать метод `splice()`. Он немного сложнее:

```
let arrOfShapes = ["circle", "triangle", "rectangle", "pentagon"];
arrOfShapes.splice(2, 0, "square", "trapezoid");
console.log(arrOfShapes);
```

После этой операции вывод будет выглядеть так:

```
[
 'circle',
 'triangle',
 'square',
 'trapezoid',
 'rectangle',
 'pentagon'
]
```

Сначала обратим внимание на то, что значения в консоли вывелись не в строку, а в столбец. Все зависит от используемого вами интерпретатора: в какой-то момент он решит, что массив слишком длинный для одной строки, и применит автоформатирование, чтобы сделать массив более читабельным. Значение массива при этом не изменится — это просто другая форма представления одних и тех же значений.

Как вы могли заметить, квадрат (square) и трапеция (trapezoid) вставляются в индекс 2. Остальная часть массива смещается вправо. Метод `splice()` использует несколько параметров. Первый параметр (в нашем случае `2`) — это индекс массива, с которого мы хотим начать делать вставку. Второй параметр (в нашем случае `0`) — это число элементов, которые мы хотим удалить, начиная с ранее определенного стартового значения. После этих двух параметров следуют значения (в нашем случае `square` и `trapezoid`), которые требуется внести, начиная с определенного индекса.

Если вместо этого мы введем:

```
arrOfShapes.splice(2, 2, "square", "trapezoid");
console.log(arrOfShapes);
```

программа удалит элементы со значениями `rectangle` и `pentagon` и добавит вместо них значения `square` и `trapezoid`:

```
[ 'circle', 'triangle', 'square', 'trapezoid' ]
```

Если увеличить второй параметр до числа, превышающего размер массива, результат не изменится. В этом случае JavaScript просто остановится, как только у него закончатся значения для удаления. Попробуйте ввести следующий код:

```
arrOfShapes.splice(2, 12, "square", "trapezoid");
console.log(arrOfShapes);
```

Его выполнение приведет к аналогичному выводу:

```
[ 'circle', 'triangle', 'square', 'trapezoid' ]
```

Вы также можете добавить массив в существующий массив. Для этого необходимо использовать метод `concat()`. Так у вас появится новый массив — объединение обоих массивов. Элементы первого массива будут первыми, а элементы метода `concat()` будут собраны к концу:

```
let arr5 = [1, 2, 3];
let arr6 = [4, 5, 6];
let arr7 = arr5.concat(arr6);
console.log(arr7);
```

Результат будет следующим:

```
[ 1, 2, 3, 4, 5, 6 ]
```

Но метод `concat()` способен на большее! Мы также можем использовать его для добавления значений. Мы можем добавить одно значение или разделить несколько значений запятыми:

```
let arr8 = arr7.concat(7, 8, 9);
console.log(arr8);
```

Новый вывод массива будет выглядеть так:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Удаление элементов

Есть несколько способов удалить элементы из массива. Убрать последний элемент можно с помощью метода `pop()`:

```
arr8.pop();
```

Логирование массива после выполнения команды `pop()` приводит к следующей записи:

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

Удаление первого элемента производится с помощью метода `shift()`. Это приводит к уменьшению всех оставшихся индексов на единицу:

```
arr8.shift();
```

Новый массив выглядит так:

```
[ 2, 3, 4, 5, 6, 7, 8 ]
```

Помните метод `splice()`? Это особенный метод, который также может использоваться для удаления значений. Мы указываем индекс, с которого хотим начать удаление, а затем количество элементов, которые нужно удалить.

```
arr8.splice(1, 3);
```

После этой операции значение массива будет следующим:

```
[ 2, 6, 7, 8 ]
```

Как видите, три элемента, начиная со второй позиции, были удалены. Значения `3`, `4` и `5` исчезли. Если не хотите изменять какие-либо из следующих далее индексов, вы можете использовать оператор `delete`. Это не метод, но он также способен изменить значение определенной позиции массива на `undefined`:

```
delete arr8[0];
```

Массив примет такой вид:

```
[ <1 empty item>, 6, 7, 8 ]
```

Это полезно, когда ваши действия по какой-либо причине зависят от индекса или длины массива. Например, если вы сохраняете данные, введенные пользователем,

и хотите определить количество вводов на основе массива, к которому пользователь обращается. В этом случае удаление приведет к уменьшению количества вводов и, скорее всего, вы получите не тот результат, который вам требовался.

Поиск элементов

Если вам требуется проверить, присутствует ли значение в массиве, вы можете использовать метод `find()`. Он работает несколько иначе. На самом деле это функция, которая будет применяться к каждому элементу массива, пока не найдет совпадения. Если совпадений не встретится, функция вернет значение `undefined`.

Не волнуйтесь, если сейчас это кажется слишком сложным. Все станет ясно достаточно скоро. В следующем фрагменте кода мы записываем функцию двумя разными способами. На самом деле обе строки делают одно и то же, только первая ищет элемент, равный `6`, а вторая — элемент, равный `10`:

```
arr8 = [ 2, 6, 7, 8 ];
let findValue = arr8.find(function(e) { return e === 6});
let findValue2 = arr8.find(e => e === 10);
console.log(findValue, findValue2);
```

Оператор вывода выдает `6` и `undefined`, потому что функция нашла число, равное `6`, но не нашла совпадений для числа `10`.

Функция может принимать некоторые входные данные. В нашем случае за входные данные будет принят элемент массива. Если элемент массива равен `6` (`findValue`) или `10` (`findValue2`), функция вернет это значение. Более детально функции рассматриваются в главе 6. Это очень серьезная тема для новичка, так что вы можете вернуться к ней немного позже, если пока вам что-то кажется неясным.

Часто вам понадобится не только найти элемент, но и узнать, на какой позиции он находится. Это можно реализовать с помощью метода `indexOf()`. Он возвращает значение индекса, по которому найдено значение. Если значение встречается в массиве несколько раз, метод вернет индекс первого совпадения. Если значение не найдено, вернется значение `-1`:

```
arr8 = [ 2, 6, 7, 8 ];
let findIndex = arr8.indexOf(6);
let findIndex2 = arr8.indexOf(10);
console.log(findIndex, findIndex2);
```

Таким образом, первый метод вернет `1` — это индекс позиции со значением `6`. Второй вернет значение `–1`, потому что в массиве нет числа `10`.

Чтобы узнать, где еще находится конкретное значение, помимо первого совпадения, можете добавить второй аргумент `indexOf()`, указав, с какой позиции он должен начать поиск:

```
arr8 = [ 2, 6, 7, 8 ];
let findIndex3 = arr8.indexOf(6, 2);
```

В этом случае значение `findIndex3` будет равно `-1`, поскольку `6` не может быть найдено, если поиск начнется со значения индекса 2.

Мы также можем найти последнее совпадение — с помощью метода `lastIndexOf()`:

```
let animals = ["dog", "horse", "cat", "platypus", "dog"];
let lastDog = animals.lastIndexOf("dog");
```

Значение `lastDog` будет равно `4` — это последнее совпадение значения `dog` в массиве.

Сортировка

Существует также встроенный метод сортировки массивов. Он сортирует числа от маленьких до больших и располагает строки от A к Z. Вы можете вызвать `sort()` для массива — и порядок значений массива изменится в зависимости от вида сортировки:

```
let names = ["James", "Alicia", "Fatiha", "Maria", "Bert"];
names.sort();
```

После операции сортировки массив имен будет выглядеть следующим образом:

```
[ 'Alicia', 'Bert', 'Fatiha', 'James', 'Maria' ]
```

Теперь элементы массива расположены в алфавитном порядке. Что касается чисел, то они сортируются в порядке возрастания, как вы можете видеть в следующем фрагменте кода:

```
let ages = [18, 72, 33, 56, 40];
ages.sort();
```

После исполнения метода `sort()` значения возрастов будут упорядочены:

```
[ 18, 33, 40, 56, 72 ]
```

Метод reverse

Элементы массива можно поменять местами, вызвав встроенный метод `reverse()`. Он поставит последний элемент первым, а первый элемент — последним. Не имеет значения, отсортирован массив или нет; метод просто поменяет порядок.

Вид массива перед изменением порядка элементов:

```
[ 'Alicia', 'Bert', 'Fatiha', 'James', 'Maria' ]
```

Теперь вызовем метод `reverse()`:

```
names.reverse();
```

Новый порядок элементов будет таким:

```
[ 'Maria', 'James', 'Fatiha', 'Bert', 'Alicia' ]
```

Практическое занятие 3.2

1. Создайте массив — список покупок в продуктовом магазине.
2. Добавьте в список `Milk`, `Bread` и `Apples`.
3. Замените `Bread` на `Bananas` и `Eggs`.
4. Удалите последний элемент из массива и выведите его на экран.
5. Отсортируйте список в алфавитном порядке.
6. Найдите и выведите значение индекса `Milk`.
7. После `Bananas` добавьте `Carrots` и `Lettuce`.
8. Создайте новый список, в котором будет `Juice` и `Pop`.
9. Объедините оба списка, добавив новый список дважды в конец первого списка.
10. Получите последнее значение индекса `Pop` и выведите его на экран.
11. Итоговый результат должен быть таким:

```
["Bananas", "Carrots", "Lettuce", "Eggs", "Milk", "Juice",
"Pop", "Juice", "Pop"]
```

Многомерные массивы

Ранее мы установили, что массивы могут содержать любые типы данных. Это означает, что массивы также могут содержать другие массивы (которые, в свою очередь, могут содержать... другие массивы!). Это называется *многомерным массивом*. Звучит сложно, но перед вами всего лишь массив массивов — список списков:

```
let someValues1 = [1, 2, 3];
let someValues2 = [4, 5, 6];
let someValues3 = [7, 8, 9];

let arrOfArrays = [someValues1, someValues2, someValues3];
```

Таким образом, мы можем создать массив из существующих массивов — он будет называться двумерным. Записать его можно так:

```
let arrOfArrays2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

Если потребуется получить доступ к элементам внутренних массивов, то указывать индекс придется дважды:

```
let value1 = arrOfArrays[0][1];
```

Оператор охватит первый массив — индекс 0. Из этого первого массива он примет второе значение (индекс 1). Затем он сохранит это значение в переменной `value1` — так значение `value1` станет равно `2`. Можете ли вы определить, каким будет значение `value2`?

```
let value2 = arrOfArrays[2][2];
```

Этот оператор охватывает третий массив, и из этого третьего массива он получает третье значение. Таким образом `value2` будет присвоено значение `9`. Но он не останавливается на достигнутом. Оператор может идти на много уровней в глубину. Рассмотрим это, создав массив из нашего массива массивов, — просто запишем этот массив три раза в другом массиве:

```
arrOfArraysOfArrays = [arrOfArrays, arrOfArrays, arrOfArrays];
```

Вот как выглядит массив с точки зрения значений:

```
[
 [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
 [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
 [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
]
```

Извлечем средний элемент этого массива со значением `5`, принадлежащим второму массиву массивов. Это делается так:

```
let middleValue = arrOfArraysOfArrays[1][1][1];
```

Первый шаг — получить второй массив массивов, поэтому используем индекс 1. Затем следует достать второй массив этого массива, тоже хранящийся под индексом 1. Теперь мы достигли уровня значений; нам нужно второе значение, поэтому мы снова используем индекс 1. Данная схема действий применяется во многих ситуациях: например, когда требуется работать с матрицами.

Практическое занятие 3.3

1. Создайте массив, содержащий три значения: `1`, `2` и `3`.
2. Трижды вложите исходный массив в новый массив.
3. Выведите на экран значение `2` из любого массива.

Объекты в JavaScript

Пришло время рассмотреть другую сложную структуру данных, которая может содержать более одного значения: объекты! Объекты очень полезны и могут использоваться для описания реальных предметов, а также более сложных абстрактных понятий, которые обеспечивают большую гибкость в коде.

По секрету, с объектами вы уже на самом деле знакомы: массивы — это особенный тип объектов, объектов с индексированными свойствами. Все остальные объекты (а также объекты, которые мы рассмотрим в данном разделе) являются объектами с именованными свойствами. Это значит, что вместо автоматически сгенерированного индексного номера мы присвоим им пользовательские описательные имена.

Следующий код подтверждает, что тип массивов определяется JavaScript как объектный:

```
let arr = [0, 1, 2];
console.log(typeof arr);
```

Результат логирования этого кода выглядит следующим образом:

```
Object
```

Объекты не так уж сильно отличаются от предметов реального мира. У них *есть* свойства, и они могут *выполнять* действия, методы. Ниже мы будем иметь дело только со свойствами. Методы появятся в главе 7, сразу после функций. Объект — это возможность сгруппировать несколько переменных в одну, что делается с помощью фигурных скобок: `{` и `}`. Давайте рассмотрим следующий объект — собаку:

```
let dog = { dogName: "JavaScript",
            weight: 2.4,
            color: "brown",
            breed: "chihuahua",
            age: 3,
            burglarBiter: true
          };
```

Мы создали переменную dog и присвоили этому объекту значение. Распознать, что это объект, можно по фигурным скобкам: { и }. Между ними находится множество свойств объекта и их значений.

Если вы когда-нибудь задавались вопросом, является ли что-то свойством или нет, просто взгляните на следующее шаблонное предложение:

*у *название_объекта* есть *название_свойства**

Например, у собаки *есть* имя, у собаки *есть* цвет и у собаки *есть* вес. Логические свойства определяются по-другому: вместо «есть» в предложение подставляются «является» или «не является».

Мы можем получить доступ к свойствам объекта по очень похожей на случай с массивом схеме. Но на этот раз, чтобы получить значение, мы используем не номер индекса, а название свойства:

```
let dogColor1 = dog["color"];
```

Есть и другой способ прийти к такому результату. Вместо использования квадратных скобок можно записать имя объекта и имя свойства через точку:

```
let dogColor2 = dog.color;
```

Такой формат может показаться знакомым. Помните, как мы получили длину массива со встроенным свойством length? Да — точно так же! Разница между свойствами и методами заключается в отсутствии у первых круглых скобок.

Обновление объектов

Мы можем изменять значения свойств объектов. Опять же с массивами мы работали похожим образом, потому что массив также является объектом, но, что касается свойств, у нас есть два варианта:

```
dog["color"] = "blue";
dog.weight = 2.3;
```

Код изменил свойства нашей чихуахуа. Цвет стал синим, а сама собака немного потеряла в весе (на 0,1). Если вывести данные о собаке:

```
console.log(dog);
```

получим такой результат:

```
{
  dogName: 'JavaScript',
  weight: 2.3,
  color: 'blue',
```

```
  breed: 'chihuahua',
  age: 3,
  burglarBiter: true
}
```

Полезно помнить, что, если мы изменим тип данных одного из свойств, например:

```
dog["age"] = "three";
```

JavaScript не воспримет это как проблему, а просто изменит все значения и тип данных в соответствии с новой ситуацией.

Еще одна деталь, которую следует отметить: теперь, чтобы сослаться на свойства объекта, мы используем литеральные значения, но мы также можем работать с переменными. Например:

```
let variable = "age";
console.log(dog[variable]);
```

Программа все равно выведет `three`, так как мы только что изменили значение возраста на три. Если изменить значение переменной на другое свойство собаки, мы получим к нему доступ:

```
variable = "breed";
console.log(dog[variable]);
```

Код выведет в консоль `chihuahua`. Когда мы обновляем значение, полученное подобным образом, все произойдет так же, как если бы мы обращались к нему с помощью литеральной строки:

```
dog[variable] = "dachshund";
console.log(dog["breed"]);
```

Эта строка вернет значение `dachshund` в консоль.

Практическое занятие 3.4

1. Создайте новый объект `myCar` для описания автомобиля. Добавьте несколько свойств, включая `make` и `model` (не ограничивайтесь только ими), и значения, характерные для любого или вашего автомобиля. Смело используйте логические значения, строки или числа.
2. Создайте переменную `color`, которая может содержать значение типа `string`. Эта переменная теперь может использоваться для ссылки на имя свойства объекта `myCar`. Затем включите переменную в квадратные скобки, чтобы присвоить новое значение цвета для `myCar`.

3. Возьмите ту же переменную и назначьте ей новое свойство типа `string`, например `forSale`. Снова используйте скобки, чтобы присвоить значение свойству `forSale`, указывающее, выставлен ли автомобиль на продажу.
4. Выведите `make` и `model` на экран.
5. Выведите значение `forSale` на экран.

Работа с объектами и массивами

При работе с объектами и массивами вы заметите, что они часто комбинируются. В последнем разделе этой главы мы рассмотрим объединение объектов и массивов, а также объектов внутри объектов.

Объекты внутри объектов

Допустим, мы хотим создать объект компании. У этой компании будет адрес. Но адрес будет отдельным объектом. Когда мы присвоим компании адрес, мы реализуем объект внутри другого объекта:

```
let company = { companyName: "Healthy Candy",
                activity: "food manufacturing",
                address: {
                  street: "2nd street",
                  number: "123",
                  zipcode: "33116",
                  city: "Miami",
                  state: "Florida"
                },
                yearOfEstablishment: 2021
              };
```

Во фрагменте кода видно, что объект `company` содержит в себе наполненный значениями объект `address`. Подобное погружение можно продолжать столько, сколько требуется.

Чтобы получить доступ к адресу или изменить одно из его свойств, мы можем использовать два подхода:

```
company.address.zipcode = "33117";
company["address"]["number"] = "100";
```

Как видите, очень похоже на работу с массивами: сначала нужно выбрать адрес, а затем пойти по уже знакомому пути, чтобы получить доступ к свойству, которое мы хотим изменить.

Массивы внутри объектов

У нашей компании может быть не одно, а несколько направлений деятельности. Чтобы указать это, мы можем просто заменить компонент `activity` из нашего предыдущего примера массивом:

```
company = { companyName: "Healthy Candy",
                activities: ["food manufacturing",
                "improving kids' health", "manufacturing toys"],
                address: {
                  street: "2nd street",
                  number: "123",
                  zipcode: "33116",
                  city: "Miami",
                  state: "Florida"
                },
                yearOfEstablishment: 2021
              };
```

Теперь мы создали массив в объекте `company` — можно просто использовать этот массив с квадратными скобками после свойства. Процесс извлечения отдельных значений из него очень похож. Второе значение массива `activities` можно получить с помощью следующего оператора:

```
let activity = company.activities[1];
```

Здесь мы называем интересующий объект `company`, после — соответствующий массив `activities`. Далее указываем индекс позиции, которую вызываем, — `1`.

Объекты внутри массивов

Очень возможно, что у нашей компании не один адрес, а несколько. Мы можем прописать это, создав массив адресных объектов. В нашем случае — массив из двух объектов:

```
let addresses = [{
    street: "2nd street",
    number: "123",
    zipcode: "33116",
    city: "Miami",
    state: "Florida"
  },
  {
    street: "1st West avenue",
    number: "5",
    zipcode: "75001",
    city: "Addison",
    state: "Texas"
  }];
```

Итак, массивы можно распознать по квадратным, а объекты — по фигурным скобкам. Название улицы первого объекта можно получить с помощью следующего оператора:

```
let streetName = addresses[0].street;
```

Здесь мы указываем требуемый массив `addresses` со ссылкой на индекс позиции в массиве — `0`, а после вызываем нужную переменную (`street`) из объекта. Да, процесс может показаться сложным, но, по сути, здесь просто меняется синтаксис, который мы применяли для извлечения переменной из массива внутри объекта в предыдущем разделе. Стоит попрактиковаться в вызове переменных из вложенных массивов и объектов, пока вы совсем с этим не освоитесь!

Объекты внутри массивов внутри объектов

Просто чтобы показать, что операция может пройти через столько уровней, сколько нам потребуется, мы собираемся предоставить нашему объекту `company` массив адресных объектов. Итак, добавим этот массив в `company`. Так, у нашей компании появится множество адресов:

```
company = { companyName: "Healthy Candy",
                    activities: [ "food manufacturing",
                    "improving kids' health",
                    "manufacturing toys"],
                    address: [{
                      street: "2nd street",
                      number: "123",
                      zipcode: "33116",
                      city: "Miami",
                      state: "Florida"
                    },
                    {
                      street: "1st West avenue",
                      number: "5",
                      zipcode: "75001",
                      city: "Addison",
                      state: "Texas"
                    }],
                    yearOfEstablishment: 2021
                  };
```

Для доступа к элементам объектов и массивов, вложенных еще глубже, мы просто применим ту же логику, что и в предыдущих разделах. Чтобы получить доступ к названию улицы первого адреса `Healthy Candy`, можно использовать следующий код:

```
let streetName = company.address[0].street;
```

Как видим, запросы на элементы объектов и массивов можно делать бесконечно.

На данный момент мы не будем еще больше усложнять задачу. Всякий раз, когда вам понадобится список чего-либо, вы будете использовать массив. Если захотите представить что-то со свойствами, имеющими описательные имена, лучше использовать объект — просто помните, что свойства объекта могут быть любого типа.

Практическое занятие 3.5

1. Создайте объект с именем `people`, содержащий пустой массив под названием `friends`.
2. Создайте три переменные, каждая из которых содержит объект, включающий имя, фамилию и значение ID вашего друга.
3. Добавьте трех друзей в массив `friend`.
4. Выведите результат на экран.

Проекты текущей главы

Управление массивом

Рассмотрите следующий массив:

```
const theList = ['Laurence', 'Svekis', true, 35, null, undefined,
{test: 'one', score: 55}, ['one', 'two']];
```

Используя методы `pop()`, `push()`, `shift()` и `unshift()`, добейтесь следующего результата в консоли:

```
["FIRST", "Svekis", "MIDDLE", "hello World", "LAST"]
```

Вы можете предпринять следующие шаги или реализовать собственный подход:

- удалите первый и последний элементы массива;
- добавьте значение `FIRST` в начало массива;
- присвойте значение `hello World` четвертому элементу;
- присвойте значение `MIDDLE` элементу с третьим индексом;
- добавьте значение `LAST` в конечную позицию массива;
- выведите результат на экран.

Каталог продукции компании

В этом проекте вы реализуете структуру данных для каталога товаров и создадите запросы для извлечения данных.

- Создайте массив для хранения перечня товаров магазина.
- Создайте три элемента, каждый из которых имеет свойства имени, модели, стоимости и количества.
- Добавьте все три объекта в основной массив с помощью соответствующего метода, а затем выведите получившийся массив на экран.
- Получите доступ к значению количества вашего третьего товара и зафиксируйте его в консоли. Поэкспериментируйте, добавляя новые элементы и получая доступ к большему числу элементов в вашей структуре данных.

Вопросы для самопроверки

1. Можете ли вы использовать `const` для изменения значения внутри массива?
2. Какое свойство массива определяет количество элементов, содержащихся в массиве?
3. Какой результат вы увидите в консоли?

```
const myArr1 = [1,3,5,6,8,9,15];
console.log(myArr1.indexOf(0));
console.log(myArr1.indexOf(3));
```

4. Как заменить второй элемент массива `myArr = [1,3,5,6,8,9,15]` на значение `4`?
5. Какой результат вы увидите на экране?

```
const myArr2 = [];
myArr2[10] = 'test'
console.log(myArr2);
console.log(myArr2[2]);
```

6. Какой результат отобразится в консоли?

```
const myArr3 = [3,6,8,9,3,55,553,434];
myArr3.sort();
myArr3.length = 0;
console.log(myArr3[0]);
```

Резюме

Итак, в этой главе мы рассмотрели массивы и объекты. Массивы — это списки значений. Это могут быть значения как одного типа, так и разных. Каждому элементу массива присваивается индекс. Значение начального индекса равно 0. Для получения доступа к элементам массива можно использовать значения индексов. Также, ссылаясь на индексы, можно удалять элементы или присваивать им новые значения.

Затем мы изучили создание массивов, содержащих другие массивы, — многомерных массивов. Чтобы получить доступ к элементам многомерного массива, вам потребуется столько индексов, сколько существует вложенных массивов.

Далее мы обратились к объектам и вы узнали, что массивы — это особый вид объектов. Объекты содержат свойства и методы. Вы изучили свойства объектов и увидели, что этим свойствам присваивается имя, с помощью которого к ним можно получить доступ и изменить значения.

В завершение главы мы рассмотрели массивы, содержащие внутри себя объекты, объекты, содержащие внутри себя массивы, и многое другое. Благодаря таким элементам мы можем создавать сложные объектные структуры, которые будут очень полезны при проектировании реальных приложений.

4 Логические операторы

До настоящего момента наш код был довольно статичным. Каждый раз, когда мы его вызывали, он выполнялся одинаково. В этой главе все изменится. Мы будем иметь дело с логическими операторами. Они позволяют прописывать в коде различные варианты развития событий. В зависимости от результата, полученного после применения определенного выражения, мы будем выполнять ту или иную ветвь кода.

Существуют различные логические утверждения — мы рассмотрим их в этой главе. Начнем с операторов `if` и `if else`. Далее познакомимся с тернарными операторами. Последним изучим оператор `switch`.

Мы рассмотрим следующие темы:

- операторы `if` и `if else`;
- операторы `else if`;
- условные тернарные операторы;
- операторы `switch`.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Операторы if и if else

Мы можем принимать решения в нашем коде, используя операторы `if` и `if else`. Логика действий очень похожа на описанную в данном шаблоне:

> *если (if) некоторые условия будут исполнены (true), в таком случае (then) будет выполнено определенное действие, в ином случае (else) выполнится другое действие.*

Например, *если (if)* на улице идет дождь, *тогда (then)* я возьму зонт, *в ином случае (else)* я не буду брать зонт с собой. Код не будет слишком сильно отличаться от этой формулировки:

```
let rain = true;

if(rain){
  console.log("** нужно взять зонт, когда я пойду на улицу **");
} else {
  console.log("** я могу оставить зонт дома **");
}
```

В этом случае значением `rain` является `true`. Код вернет на экран:

```
** нужно взять зонт, когда я пойду на улицу **
```

Но вернемся на шаг назад и взглянем на синтаксис. Мы начинаем со слова `if`. После этого получаем некое значение в скобках. То, что находится между этими скобками, будет преобразовано в логическое значение. Значение данной логической переменной `true`, то есть будет исполнен блок кода, связанный с оператором `if`. Этот блок легко распознать по фигурным скобкам.

Следующая часть является необязательной. Это блок `else`. Данный блок начинается словом `else` и исполняется только в том случае, если логическая переменная получает значение `false`. Если блока `else` не существует, а условие оценивается как ложное, программа просто перейдет к коду ниже оператора `if`.

Таким образом, будет выполнен только один из двух блоков: блок `if`, если выражение истинно, или блок `else`, если выражение ложно:

```
if(expression) {
  // код, связанный с блоком if,
  // будет выполнен, только если выражение верно
} else {
  // код, связанный с блоком else,
  // не обязательный, добавляется при необходимости
  // и будет выполнен, только если выражение ложно
}
```

Вот еще один пример. Если пользователю меньше 18 лет, на экран выводится сообщение, что в доступе отказано; в противном случае появляется сообщение, что доступ в консоль разрешен:

```
if(age < 18) {
  console.log("We're very sorry, but you can't get in under 18");
} else {
  console.log("Welcome!");
}
```

Существует распространенная ошибка кодирования, связанная с операторами `if`. Мы допустили ее в следующем фрагменте кода. Можете понять, что он делает?

```
let hobby = "dancing";

if(hobby = "coding"){
  console.log("** I love coding too! **");
} else {
  console.log("** Can you teach me that? **");
}
```

В консоли отобразится следующая надпись:

```
** I love coding too! **
```

Возможно, вы удивлены. Проблема кроется в операторе `if`: там не хватает одного знака равно. Вместо оценки состояния он присваивает элементу `hobby` значение `coding`. И затем преобразует `coding` в логическое значение, а поскольку это не пустая строка, значение станет истинным, поэтому блок `if` будет выполнен. Так что всегда помните: в этом случае следует использовать двойной знак равенства.

Давайте проверим ваши знания, выполнив упражнение.

Практическое занятие 4.1

1. Создайте переменную с логическим значением.
2. Выведите значение переменной на экран.
3. Проверьте, является ли переменная истинной, и, если да, выведите сообщение на экран, используя следующий синтаксис:

   ```
   if(myVariable){
            // действие
   }
   ```

4. Добавьте еще один оператор `if` с `!` перед переменной, чтобы проверить, является ли условие неверным, и создайте сообщение, которое будет выведено на экран в этом случае. У вас должно быть два оператора `if`: один с `!`, а другой без. Вы также можете вместо этого использовать операторы `if` и `else` — поэкспериментируйте!
5. Измените значение переменной на противоположное, чтобы увидеть, как от этого изменится результат.

Операторы else if

Разновидностью оператора `if` является оператор `if` со множеством блоков `else if`. Их применение в определенных ситуациях может быть более эффективным, потому что в этом случае всегда будет требоваться выполнение только одного или ни одного из блоков.

Вот шаблонная логика действий:

> *если (if) значение попадает в определенную категорию, то (then) произойдет определенное действие, иначе, если (else if) значение попадает в категорию, отличную от предыдущей инструкции, тогда (then) произойдет определенное действие, иначе, если (else if) значение попадает в категорию, отличную от любой из предыдущих, тогда (then) произойдет определенное действие.*

Возьмем этот шаблон, чтобы определить, какой должна быть цена билета. Если человек младше 3 лет, то вход бесплатный, иначе, если человек старше 3 лет и младше 12 лет, вход стоит 5 долларов, иначе, если человек старше 12 лет и моложе 65 лет, вход стоит 10 долларов, иначе, если человеку 65 лет и больше, вход стоит 7 долларов:

```
let age = 10;
let cost = 0;
let message;
if (age < 3) {
    cost = 0;
    message = "Access is free under three.";
} else if (age >= 3 && age < 12) {
    cost = 5;
    message ="With the child discount, the fee is 5 dollars";
} else if (age >= 12 && age < 65) {
    cost = 10;
    message ="A regular ticket costs 10 dollars.";
} else {
    cost = 7;
    message ="A ticket is 7 dollars.";
}

console.log(message);
console.log("Your Total cost "+cost);
```

Скорее всего, вам покажется, что код читается гораздо легче, чем шаблон. Это здорово! Вы уже начинаете мыслить как разработчики JavaScript.

Код выполняется сверху вниз, и будет выполнен только один из блоков. Как только встретится истинное выражение, остальные будут проигнорированы. Вот почему мы также можем оформить наш шаблон следующим образом:

```
if(age < 3){
  console.log("Access is free under three.");
} else if(age < 12) {
  console.log("With the child discount, the fee is 5 dollars");
} else if(age < 65) {
  console.log("A regular ticket costs 10 dollars.");
} else if(age >= 65) {
  console.log("A ticket is 7 dollars.");
}
```

Практическое занятие 4.2

1. Создайте запрос `prompt` для ввода данных о возрасте пользователя.
2. Преобразуйте ответ из запроса в число.
3. Объявите переменную `message`, которую вы будете использовать для хранения консольного сообщения для пользователя.
4. Если вводимый возраст равен или превышает 21 год, переменная `message` должна выводиться со значением «разрешить вход в заведение и покупку алкоголя».
5. Если вводимый возраст равен или превышает 19 лет, переменная `message` должна выводиться со значением «разрешить вход в заведение и запретить покупку алкоголя».
6. Создайте оператор `else` по умолчанию, чтобы задать переменной `message` значение «запретить вход», если ни одно из условий не является истинным.
7. Выведите значение переменной `message` на экран.

Условные тернарные операторы

В главе 2 мы решили пропустить этот очень важный оператор в соответствующем разделе и обратиться к нему здесь. В первую очередь потому, что он помогает понять оператор `if else`. Помните, у нас был унарный оператор, который назывался унарным, потому что у него был только один операнд? Вот почему тернарный — троичный — оператор носит свое название: он состоит из трех операндов. Например:

```
операнд1 ? операнд2 : операнд3;
```

`операнд1` — это выражение, которое должно быть оценено. Если выражению будет присвоено значение `true`, то будет исполнен `операнд2`. Если `false` — то `операнд3`. Вы можете прочитать знак вопроса как *then*, а двоеточие — как *else*:

```
выражение ? оператор в случае true : оператор в случае false;
```

Шаблонное утверждение будет таким:

если (if) операнд1, тогда (then) операнд2, иначе (else) операнд3.

Рассмотрим несколько примеров:

```
let access = age < 18 ? "denied" : "allowed";
```

Этот небольшой фрагмент кода присвоит значение переменной доступа — `access`. *Если (if)* возраст ниже 18, *тогда (then)* запретить доступ — `denied`, *иначе (else)* разрешить — `allowed`. Вы также можете указать действие в тернарном операторе, например:

```
age < 18 ? console.log("denied") : console.log("allowed");
```

Поначалу подобный синтаксис может сбить с толку. В этом случае шаблон логики, которую нужно воспроизвести в голове, действительно будет полезен. Вы можете использовать тернарные операторы только для очень коротких действий. В приведенных выше примерах тернарные операторы действительно облегчают чтение кода. Однако, если логика содержит несколько аргументов сравнения, вам придется использовать обычную схему `if else`.

Практическое занятие 4.3

1. Создайте переменную ID (удостоверение личности) с логическим значением.
2. Используя тернарный оператор, создайте переменную `message`, которая проверит, является ли ID действительным, и либо разрешит человеку войти в заведение, либо нет.
3. Выведите результат на экран.

Операторы switch

Операторы `if else` отлично подходят для оценки логических условий. Есть много решений, которые вы можете реализовать с их помощью, но в некоторых случаях лучше заменить `if else` оператором `switch`. Это особенно актуально при оценке более четырех или пяти значений.

Рассмотрим, чем операторы `switch` могут нам помочь и как они выглядят. Сначала взгляните на оператор `if else`:

```
if(activity === "Get up") {
  console.log("It is 6:30AM");
} else if(activity === "Breakfast") {
  console.log("It is 7:00AM");
} else if(activity === "Drive to work") {
  console.log("It is 8:00AM");
} else if(activity === "Lunch") {
  console.log("It is 12.00PM");
} else if(activity === "Drive home") {
  console.log("It is 5:00PM")
} else if(activity === "Dinner") {
  console.log("It is 6:30PM");
}
```

Данный код на основании ваших действий определяет текущее время суток. Но лучше было бы его реализовать с помощью оператора `switch`. Синтаксис оператора `switch` выглядит так:

```
switch(expression){
  case value1:
    // исполняемый код
    break;
  case value2:
    // исполняемый код
    break;
  case value-n:
    // исполняемый код
    break;
}
```

Вы можете прочитать это следующим образом:

> *если выражение равно value1, выполните код, указанный для этого случая. Если выражение равно value2, выполните код, указанный для этого случая, и т. д.*

Вот как можно переписать длинный код с `if else` гораздо более чистым способом, используя оператор `switch`:

```
switch(activity) {
  case "Get up":
    console.log("It is 6:30AM");
    break;
  case "Breakfast":
    console.log("It is 7:00AM");
    break;
  case "Drive to work":
    console.log("It is 8:00AM");
    break;
  case "Lunch":
```

```
    console.log("It is 12:00PM");
    break;
  case "Drive home":
    console.log("It is 5:00PM");
    break;
  case "Dinner":
    console.log("It is 6:30PM");
    break;
}
```

Если значение того, чем мы сейчас занимаемся, равняется `Lunch`, программа выведет на экран следующее:

```
It is 12:00PM
```

Возможно, вам интересно, что это за `break` такой в конце блоков и зачем он нужен? Если вы не напишете `break`, код выполнит следующий блок тоже. Так будет происходить с момента совпадения и до конца блока `switch` — или до тех пор, пока мы не столкнемся с оператором `break`. Вот каким бы был результат работы оператора `switch` без операторов `break` в результате ввода значения `Lunch`:

```
It is 12:00PM
It is 5:00PM
It is 6:30PM
```

Одно последнее замечание. Оператор `switch` использует строгую проверку типов (стратегия тройного равенства) для определения равенства, проверяя как значения, так и тип данных.

Случай по умолчанию

Есть одна часть `switch`, с которой мы еще не работали, — это метка случая по умолчанию, называемая `default`. Она очень похожа на часть `else` оператора `if else`. Если `switch` не находит совпадения ни с одним из случаев, но потом видит код, который предлагается выполнить по умолчанию, он выполнит именно его. Вот шаблон инструкции `switch` с вариантом по умолчанию:

```
switch(expression) { case value1:
  // исполняемый код
  break;
case value2:
  // исполняемый код
  break;
case value-n:
  // исполняемый код
  break;
default:
```

```
  // исполняемый код, если совпадения не найдены
  break;
}
```

Общепринято использовать оператор `default` в качестве последнего варианта в инструкции `switch`, но код будет работать без сбоев, даже если блок `default` расположен в середине или в начале. Но все-таки рекомендуем придерживаться привычной для всех последовательности, поскольку именно этого будут ожидать другие разработчики (и, возможно, вы сами в будущем) при работе с уже написанным кодом.

Допустим, с нашим длинным условием `if` связан оператор `else`, который выглядит следующим образом:

```
if(…) {
   // опустим, чтобы не делать код излишне длинным
} else {
  console.log("I cannot determine the current time.");
}
```

С использованием оператора `switch` код примет вид:

```
switch(activity) {
  case "Get up":
    console.log("It is 6:30AM");
    break;
  case "Breakfast":
    console.log("It is 7:00AM");
    break;
  case "Drive to work":
    console.log("It is 8:00AM");
    break;
  case "Lunch":
    console.log("It is 12:00PM");
    break;
  case "Drive home":
    console.log("It is 5:00PM");
    break;
  case "Dinner":
    console.log("It is 6:30PM");
    break;
  default:
    console.log("I cannot determine the current time.");
    break;
}
```

Если бы значение действия было чем-то, что не указано в качестве действия в коде (например, *Watch Netflix*), в консоли отобразилось бы следующее:

```
I cannot determine the current time.
```

Практическое занятие 4.4

Как уже говорилось в главе 1, JavaScript-функция `Math.random()` вернет случайное числовое значение в диапазоне от 0 до 1, включая 0 и исключая 1. Затем вы можете масштабировать его до нужного диапазона, умножив результат и используя `Math.floor()`, чтобы округлить его до ближайшего целого числа; например, для генерации случайного числа от 0 до 9:

```
// случайное число от 0 до 1
let randomNumber = Math.random();
// умножаем на 10, чтобы получить число от 0 до 10
randomNumber = randomNumber * 10;
// удаляем значки после десятичной запятой, чтобы получить целое число
RandomNumber = Math.floor(randomNumber);
```

В этом упражнении мы создадим магический шар судьбы.

1. Начнем с объявления переменной, которой присваивается случайное значение путем генерации случайного числа от 0 до 5 для шести возможных результатов. Вы можете увеличить это число по мере добавления новых результатов.
2. Создаем окно запроса, которое получает строковое значение, введенное пользователем, — его вы можете повторить в окончательном выводе.
3. Прописываем в `switch` шесть возможных ответов на вопрос пользователя, каждому из которых будет присвоено свое значение из генератора случайных чисел.
4. Создаем переменную для хранения конечного ответа — сообщения, выдаваемого в ответ на запрос пользователя. Вы можете давать разные строковые значения для каждого случая, присваивая новые значения в зависимости от величины случайной цифры.
5. Выводим вопрос пользователя и случайно выбранный ответ на экран, как только пользователь ввел какой-то текст в окне запроса.

Сочетание операторов

Иногда может понадобиться совершить одно и то же действие в нескольких разных случаях. В операторе `if` вам придется для этого указать все возможные варианты «ИЛИ» (`||`). В инструкции `switch` вы можете просто объединить случаи, написав их друг за другом следующим образом:

```
switch(grade){
  case "F":
  case "D":
```

```
    console.log("You've failed!");
    break;
  case "C":
  case "B":
    console.log("You've passed!");
    break;
  case "A":
    console.log("Nice!");
    break;
  default:
    console.log("I don't know this grade.");
}
```

Для значений `F` и `D` происходит одно и то же действие — так же как для `C` и `B`. Если значение `grade` равно либо `C`, либо `B`, на экран будет выведен следующий результат:

```
You've passed!
```

Данный вариант более читабельный, чем альтернативный оператор `if else`:

```
if(grade === "F" || grade === "D") {
  console.log("You've failed!");
} else if(grade === "C" || grade === "B") {
  console.log("You've passed!");
} else if(grade === "A") {
  console.log("Nice!");
} else {
  console.log("I don't know this grade.");
}
```

Практическое занятие 4.5

1. Создайте переменную с именем `prize` и сформируйте запрос для пользователя с просьбой значения, выбрав число от 0 до 10.
2. Преобразуйте введенный ответ в числовой тип данных.
3. Объявите переменную, которая будет использоваться для формирования всплывающего итогового сообщения, со значением `My Selection:`.
4. Используя инструкцию `switch` (и смекалку), напишите код, который, в зависимости от выбранного номера, будет говорить пользователю, какой приз он получает.
5. Используйте оператор `break`, чтобы объединить результаты и выдавать один и тот же приз для разных случаев.
6. Выведите сообщение для пользователя, объединив строки вашей переменной `prize` и строку итогового сообщения.

Проекты текущей главы

Игра в рулетку

Попросите пользователя ввести число и проверьте, больше оно, равно или меньше динамического значения числа в вашем коде. Выведите результаты на экран.

Игра «Проверь друга»

Попросите пользователя ввести имя, используя оператор `switch`, чтобы в результате вернуть подтверждение того, что пользователь является другом, если выбранное имя прописано в одном из операторов `case`. Можно добавить ответ по умолчанию о том, что вы не знаете этого человека, если имя не содержится в коде программы. Выведите результаты на экран.

Игра «Камень — ножницы — бумага»

Это игра между человеком и компьютером, в которой оба будут делать случайный выбор: камень, бумага или ножницы (в качестве альтернативы можете создать версию, использующую ввод реального игрока!). Камень бьет ножницы, бумага бьет камень, а ножницы бьют бумагу. Вы можете использовать JavaScript для создания собственной версии этой игры, прописывая логику с помощью оператора `if`. Поскольку этот проект немного сложнее, предлагаем несколько шагов.

1. Создайте массив, содержащий переменные `Rock`, `Paper`, `Scissors`.
2. Настройте переменную, которая генерирует случайное число от 0 до 2 для игрока, а затем сделайте то же самое для выбора компьютера. Число представляет собой значение индекса в массиве из трех элементов.
3. Создайте переменную, в которой будет храниться результат для пользователя. Она может показать итог случайного выбора соответствующего элемента из массива для игрока, а затем — для компьютера.
4. Создайте условие для обработки выбора игрока и компьютера. Если результаты выбора совпадают — фиксируется ничья.
5. Используйте условия, чтобы применить логику игры и вернуть пользователю правильные результаты. С помощью операторов условий вы можете это сделать несколькими способами. Подсказка: можете сравнить значение индексов выбранных вариантов и назначить победу тому, чей индекс больше. «Камень, бьющий ножницы» в таком случае будет исключением.
6. Добавьте новое выходное сообщение, в котором отобразится выбор игрока, выбор компьютера и результат игры.

Вопросы для самопроверки

1. Какой результат будет выведен на экран?

```
const q = '1';
switch (q) {
    case '1':
        answer = "one";
        break;
    case 1:
        answer = 1;
        break;
    case 2:
        answer = "this is the one";
        break;
    default:
        answer = "not working";
}
console.log(answer);
```

2. Что отобразится в консоли после выполнения данного кода?

```
const q = 1;

switch (q) {
    case '1':
        answer = "one";
    case 1:
        answer = 1;
    case 2:
        answer = "this is the one";
        break;
    default:
        answer = "not working";
}
console.log(answer);
```

3. Какой результат появится на экране?

```
let login = false;
let outputHolder = "";
let userOkay = login ? outputHolder = "logout" : outputHolder =
"login";
console.log(userOkay);
```

4. Каким будет итог выполнения этого кода?

```
const userNames = ["Mike", "John", "Larry"];
const userInput = "John";
```

```
let htmlOutput = "";
if (userNames.indexOf(userInput) > -1) {
    htmlOutput = "Welcome, that is a user";
} else {
    htmlOutput = "Denied, not a user ";
}
console.log(htmlOutput + ": " + userInput);
```

5. Какой результат будет выведен в консоль в этом примере?

```
let myTime = 9;
let output;
if (myTime >= 8 && myTime < 12) {
    output = "Wake up, it's morning";
} else if (myTime >= 12 && myTime < 13) {
    output = "Go to lunch";
} else if (myTime >= 13 && myTime <= 16) {
    output = "Go to work";
} else if (myTime > 16 && myTime < 20) {
    output = "Dinner time";
} else if (myTime >= 22) {
    output = "Time to go to sleep";
} else {
    output = "You should be sleeping";
}
       console.log(output);
```

6. Что выдаст программа после выполнения данного кода?

```
let a = 5;
let b = 10;
let c = 20;
let d = 30;
console.log(a > b || b > a);
console.log(a > b && b > a);
console.log(d > b || b > a);
console.log(d > b && b > a);
```

7. Что будет выведено на экран в этом примере?

```
let val = 100;
let message = (val > 100) ? `${val} was greater than 100` :
`${val} was LESS or Equal to 100`;
console.log(message);
let check = (val % 2) ? `Odd` : `Even`;
check = `${val} is ${check}`;
console.log(check);
```

Резюме

В этой главе мы рассмотрели условные операторы. Мы начали с операторов `if else`. Блок `if` выполняется каждый раз, когда условие, связанное с ним, является истиной. Если условие ложно и присутствует блок `else`, будет исполнен именно он. Мы также рассмотрели тернарные операторы и их необычный синтаксис. Это быстрый способ написания оператора `if else`, если вам нужен только один оператор в блоке.

И наконец, мы рассмотрели инструкции `switch` и возможности их использования для оптимизации кода с набором условий. С помощью оператора `switch` мы можем сравнить одно условие со многими различными вариантами. Когда они равны (по значению и типу), выполняется код, связанный с конкретным случаем.

В следующей главе мы добавим ко всему этому циклы! Они помогут писать более эффективный код и алгоритмы.

5 Циклы

Наша JavaScript-база становится все лучше и обширнее. В этой главе сосредоточимся на очень важной концепции управления потоком данных: циклах. Циклы выполняют блок кода определенное количество раз. Их можно использовать во многих задачах, таких как многократное повторение операций, перебор наборов данных, массивов и объектов. Всякий раз, когда вы захотите скопировать небольшой фрагмент кода и продублировать его в том же месте, откуда вы его взяли, вместо этого действия используйте цикл.

Сначала рассмотрим основы циклов, а затем обсудим вложенные циклы, которые используют циклы внутри циклов. Кроме того, мы разберем построение циклов на примере двух сложных конструкций, которые уже вам знакомы, — массивов и объектов. И наконец, мы введем два ключевых слова, связанных с циклами, — `break` и `continue`, чтобы еще лучше контролировать поток данных в цикле.

Существует метод, который тесно связан с циклами, но который мы пока не будем затрагивать, — встроенный метод foreach. Его можно использовать для перебора массивов, когда потребуется стрелочная функция. Сюда мы его не включали — но обязательно к нему вернемся в следующей главе.

Перед вами различные циклы с условиями, которые мы будем рассматривать в этой главе:

- цикл `while`;
- цикл `do while`;
- цикл `for`;
- цикл `for in`;
- цикл `for of`.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Циклы while

Первым рассмотрим цикл с условием `while`. Данный цикл выполняет определенный блок кода до тех пор, пока выражение будет иметь значение `true`. Следующий фрагмент кода показывает синтаксис цикла `while`:

```
while (condition) {
   // код, который выполняется, пока значение условия (condition) true
}
```

Цикл `while` будет выполняться, пока значение условия `true`. Если начальное условие изменится на `false`, код внутри не будет исполнен.

Вот очень простой пример цикла `while`, выводящего числа от `0` до `10` (исключая `10`) на экран:

```
let i = 0;
while (i < 10) {
   console.log(i);
   i++;
}
```

Результат будет таким:

```
1
2
3
4
5
6
7
8
9
```

Ниже описаны шаги, которые выполнялись в коде.

1. Создается переменная `i`, которой присваивается начальное значение `0`.
2. Начинается цикл `while`, он проверяет код на соответствие условию, по которому значение `i` должно быть меньше 10.
3. Поскольку условие возвращает значение `true`, переменная `i` логируется и ее значение увеличивается на 1.
4. Условие проверяется снова; 1 тоже меньше 10.
5. Поскольку условие возвращает значение `true`, `i` выводится и ее значение увеличивается на 1.
6. Вывод и увеличение значений продолжаются до тех пор, пока `i` не примет значение `10`.
7. Десять не меньше десяти, значит, цикл закончен.

Мы можем создать цикл `while`, который будет искать нужное значение в массиве, например:

```
let someArray = ["Mike", "Antal", "Marc", "Emir", "Louiza", "Jacky"];
let notFound = true;

while (notFound && someArray.length > 0) {
   if (someArray[0] === "Louiza") {
      console.log("Found her!");
      notFound = false;
   } else {
      someArray.shift();
   }
}
```

Он проверяет, является ли первое значение массива требуемым, и, если это не так, оператор удаляет это значение из массива с помощью метода `shift`. Помните его? Да, этот метод удаляет первое значение в массиве. Таким образом, в следующей итерации цикла первое значение изменилось и проверяется снова. Если код найдет нужное значение, он выведет его на экран и изменит логическое значение переменной `notFound` на `false`, потому что обнаружил искомый элемент. Это станет последней итерацией — цикл завершится. Мы получим сообщение:

```
Found her! false
```

Как вы думаете, почему мы добавили запись `&& someArray.length > 0`? Если бы мы этого не предусмотрели, а искомого значения в массиве не нашлось, цикл стал бы бесконечным. Вот почему стоит следить за тем, чтобы цикл завершал работу, если требуемый результат отсутствует, и мы могли перейти к выполнению оставшегося кода.

С помощью циклов можно делать и более сложные вещи. Посмотрим, как легко массив заполняется последовательностью Фибоначчи:

```
let nr1 = 0;
let nr2 = 1;
let temp;
fibonacciArray = [];

while (fibonacciArray.length < 25) {
   fibonacciArray.push(nr1);
   temp = nr1 + nr2;
   nr1 = nr2;
   nr2 = temp;
}
```

В последовательности Фибоначчи каждое значение представляет собой сумму двух предыдущих, начиная со значений `0` и `1`. Мы можем описать это в цикле `while`, как указано выше. Мы создаем два числа, и они меняются на каждой итерации. Мы ограничили количество итераций длиной массива `fibonacciArray`, чтобы избежать бесконечного цикла. В нашем случае цикл будет выполнен, как только длина массива достигнет значения `25`.

Нам потребуется временная переменная, которая будет хранить следующее значение для `nr2`. Каждую итерацию мы помещаем значение первого числа в массив. Если вывести значения массива, вы увидите, что значения очень быстро становятся довольно большими. Представьте, если бы вам нужно было самим генерировать эти значения в коде одно за другим!

```
[
     0,     1,     1,     2,     3,
     5,     8,    13,    21,    34,
    55,    89,   144,   233,   377,
   610,   987,  1597,  2584,  4181,
  6765, 10946, 17711, 28657, 46368,
]
```

Практическое занятие 5.1

В этом упражнении мы создадим игру «Угадай число», которая принимает пользовательский ввод и отвечает, насколько точным было предположение.

1. Объявите переменную, которая будет использоваться в качестве максимального значения для игры.
2. Сгенерируйте случайное число (которое будет угадываемым), используя `Math.random()` и `Math.floor()`. Вам также нужно будет добавить `1`, чтобы значение возвращалось как `1 – [любое установленное максимальное значение]`. Можете при создании игры периодически выводить это значение в консоли, а затем, когда написание игры будет завершено, закомментировать его.
3. Создайте переменную, которая будет отслеживать соответствие пользовательского ввода угадываемому числу, и присвойте ей логическое значение по умолчанию `false`. Если пользователь угадает число, переменная должна вернуть значение `true`.
4. Используйте цикл `while`, чтобы создать повторяющийся запрос, который будет предлагать пользователю ввести в строке число от `1` до `5`, и преобразуйте полученные данные, чтобы они соответствовали типу случайных чисел.
5. Внутри цикла `while` с помощью условия проверьте, равно ли значение запроса угадываемому значению. Постройте логику таким образом, чтобы, когда число верно, устанавливался статус `true` и цикл завершался командой `break`. Предоставьте игроку обратную связь относительно того, было ли предположение близким к истине, и пригласите ввести новое значение. Делайте так до тех пор, пока пользователь не угадает число. Таким образом, мы используем цикл, чтобы продолжать задавать вопросы, пока решение не будет верным. В этот момент можно остановить итерацию.

Циклы do while

В некоторых случаях требуется, чтобы блок кода был выполнен хотя бы один раз. Например, при авторизации пользователя данные от него нужно запросить хотя бы однажды. Или при подключении к базе данных или к какому-либо другому внешнему источнику: вам придется сделать хотя бы одну попытку, и, вероятно, она будет успешной. Но вы можете ошибиться, и придется вводить данные до тех пор, пока не получится нужный результат. Для описания подобной логики можно использовать цикл `do while`.

Синтаксис выглядит так:

```
do {
  // исполняемый код, если значение условия true
} while (condition);
```

Здесь то, что находится в блоке `do`, сначала выполняется, а после этого оценивается (`while`). Если условие принимает значение `true`, код блока `do` снова будет выполняться. Так будет продолжаться до тех пор, пока условие блока `while` не примет значение `false`.

Для получения вводимых пользователем данных можно использовать метод `prompt()`. Давайте применим цикл `do while`, чтобы попросить пользователя ввести число от `0` до `100`.

```
let number;
do {
  number = prompt("Please enter a number between 0 and 100: ");
} while (!(number >= 0 && number < 100));
```

Вот что должно получиться в результате (цифры в данном случае вводятся в консоли вручную):

```
Please enter a number between 0 and 100: > -50
Please enter a number between 0 and 100: > 150
Please enter a number between 0 and 100: > 34
```

Все, что находится после >, — это данные, введенные пользователем. Знак > — это часть кода. Консоль добавляет его, чтобы четко различать вывод на экран (Please enter a number between 0 and 100) и введенные данные (–50, 150 и 34).

Программа делала запрос трижды, потому что в первые две попытки число не находилось между 0 и 100, а значение условия в блоке `while` было `true`. При вводе `34` условие блока `while` приняло значение `false` — и цикл завершился.

Практическое занятие 5.2

В этом упражнении мы создадим базовый счетчик, который будет увеличивать динамическую переменную на установленное значение шага, пока не достигнет верхнего предела.

1. Установите начальный счетчик на `0`.
2. Создайте переменную `step`, которая будет показывать, на сколько нужно увеличить значение счетчика.
3. Добавьте цикл `do while`, который в каждой итерации будет выводить на экран значение счетчика и увеличивать его на значение переменной `step`.
4. Цикл будет продолжаться, пока счетчик не достигнет числа `100` или больше.

Цикл for

Циклы с условием `for` — это особенные циклы. Поначалу синтаксис может показаться немного запутанным, но очень скоро вы будете использовать эти циклы постоянно — просто потому, что они очень полезны:

```
for (инициализируемая переменная; условие; оператор)
{
 // исполняемый код
}
```

Между скобками после оператора `for` записаны три элемента, разделенные точкой с запятой. Первый из них инициализирует переменные, которые могут быть использованы в цикле `for`. Второе — это условие: пока оно верно, цикл будет повторяться. Условие проверяется после инициализации переменных перед первой итерацией (которая происходит, только если условие равно `true`). Последний — это оператор. Он будет выполняться после каждой итерации цикла. Алгоритм работы цикла `for` выглядит следующим образом.

1. Инициализация переменных.
2. Проверка условия.
3. Выполнение блока кода, если значение условия `true`. Если значение условия `false`, цикл завершается.
4. Выполнение оператора (третьего элемента, например `i++`).
5. Переход к пункту 2.

Приведем простой пример, который выводит в консоль числа от `0` до `10` (исключая `10`):

```
for (let i = 0; i < 10; i++) {
   console.log(i);
}
```

Он начинается с создания переменной `i` и присвоения ей значения `0`. Далее проверяется условие: `i` меньше `10`. Если значение условия `true`, выполняется логирование. После этого вызывается оператор `i++` — и значение переменной `i` увеличивается на единицу.

Если не увеличивать значение i, мы застрянем в бесконечном цикле, потому что значение i не будет меняться и навсегда зависнет на уровне меньше 10. Обращайте внимание на этот аспект во всех циклах!

Далее условие проверяется снова. Так продолжается до тех пор, пока `i` не достигнет значения `10`. Десять не меньше десяти — цикл завершен, а на экран выведены значения от `0` до `9`.

Цикл `for` можно также использовать для создания последовательности и добавления значений в массив, например:

```
let arr = [];
   for (let i = 0; i < 100; i++) {
   arr.push(i);
}
```

Вот как выглядит массив после выполнения этого цикла:

```
[
  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
 96, 97, 98, 99
]
```

Поскольку цикл выполнял блок кода 100 раз, начиная со стартового значения `i`, равного `0`, увеличивающееся значение будет добавляться в конец массива. В результате получится массив с числами `0`–`99` и длиной 100 элементов. Поскольку массивы начинаются со значения индекса, равного `0`, наши значения в массиве фактически будут совпадать со значениями индекса элементов в массиве.

Или мы могли бы создать массив, содержащий только четные значения:

```
let arr = [];
for (let i = 0; i < 100; i = i + 2) {
   arr.push(i);
}
```

Результат выполнения:

```
[
  0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20,
 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64,
 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
 88, 90, 92, 94, 96, 98
]
```

Наиболее распространено использование оператора `i++` в качестве третьего элемента цикла `for`. Но обратите внимание: вы можете указать там любой оператор. В нашем случае мы использовали `i = i + 2`, чтобы каждый раз добавлять 2 к предыдущему значению, создавая массив только с четными числами.

Практическое занятие 5.3

В этом упражнении мы используем цикл `for` для создания массива объектов. Начнем с пустого массива; блок кода в цикле создаст объект, который будет вставлен в массив.

1. Присвойте пустому массиву имя `myWork`.
2. Используя цикл `for`, составьте список из десяти объектов, каждый из которых — это пронумерованный урок (например, `Lesson 1`, `Lesson 2`, `Lesson 3`...). Задайте чередующиеся логические значения `true`/`false` для каждого элемента, чтобы указать, будет ли занятие в этом году. Например:

   ```
   name: 'Lesson 1', status: true
   ```

3. Вы можете указать статус с помощью тернарного оператора, который проверяет, равно ли значение по модулю заданного значения урока нулю, и настраивает логические значения для чередования результатов на каждой итерации.
4. Создайте элемент `lesson`, используя временную объектную переменную, содержащую имя (`lesson` с числовым значением) и заданный статус (который мы настроили на предыдущем шаге).
5. Внесите методом `push` объекты в массив `myWork`.
6. Выведите данные массива на экран.

Вложенные циклы

Иногда вам может понадобиться цикл внутри цикла. Такой цикл называется вложенным. Зачастую это не лучшее решение проблемы. Вложенные циклы могут быть даже признаком плохо написанного кода (среди программистов его иногда называют «кодом с запашком» — code smell) — но порой это совершенно правильный выбор.

Вот как это выглядит для циклов `while`:

```
while (condition 1) {
  // код, выполняемый, пока условие 1 в значении true
  // этот цикл зависит от значения true условия 1
  while (condition 2) {
      // код, который выполняется, пока условие 2 в значении true
  }
}
```

Вложения также могут быть созданы с помощью цикла `for`, или комбинацией циклов `for` и `while`, или даже всеми видами циклов. Такие вложения могут уходить на несколько уровней в глубину.

Случай, в котором мы могли бы использовать вложенные циклы, — создание массива массивов. С помощью внешнего цикла мы прописываем массив верхнего уровня, а с помощью внутреннего цикла добавляем значения в массив.

```
let arrOfArrays = [];
for (let i = 0; i < 3; i++){
   arrOfArrays.push([]);
   for (let j = 0; j < 7; j++) {
      arrOfArrays[i].push(j);
   }
}
```

Логирование массива:

```
console.log(arrOfArrays);
```

В результате мы видим массив массивов со значениями от `0` до `6`.

```
[
  [
    0, 1, 2, 3, 4, 5, 6
  ],
  [
    0, 1, 2, 3, 4, 5, 6
  ],
  [
    0, 1, 2, 3, 4, 5, 6
  ]
]
```

Мы использовали вложенные циклы для создания массива в массиве и теперь можем работать со строками и столбцами в них. Таким образом, вложенные циклы полезны для отображения табличных данных. Мы можем сгруппировать представленный выше вывод в виде таблицы с помощью метода `console.table()`:

```
console.table(arrOfArrays);
```

Результат на экране:

```
┌─────────┬───┬───┬───┬───┬───┬───┬───┐
│ (index) │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
├─────────┼───┼───┼───┼───┼───┼───┼───┤
│    0    │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
│    1    │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
│    2    │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
└─────────┴───┴───┴───┴───┴───┴───┴───┘
```

Давайте применим это в следующем упражнении.

Практическое занятие 5.4

В этом упражнении мы создадим таблицу значений. Используем циклы для формирования строк, а также столбцов, которые будут вложены в строки. Вложенные циклы могут быть использованы для представления строк в таблице. Это обычная структура для электронных таблиц, где каждая строка представляет собой вложенный массив внутри таблицы, а содержимое этих строк — это ячейки таблицы. Поскольку мы создаем равное количество ячеек в каждой строке, столбцы будут выровнены.

1. Чтобы реализовать генератор таблиц, сначала создайте пустой массив `myTable` для хранения данных таблицы.
2. Задайте значения переменных для количества строк и столбцов — это позволит осуществлять динамический контроль за их количеством. Отделение значений от основного кода облегчает изменение параметров таблицы.
3. Задайте переменную-счетчик `counter` с начальным значением `0`. Счетчик будет использоваться для установки содержимого и подсчета значений ячеек в таблице.
4. Создайте цикл `for` с условиями, чтобы задать количество итераций и построить каждую строку таблицы. Внутри него создайте новый временный массив (`TempTable`) для хранения каждой строки данных. Столбцы будут вложены в строки, создавая таким образом каждую ячейку конкретного столбца.

5. Вложите второй цикл в первый, чтобы подсчитать столбцы. Столбцы формируются в цикле строк, чтобы их количество в таблице получилось одинаковым.
6. Увеличивайте основной счетчик на каждой итерации внутреннего цикла, чтобы отслеживать общее количество ячеек и количество созданных ячеек.
7. Перенесите значения счетчика во временный массив `tempTable`. Поскольку массив, представляющий таблицу, является вложенным, значения счетчика также можно использовать для отображения значений соседних ячеек в таблице. Хотя это отдельные массивы, представляющие новые строки, счетчик покажет общую последовательность ячеек в итоговой таблице.
8. Переместите временный массив в основную таблицу. По мере того как каждая итерация создает новую строку элементов массива, основная таблица в массиве будет продолжать увеличиваться.
9. Выведите результат на экран с помощью `console.table(myTable)`. Этот метод даст визуальное представление структуры таблицы.

Циклы и массивы

Если вы еще не убеждены в том, насколько полезны циклы, посмотрите, как они взаимодействуют с массивами. Циклы делают работу с массивами намного более комфортной.

Мы можем совместить свойство `length` и условную часть циклов `for` или `while` для зацикливания массивов. В случае с циклом `for` это будет выглядеть так:

```
let arr = [some array];
for (initialize variable; variable smaller than arr.length; statement)
{
  // исполняемый код
}
```

Начнем с простого примера, который будет выводить каждое значение массива:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i ++){
  console.log(names[i]);
}
```

На экране отобразится следующий результат:

```
Chantal
John
Maxime
Bobbi
Jair
```

Мы использовали свойство `length`, чтобы определить максимальное значение индекса. Значение индекса начинает отсчитываться с `0`, но не длина. Индекс всегда на единицу меньше длины. Следовательно, мы перебираем значения массива, увеличивая длину.

Пока мы не делаем ничего интересного. Просто выводим значения. Но мы можем начать изменять значения массива в цикле, например, так:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i ++){
  names[i] = "hello " + names[i];
}
```

Мы добавили к именам слово `hello`. Массив изменяется и после выполнения цикла получает новое значение:

```
[
 'hello Chantal',
 'hello John',
 'hello Maxime',
 'hello Bobbi',
 'hello Jair'
]
```

Возможности использования циклов и массивов поистине безграничны. Если где-то в приложении появляется массив, он позволяет, ссылаясь на значения, отправить данные в базу. Точно так же данные могут быть изменены или даже отфильтрованы:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i ++){
  if(names[i].startsWith("M")){
    delete names[i];
    continue;
  }
  names[i] = "hello " + names[i];
}
console.log(names);
```

Метод `StartsWith()` проверяет, начинается ли строка с определенного символа (в данном случае — начинается ли имя с буквы M).

Мы еще рассмотрим математические операторы в главе 8.

Так будет выглядеть результат:

```
[
  'hello Chantal',
  'hello John',
  <1 empty item>,
  'hello Bobbi',
  'hello Jair'
]
```

Однако будьте здесь осторожны. Если бы мы просто удалили элемент и не оставили пустое значение, мы бы случайно пропустили следующее значение: оно бы получило индекс удаленного фрагмента, а переменная `i` увеличилась бы на 1 и перешла к следующему индексу.

Как вы думаете, каким будет результат?

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let i = 0; i < names.length; i++){
  names.push("...")
}
```

Программа застрянет в бесконечном цикле. Поскольку значение добавляется каждую итерацию, длина цикла с каждой итерацией также увеличивается — соответственно, значение `i` никогда не будет больше или равно значению `length`.

Практическое занятие 5.5

Давайте узнаем, как создать табличную сетку, где в качестве строк таблицы будут выступать вложенные массивы. Каждая строка будет содержать столько ячеек, сколько необходимо для заданного в переменных числа столбцов. В зависимости от значений переменных эта таблица будет динамически корректироваться.

1. Создайте переменную массива сетки.
2. Числу ячеек присвойте значение `64`.
3. Счетчик установите на `0`.

4. Создайте глобальную переменную для использования в массиве `row`.
5. Создайте цикл, который будет повторяться до нужного вам количества ячеек в массиве, и добавьте единицу, чтобы включить нулевое значение (для нашего примера цифра будет следующей: `64+1`).
6. Добавьте внешний оператор `if`, который использует модуль, чтобы проверить, делится ли основной счетчик на 8 или на любое задуманное количество столбцов.
7. Внутрь оператора `if` добавьте еще один `if`, чтобы проверить наличие неопределенных строк, указывая: строка только начинает заполняться или ее формирование уже завершено. Если строка была определена, ее надо добавить в сетку главного массива.
8. Для завершения работы внешнего оператора `if`, если значение счетчика делится на 8, очистите массив `row` — он уже был добавлен в сетку значений внутренним оператором `if`.
9. В конце цикла `for` увеличьте основной счетчик на 1.
10. Задайте временную переменную для хранения значения счетчика и переместите ее в массив `row`.
11. В рамках итерации цикла проверьте, равно ли значение счетчика общему количеству нужных вам столбцов. Если да, то добавьте текущую строку в сетку значений.
12. Обратите внимание, что дополнительная ячейка не будет добавлена в сетку: в соответствии с установленным выше условием для создания новой строки этой ячейки будет недостаточно. Альтернативным решением было бы удалить `+1` из условия цикла и добавить `grid.push(row)` после завершения цикла — оба варианта обеспечат одинаковый вывод.
13. Выведите сетку значений на экран.

Цикл for of

Существует еще один цикл, который можно использовать для перебора элементов массива: цикл `for of`. С его помощью нельзя изменить значение, связанное с индексом, как получилось бы сделать в обычном цикле, но для обработки значений данная опция очень удобная и читаемая.

Синтаксис выглядит так:

```
let arr = [some array];
for (let variableName of arr) {
  // исполняемый код
```

```
  // значение variableName, обновляемое с каждой итерацией
  // всем значениям массива однажды будет присвоено имя variableName
}
```

Вы можете прочитать эту запись так: *каждое значение массива назовите variableName и выполните следующие действия*. Используя данный цикл, можно вывести значения массива `names`:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let name of names){
  console.log(name);
}
```

Нам нужно указать временную переменную. В данном случае мы назвали ее `name`. Она нужна для ввода значения текущей итерации и после итерации заменяется следующим значением. В результате выполнения кода вы получите:

```
Chantal
John
Maxime
Bobbi
Jair
```

У данного способа есть некоторые ограничения. Мы не можем изменять массив, однако у нас получилось бы записать все элементы в базу данных или файл или отправить их куда-нибудь еще. Есть и преимущество — мы не застрянем случайно в бесконечном цикле и не пропустим значения.

Практическое занятие 5.6

В данном упражнении мы создадим массив, перебирая увеличивающиеся значения `x`. После этого продемонстрируем несколько способов вывода содержимого массива.

1. Создайте пустой массив.
2. Запустите цикл десять раз, добавляя каждое новое увеличивающееся значение в массив.
3. Выведите данные массива на экран.
4. Используйте цикл `for` для итерации по массиву (установите количество итераций, равное количеству значений в вашем массиве) и выведите в консоль.
5. Используйте цикл `for` для вывода значений из массива на экран.

Циклы и объекты

Мы только что рассмотрели, как перебирать значения массива — но с помощью циклов мы также можем перебирать свойства объекта. Это может быть полезно, когда нам нужно просмотреть все свойства объекта, используемого в итерации, но мы не знаем, какие именно нам нужны.

Включить объект в цикл можно несколькими способами: непосредственно с помощью цикла `for in` или преобразовав объект в массив и работая уже с ним. Далее мы рассмотрим и то и другое.

Цикл for in

Цикл `for in` чем-то похож на цикл `for of`. Нам снова нужно создать временную переменную, также называемую ключом (key), для хранения каждого имени свойства. Как это работает:

```
let car = {
  model: "Golf",
  make: "Volkswagen",
  year: 1999,
  color: "black",
};

for (let prop in car){
  console.log(car[prop]);
}
```

Нам нужно использовать метод `prop` в каждой итерации цикла, чтобы получить значение из объекта `car`. Массив примет следующий вид:

```
Golf
Volkswagen
1999
black
```

А если логировать `prop` таким образом:

```
for (let prop in car){
  console.log(prop);
}
```

на выходе мы получим:

```
model
make
year
color
```

Как видите, названия свойств были выведены, а значения — нет. Это происходит потому, что цикл `for in` получает имена свойств (ключи), но не их значения. А цикл `for of` реализует противоположное действие: получает значения свойств, но не ключи.

Цикл `for in` можно использовать при работе с массивами, но это не самый популярный ход. Он вернет только индексы, так как это «ключи» значений массивов. Также следует отметить, что цикл не может гарантировать порядок выполнения, хотя обычно для массивов это важно. Поэтому лучше использовать подходы, упомянутые в подразделе о циклах и массивах.

Практическое занятие 5.7

В этом упражнении мы поэкспериментируем с циклическим перебором объектов и внутренних массивов.

1. Создайте простой объект, содержащий три элемента.
2. Используя цикл `for in`, получите имена и значения свойств объекта и выведите их на экран.
3. Создайте массив, содержащий те же самые три значения. Используйте цикл `for` или `for in` для вывода значений из массива на экран.

Цикл из объектов, преобразованных в массивы

Вы можете использовать любой цикл для объектов, как только преобразуете объект в массив. Реализовать подобное можно тремя способами:

- преобразовать ключи объекта в массив;
- преобразовать значения объекта в массив;
- преобразовать в массив записи и значений, и ключей (который будет содержать массивы с двумя элементами: ключом объекта и значением объекта).

Возьмем уже знакомый вам фрагмент кода:

```
let car = {
  model: "Golf",
  make: "Volkswagen",
  year: 1999,
  color: "black",
};
```

Для перебора ключей объекта можно взять цикл `for in`, как в предыдущем подразделе, но также можно использовать цикл `for of` (преобразовав перед этим

объект в массив). Реализовать второе можно с помощью встроенной функции `Object.keys(nameOfObject)`. Она берет объект, получает все свойства этого объекта и преобразует их в массив.

Работает это следующим образом:

```
let arrKeys = Object.keys(car);
console.log(arrKeys);
```

В консоли отобразится следующее:

```
[ 'model', 'make', 'year', 'color' ]
```

Далее, используя цикл `for of`, мы можем перебирать свойства этого массива:

```
for(let key of Object.keys(car)) {
  console.log(key);
}
```

Вы увидите результат:

```
model
make
year
color
```

Аналогичным образом мы можем использовать цикл `for of` для перебора значений объекта, преобразовав значения в массив. Главное отличие будет заключаться в использовании `Object.values(nameOfObject)`:

```
for(let key of Object.values(car)) {
  console.log(key);
}
```

Вы можете перебирать значения этих массивов так же, как перебираете значения любого массива. Можно использовать длину и индекс в цикле `for`, как обычно:

```
let arrKeys = Object.keys(car);
for(let i = 0; i < arrKeys.length; i++) {
  console.log(arrKeys[i] + ": " + car[arrKeys[i]]);
}
```

Результат вывода на экран:

```
model: Golf
make: Volkswagen
year: 1999
color: black
```

Более интересной задачей будет перебор значений обоих массивов одновременно с использованием цикла `for of`. Для этого нам понадобится `Object.entries()`. Посмотрим, что он делает:

```
let arrEntries = Object.entries(car);
  console.log(arrEntries);
```

Результат вывода на экран:

```
[
 [ 'model', 'Golf' ],
 [ 'make', 'Volkswagen' ],
 [ 'year', 1999 ],
 [ 'color', 'black' ]
]
```

Как видите, программа возвращает двумерный массив, содержащий пары значений-ключей. Мы можем перебрать значения следующим образом:

```
for (const [key, value] of Object.entries(car)) {
console.log(key, ":", value);
}
```

Результат вывода на экран:

```
model : Golf
make : Volkswagen
year : 1999
color : black
```

Отлично, теперь вы знаете много способов перебора данных у объектов. Большинство из них сводятся к преобразованию объекта в массив. Может показаться, что в этот момент вам не помешал бы перерыв (`break`). Или хотите продолжить (`continue`)?

Операторы break и continue

`break` и `continue` — это два ключевых слова, позволяющих лучше контролировать исполнение цикла. `break` остановит цикл и отправит программу выполнять код после цикла. `continue` остановит текущую итерацию и вернется к началу цикла, проверяя условие (или, в случае с циклом `for`, сначала выполнив оператор, а потом проверив условие).

Используем массив объектов `car`, чтобы продемонстрировать работу операторов `break` и `continue`:

```
let cars = [
  {
    model: "Golf",
    make: "Volkswagen",
    year: 1999,
    color: "black",
  },
  {
    model: "Picanto",
    make: "Kia",
    year: 2020,
    color: "red",
  },
  {
    model: "Peugeot",
    make: "208",
    year: 2021,
    color: "black",
  },
  {
    model: "Fiat",
    make: "Punto",
    year: 2020,
    color: "black",
  }
];
```

Для начала сосредоточимся на ключевом слове `break`.

break

Мы уже встречали `break` в операторе `switch`. Когда был выполнен `break`, выполнение оператора `switch` завершилось. В циклах ситуация не особо отличается: когда выполняется оператор `break`, цикл завершается, даже если значение условия все еще `true`.

Перед вами нелепый пример, демонстрирующий, как работает `break`:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
  if (i === 4) {
    break;
  }
}
```

Это похоже на цикл, который выводит на экран числа от 0 до 10 (опять же исключая 10). Хотя здесь есть одна загвоздка: как только переменная `i` становится равной `4`, оператор `break` немедленно завершает цикл. После этого код цикла больше не выполняется.

Мы также можем использовать `break`, чтобы прекратить перебирать массив `cars`, когда будет найден автомобиль, соответствующий нашим требованиям.

```
for (let i = 0; i < cars.length; i++) {
  if (cars[i].year >= 2020) {
    if (cars[i].color === "black") {
      console.log("I have found my new car:", cars[i]);
      break;
    }
  }
}
```

Как только обнаружится черная машина 2020 года выпуска или позже, поиск прекращается — и мы просто купим выбранную модель. Последняя машина в массиве тоже была бы неплохим вариантом, но мы даже не рассматривали ее, потому что уже нашли свою. Фрагмент кода выведет следующие данные:

```
I have found my new car: { model: 'Peugeot', make: '208', year: 2021,
color: 'black' }
```

Тем не менее это не лучший способ использовать `break`. Гораздо полезнее, если вместо этого вы можете контролировать условия выхода из цикла. Это предотвращает застревание в бесконечном цикле, и код легче читается.

Если условие цикла на самом деле не является условием как таковым, но представляет собой бесконечный оператор, код становится трудным для восприятия.

Рассмотрим следующий фрагмент:

```
while (true) {
  if (superLongArray[0] != 42 && superLongArray.length > 0) {
    superLongArray.shift();
  } else {
    console.log("Found 42!");
    break;
  }
}
```

Было бы лучше оформить его без оператора `break` и без этого страшного `while(true)`. Давайте лучше сделаем это так:

```
while (superLongArray.length > 0 && notFound) {
  if (superLongArray[0] != 42) {
    superLongArray.shift();
  } else {
    console.log("Found 42!");
    notFound = false;
  }
}
```

Во втором примере мы можем легко увидеть условия цикла, а именно длину массива и значение статуса `notFound`. Однако, используя `while(true)`, мы как бы неправильно интерпретируем концепцию `while`. Вам стоит указать условие, которое должно быть оценено как `true` или `false`, — так ваш код будет приятно читать. Если вы используете `while(true)`, вы на самом деле говорите «навсегда». В этом случае читателю вашего кода придется анализировать его построчно, чтобы понять, что происходит и где цикл заканчивается оператором `break`.

continue

Оператор `break` может использоваться для выхода из цикла, а `continue` — для перехода на новую итерацию цикла. Он завершает текущую итерацию и возвращается в начало, к проверке условия и новой итерации.

Перед вами пример использования оператора `continue`:

```
for (let car of cars){
  if(car.color !== "black"){
    continue;
  }
  if (car.year >= 2020) {
    console.log("we could get this one:", car);
  }
}
```

Здесь применяется следующий подход: мы пропускаем каждую машину, которая не черного цвета, и рассматриваем все остальные, которые не старше 2020 года выпуска. Фрагмент кода выведет следующие данные:

```
we could get this one: { model: 'Peugeot', make: '208', year: 2021,
color: 'black' }
we could get this one: { model: 'Fiat', make: 'Punto', year: 2020,
color: 'black' }
```

Будьте внимательны при использовании `continue` в цикле `while`. Сможете определить, что делает следующий фрагмент кода, не запуская его?

```
// давайте выведем на экран только нечетные числа
let i = 1;
while (i < 50) {
  if (i % 2 === 0){
    continue;
  }
  console.log(i);
  i++;
}
```

В результате вы увидите только значение 1, после чего программа войдет в бесконечный цикл. Это произойдет потому, что continue выполняется до того момента, когда меняется значение i. Таким образом, цикл попадет на оператор continue снова, и снова и т. д. Это можно исправить, передвинув i++ выше и отняв 1 от i, например, так:

```
let i = 1;
while (i < 50) {
  i++;
  if ((i-1) % 2 === 0){
    continue;
  }
  console.log(i-1);
}
```

Но опять же есть лучший способ — без continue в принципе. Шанс получить ошибку на выходе намного меньше:

```
for (let i = 1; i < 50; i = i + 2) {
  console.log(i);
}
```

Как видите, этот вариант намного короче и читабельнее. Значения break и continue обычно вводят, когда необходимо осуществить перебор больших объемов данных, вероятно получаемых приложением со стороны. В такой ситуации у вас будет меньше возможностей применить другие виды контроля. Использование break и continue не лучший способ решения простых задач, но отличный способ знакомства с концепциями.

Практическое занятие 5.8

Это упражнение продемонстрирует создание строки со всеми цифрами по мере их перебора. Мы также можем установить значение для пропуска, добавив условие, в котором будет использоваться continue, пропуская условие соответствия. Второй вариант — выполнить то же упражнение и использовать ключевое слово break.

1. Создайте строковую переменную для ее использования в качестве вывода.
2. Выберите число, которое нужно пропустить, и установите его в качестве переменной.
3. Создайте цикл for, который будет считать до 10.
4. Добавьте условие, чтобы проверить, равно ли значение переменной из текущей итерации числу, которое следует пропустить.

5. Если, согласно заданному условию, число нужно пропустить, переходите к следующей цифре с помощью `continue`.
6. По мере перебора значений добавляйте новое значение счетчика в конец основной выходной переменной.
7. Выведите основную переменную после окончания цикла.
8. Запустите код снова, но замените `continue` на `break` и обратите внимание на разницу в выполнении. При достижении пропускаемого значения цикл не должен останавливаться.

break, continue и вложенные циклы

`break` и `continue` можно также использовать во вложенных циклах. Важно знать следующее: когда во вложенном цикле появляются `break` или `continue`, внешний цикл не останавливается.

Чтобы разобрать работу операторов `break` и `continue` во вложенных циклах, мы реализуем это решение в массиве массивов:

```
let groups = [
  ["Martin", "Daniel", "Keith"],
  ["Margot", "Marina", "Ali"],
  ["Helen", "Jonah", "Sambikos"],
];
```

Давайте проанализируем этот пример. Мы ищем группы, в которых содержатся имена, начинающиеся на M. Если такая группа обнаружена, мы фиксируем ее.

```
for (let i = 0; i < groups.length; i++) {
  let matches = 0;

for (let j = 0; j < groups[i].length; j++) {
  if(groups[i][j].startsWith("M")){
      matches++;
    } else {
      continue;
    }
  if (matches === 2){
      console.log("Found a group with two names starting with an M:");
      console.log(groups[i]);
      break;
    }
  }
}
```

Сначала мы перебираем массивы верхнего уровня и устанавливаем счетчик `matches` с начальным значением `0`. Мы будем перебирать значения для каждого из этих массивов. Когда значение начинается на M, мы поднимаем значение `matches` на один и проверяем, есть ли новое совпадение. Если две M уже найдены, мы выходим из внутреннего цикла и продолжаем работу во внешнем цикле, который перейдет к следующему массиву верхнего уровня, раз после внутреннего цикла ничего не происходит.

Если имя не начинается с M, нам не требуется проверить, равно ли значение `matches` двум. После этого можно переходить к сравнению значения следующего элемента внутреннего массива.

Взгляните на этот пример. Как думаете, что он выведет в качестве результата?

```
for (let group of groups){
  for (let member of group){
    if (member.startsWith("M")){
      console.log("found one starting with M:", member);
      break;
    }
  }
}
```

Этот цикл перебирает массивы и для каждого массива проверяет значение, чтобы увидеть, начинается ли оно с буквы M. Если совпадение найдено, внутренний цикл разрывается. Таким образом, если один из массивов в массиве содержит несколько значений, начинающихся с M, будет найдено только первое, итерация по этому массиву прервется и мы перейдем к следующему массиву.

Результат вывода на экран:

```
found one starting with M: Martin
found one starting with M: Margot
```

Мы видим, что он находит Марго — первую во втором массиве, но пропускает Марину, потому что она вторая. Цикл останавливается после того, как найдена одна группа, он не будет перебирать другие элементы во внутреннем массиве. Вместо этого он продолжит работу со следующим массивом, в котором нет имен, начинающихся с буквы «M».

Если бы мы хотели найти группы, в которых есть участник с именем, начинающимся на букву «M», предыдущий фрагмент кода подошел бы для этой задачи, потому что мы разрываем внутренний цикл, как только находим совпадение. Это может быть полезно всякий раз, когда вы хотите убедиться, что массив в наборе

данных содержит хотя бы одно какое-то значение. Цикл `for of` не будет указывать индекс или место, где он нашел значение, из-за своей сущности. Он просто остановится и вернет значение элемента массива. Если вам нужно больше информации о наборе данных, вы можете работать со счетчиками, обновляемыми в каждой итерации.

Если мы хотим убедиться, что только одно из всех имен в массиве массивов начинается с буквы M, придется выйти из внешнего цикла. Вот некоторые возможные операции с метками циклов.

break, continue и метки

Находясь во внутреннем цикле, мы можем выйти из внешнего цикла, но только в том случае, если присвоим нашему циклу метку. Это можно реализовать следующим образом:

```
outer:
for (let group of groups) {
  inner:
  for (let member of group) {
    if (member.startsWith("M")) {
      console.log("found one starting with M:", member);
      break outer;
    }
  }
}
```

Мы присваиваем блоку метку, помещая слово и двоеточие перед блоком кода. Эти слова могут быть практически любыми (в нашем случае `outer` и `inner`), но не зарезервированными непосредственно JavaScript словами, такими как `for`, `if`, `break`, `else` и т. д.

Цикл выводит только первое имя, начинающееся на M:

```
found one starting with M: Martin
```

Выведен будет только один результат, потому что далее мы выйдем из внешнего цикла и все циклы завершатся после нахождения первого совпадения. Аналогичным образом можно также продолжить внешний цикл.

Если, найдя одно совпадение, вам потребуется закончить выполнение кода, используйте этот вариант. Он будет идеален, например, в случаях, когда вы захотите проверить наличие ошибок и выйти, если их нет.

Проект текущей главы

Математическая таблица умножения

В этом проекте с помощью циклов вы создадите математическую таблицу умножения. Вы можете сделать это сами или воспользоваться следующими предложенными шагами.

1. Создайте пустой массив для хранения итоговой таблицы умножения.
2. Задайте переменную `value`, чтобы указать, сколько значений вы хотите умножить друг на друга и отобразить результаты этого.
3. Создайте внешний цикл `for` для перебора значений каждой строки и массив `temp` для хранения значений строк. Каждая строка будет представлять собой массив ячеек, вложенных в итоговую таблицу.
4. Добавьте внутренний цикл `for` для значений столбцов, который будет заносить умноженные значения строк и столбцов в массив `temp`.
5. Добавьте временные строки данных, содержащие рассчитанные решения, в основной массив таблицы. Окончательный результат добавит строку значений вычислений.

Вопросы для самопроверки

1. Какой результат следует ожидать после выполнения этого кода?

```
let step = 3;

for (let i = 0; i < 1000; i += step) {
    if (i > 10) {
        break;
    }
    console.log(i);
}
```

2. Каким будет итоговое значение `myArray` и каким получится результат в консоли?

```
const myArray = [1,5,7];
for(el in myArray){
    console.log(Number(el));
    el = Number(el) + 5;
    console.log(el);
}
console.log(myArray);
```

Резюме

В этой главе мы представили вам концепцию циклов. Циклы позволяют повторно выполнять определенные фрагменты кода. При создании цикла необходимо задать какое-то условие, и пока это условие верно, цикл будет работать. Как только условие станет ложным, цикл завершится.

Мы рассмотрели цикл `while`, в который просто вводится условие, и пока его логическое значение верно, цикл продолжается. Если условие не принимает значения `true`, цикл не будет выполнен даже один раз.

Этим цикл `while` отличается от цикла с условием `do while`. Во втором случае код всегда выполняется хотя бы один раз, после чего начинается проверка соответствия условия. Если условие верно, код выполняется снова, пока значение условия не изменится на противоположное. Это очень удобно, когда вы работаете с внешними данными, такими как пользовательский ввод. Мы можем сделать запрос один раз, а потом повторять его, пока введенная информация не будет корректной.

Далее мы рассмотрели цикл `for` с немного другим синтаксисом. В данном цикле мы должны определить переменную, проверить условие (желательно с использованием этой переменной, но не обязательно), а затем указать действие, которое должно выполняться после каждой итерации. Опять же лучше, чтобы действие включало переменную из первой части цикла `for` — в таком случае код будет выполняться до тех пор, пока условие истинно.

Вы также узнали два способа перебора массивов и объектов `for in` и `for of`. Цикл `for in` работает с ключами, а цикл `for of` перебирает значения. Они просматривают все имеющиеся элементы. Преимущество этих циклов в том, что JavaScript контролирует их выполнение — и мы не можем случайно застрять в бесконечном цикле или пропустить значения.

Последними мы рассмотрели `break` и `continue`. Ключевое слово `break` можно использовать, чтобы завершить цикл немедленно. Ключевое слово `continue` прервет текущую итерацию и вернется к началу, чтобы запустить следующую итерацию цикла, если условие все еще сохраняет значение `true`.

В следующей главе мы собираемся добавить в нашу копилку действительно мощный инструмент JavaScript: функции! Они позволяют поднять ваши навыки программирования на новый уровень и улучшить структуру кода.

6 Функции

Вы уже видели многое в JavaScript и теперь готовы к знакомству с функциями. Вы поймете, что использовали функции и ранее, но теперь пришло время научиться писать свои собственные. Функции — отличный строительный блок, который сократит объем кода, необходимого для вашего приложения. Вы можете вызывать функции везде, где требуется. И можете создавать их как своеобразные шаблоны с переменными. В зависимости от того, как вы прописали функцию, вы можете применить ее во множестве ситуаций.

Функции требуют, чтобы вы по-другому думали о структуре вашего кода, это может быть сложно, особенно поначалу. Но, как только вы освоите данный способ мышления, функции действительно помогут вам писать хорошо структурированный, универсальный и малообслуживаемый код. Погрузимся в новый уровень абстракции!

Мы рассмотрим следующие темы:

- основные функции;
- аргументы функций;
- оператор `return`;
- область видимости переменных в функциях;
- рекурсивные функции;
- вложенные функции;
- анонимные функции;
- функции обратного вызова.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Основные функции

В рамках этой книги у вас уже была возможность поработать с функциями. Помните `prompt()`, `console.log()`, `push()` и `sort()` в темах с массивами? Все это функции. Функции — это набор операторов, объявлений переменных, циклов и тому подобного, сложенных вместе. Вызов функции означает, что будет выполнена целая группа операторов.

Давайте сначала рассмотрим, как мы можем вызывать функции, а затем научимся писать функции самостоятельно.

Самовызывающиеся функции

Функции легко распознать по скобкам в конце. Мы можем вызывать функции так:

```
nameOfTheFunction();
functionThatTakesInput("the input", 5, true);
```

Выше представлен вызов функции, называемой `nameOfTheFunction`, без аргументов и вызов функции `functionThatTakesInput` с тремя запрошенными аргументами. Давайте попробуем написать несколько функций и посмотрим, как они могут выглядеть.

Создание функций

Обозначить функцию можно с помощью ключевого слова `function`. Перед вами шаблон синтаксиса:

```
function nameOfTheFunction() {
     // содержимое функции
}
```

Представленную функцию можно вызвать следующим образом:

```
nameOfTheFunction();
```

Напишем функцию, которая спросит ваше имя, а после поприветствует:

```
function sayHello() {
  let you = prompt("What's your name? ");
  console.log("Hello", you + "!");
}
```

Мы добавляем пробел после вопросительного знака, чтобы пользователь начал вводить свой ответ после некоторого отступа, а не сразу после текста вопроса. Вызовем эту функцию:

```
sayHello();
```

Получим сообщение:

```
What's your name? >
```

Введем имя. Результат будет следующим:

```
Hello Maaike!
```

Уделите минутку внимания взаимосвязи между функциями и переменными. Как вы уже знаете, функции могут содержать переменные, которые определяют их выполнение. Верно и обратное: переменные могут содержать функции. Вы еще здесь? Ниже представлен пример переменной, содержащей функцию (`varContainingFunction`), и переменной внутри функции (`varInFunction`):

```
let varContainingFunction = function() {
    let varInFunction = "I'm in a function.";
    console.log("hi there!", varInFunction);
};

varContainingFunction();
```

Переменные содержат определенное значение и *являются* чем-то. Но они ничего *не делают*. Функции — это действия. Они представляют собой совокупность операторов, которые могут быть исполнены, когда функцию вызовут. JavaScript не будет запускать операторы, если функции не вызываются. Мы вернемся к идее хранения функций в переменных и рассмотрим некоторые преимущества этого в разделе «Анонимные функции» в конце главы, но сейчас перейдем к наилучшему способу присвоения имен вашим функциям.

Название функции

Присвоение вашей функции имени может показаться тривиальной задачей, но здесь следует иметь в виду некоторые моменты. Если вкратце:

- используйте «горбатый регистр» — способ записи, при котором первое слово начинается со строчной буквы, а следующие — с заглавной. Это очень облегчает чтение кода и делает его более последовательным;
- убедитесь, что имя описывает действие функции: функцию добавления номеров лучше назвать `addNumbers`, чем `myFunc`;
- используйте глаголы для описания действия функции: оживите ее. Так, вместо `hiThere` возьмите `sayHi`.

Практическое занятие 6.1

Посмотрим, сможете ли вы написать функцию самостоятельно. Пусть это будет функция сложения двух чисел.

1. Создайте функцию, которая берет два параметра, складывает их вместе и возвращает результат.
2. Задайте две переменные с различными значениями.
3. Примените функцию к этим переменным и выведите результат с помощью `console.log`.
4. Пропишите повторный вызов функции с использованием еще двух цифр в качестве аргументов.

Практическое занятие 6.2

Мы собираемся создать программу, которая будет случайным образом описывать введенное имя.

1. Сформируйте массив описаний.
2. Создайте функцию, запрашивающую имя у пользователя.
3. Выберите случайное значение из массива, используя `Math.random`.
4. Выведите на экран значение из запроса и значение, выбранное в массиве.
5. Вызовите функцию.

Параметры и аргументы

Вы наверняка уже заметили выше два термина: «параметры» и «аргументы». Оба, как правило, используются для обозначения информации, которая передается в функцию:

```
function tester(para1, para2){
    return para1 + " " + para2;
}
const arg1 = "argument 1";
const arg2 = "argument 2";
tester(arg1, arg2);
```

Параметр определяется как переменная, которая указывается в скобках функции, задающих область действия функции. Параметры объявляются следующим образом:

```
function myFunc(param1, param2) {
  // код функции;
}
```

Практический пример может быть следующим. Функция берет `x` и `y` в качестве параметров:

```
function addTwoNumbers(x, y) {
  console.log(x + y);
}
```

При вызове функция просто сложит параметры и выдаст результат. Однако для этого мы можем вызвать функцию с аргументами:

```
myFunc("arg1", "arg2");
```

Мы видели различные аргументы, например:

```
console.log("this is an argument");
prompt("argument here too");

let arr = [];
arr.push("argument");
```

Меняя аргументы, которые вызываются с помощью функции, вы также можете изменять результат ее работы — это делает функцию очень мощным и гибким инструментом. Вот как на практике применяется функция `addTwoNumbers()`:

```
addTwoNumbers(3, 4);
addTwoNumbers(12,-90);
```

Результат вывода на экран:

```
7
-78
```

Как вы могли заметить, результаты вызова функции в обоих случаях отличаются. Это происходит потому, что мы вызываем ее с разными аргументами, заменяющими `x` и `y`, которые отправляются в функцию для использования в ее области действия.

Практическое занятие 6.3

Создайте простой калькулятор, который принимает два числа и одно строковое значение, называющее операцию. Если операция — сложение, то следует сложить два числа. Если вычитание — вычесть одно число из другого. Если конкретного указания нет, базовое значение должно быть равным `add`.

Выведите результат работы функции. Протестируйте ее, вызвав с помощью разных операторов и без оператора.

1. Определите две переменные, содержащие численные значения.
2. Задайте переменную для хранения оператора: `+` или `-`.
3. Создайте функцию, которая в своих параметрах извлекает два значения и строковый оператор. Используйте эти значения с условием, чтобы проверить, каким будет оператор: `+` или `-`. Далее в зависимости от выбора сложите или вычтите значения (помните, что, если указан недопустимый оператор, функция по умолчанию должна выполнить операцию сложения).
4. Вызовите функцию, используя ваши переменные, и выведите ответ на экран с помощью `console.log()`.
5. Измените значение оператора на другое, `+` или `-`, и снова вызовите функцию с обновленными аргументами.

Неопределенные параметры или параметры по умолчанию

Что будет, если мы вызовем функцию `addTwoNumbers()` без аргументов? Остановитесь на минутку и подумайте, что, по вашему мнению, она сделает:

```
addTwoNumbers();
```

Некоторые языки могут выйти из строя и завопить — но не JavaScript. JavaScript просто присваивает переменным тип по умолчанию, которым является `undefined`. А `undefined` + `undefined` равно:

```
NaN
```

В JavaScript можно задать и другие параметры по умолчанию. Это реализуется следующим образом:

```
function addTwoNumbers(x = 2, y = 3) {
  console.log(x + y);
}
```

Если вы сейчас вызовете функцию без аргументов, она автоматически назначит 2 для x и 3 для y, если только вы не определите аргументы самостоятельно. Значения, которые вы сами задаете при вызове функции, имеют приоритет над значениями аргументов по умолчанию. Итак, какими будут результаты трех вызовов приведенной выше функции?

```
addTwoNumbers();
addTwoNumbers(6, 6);
addTwoNumbers(10);
```

А результаты будут следующими:

```
5
12
13
```

Первая функция использует аргументы по умолчанию, поэтому x — это 2, а y — это 3. Вторая присваивает 6 обеим переменным, x и y. Логика действия последней немного менее очевидна. Мы приводим только один аргумент — так какому же из них будет присвоено это значение? Что ж, JavaScript не любит усложнять, поэтому даст это значение первому аргументу — x. Следовательно, x станет равным 10, а y примет значение по умолчанию 3 — в сумме получится 13.

Если вы вызовете функцию с бóльшим количеством аргументов, чем надо, ничего не произойдет. JavaScript исполнит функцию, взяв первые по порядку аргументы, которые можно сопоставить с параметрами. Например, так:

```
addTwoNumbers(1,2,3,4);
```

Результат вывода на экран:

```
3
```

Функция сложит только 1 и 2, игнорируя оставшиеся аргументы (3 и 4).

Специальные функции и операторы

Существует несколько особых способов написания функций, а также есть специальные операторы, которые однозначно пригодятся в вашей работе. Речь о стрелочных функциях arrow, а также об операторах spread и rest. Стрелочные функции отлично подходят для передачи функций в качестве параметров и использования более коротких обозначений. Операторы spread и rest также облегчают нашу жизнь — за счет того, что они более гибкие при отправке аргументов и работе с массивами.

Стрелочные функции

Способ написания стрелочных функций поначалу кажется запутанным. Их применение выглядит следующим образом:

```
(параметр1, параметр2) => тело функции;
```

Или в случае, когда параметров не задано:

```
() => тело функции;
```

Или для одного параметра (скобки здесь не нужны):

```
параметр => тело функции;
```

Или для многострочной функции с двумя параметрами:

```
(параметр1, параметр2) => {
  // строка 1;
  // сколько угодно строк;
};
```

Стрелочные функции будут нужны всякий раз, когда вы захотите написать реализацию сразу же, например, в качестве аргумента внутри другой функции. По сути, они являются более краткой записью обычных функций. Чаще всего используются в функциях, состоящих только из одного оператора. Начнем с простой функции, которую перепишем в стрелочную:

```
function doingStuff(x) {
  console.log(x);
}
```

Чтобы совершить задуманное, вам придется сохранить стрелочную функцию в переменной или отправить ее в качестве аргумента, если потом она вам еще понадобится. Для выполнения стрелочной функции мы используем имя переменной. В этом случае у нас останется только один параметр, поэтому заключать его в круглые скобки не обязательно. Записать можно так:

```
let doingArrowStuff = x => console.log(x);
```

И вызвать следующим образом:

```
doingArrowStuff("Great!");
```

В консоли выводится `Great!`. Круглые скобки вам понадобятся, если в вашей функции более одного аргумента, например:

```
let addTwoNumbers = (x, y) => console.log(x + y);
```

Вызвать можно так:

```
addTwoNumbers(5, 3);
```

Функция выведет результат `8`. Если аргументов нет, скобки также необходимы:

```
let sayHi = () => console.log("hi");
```

При вызове `sayHi()` функция выведет на экран `hi`.

В качестве итогового примера скомбинируем встроенные методы и стрелочные функции. Допустим, мы применим к массиву метод `foreach()`, выполняющий с каждым элементом этого массива определенное действие. Взгляните на следующую запись:

```
const arr = ["squirrel", "alpaca", "buddy"];
arr.forEach(e => console.log(e));
```

Результат будет таким:

```
squirrel
alpaca
buddy
```

Каждый элемент массива воспринимается как входные данные, и для него выполняется стрелочная функция. В нашем случае функция осуществляет логирование элемента. Таким образом, на выходе мы получаем каждый отдельный элемент массива.

Стрелочные функции в сочетании со встроенными — очень мощный инструмент. Мы можем сделать что-то с каждым элементом массива без необходимости вести счет и создавать сложные циклы. Позже мы рассмотрим еще больше отличных способов их применения.

Оператор spread

`spread` является особенным оператором. Это три точки, используемые перед ссылочным выражением или строкой. Данный оператор распределяет аргументы или элементы массива.

Возможно, описание звучит не очень понятно, поэтому рассмотрим простой пример:

```
let spread = ["so", "much", "fun"];
let message = ["JavaScript", "is", ...spread, "and", "very",
               "powerful"];
```

В консоли отобразится следующее значение массива:

```
['JavaScript', 'is', 'so', 'much', 'fun', 'and', 'very', 'powerful']
```

Как видите, элементы оператора `spread` становятся отдельными элементами в массиве. Оператор `spread` разделяет прежний массив на отдельные элементы в новом массиве. Его также можно использовать для отправки в функцию нескольких аргументов:

```
function addTwoNumbers(x, y) {
  console.log(x + y);
}
let arr = [5, 9];
addTwoNumbers(...arr);
```

В консоли фиксируется значение `14`, данный код делает то же самое, что и следующая функция:

```
addTwoNumbers(5, 9);
```

Оператор `spread` позволяет избежать необходимости копировать длинный массив или строку в функцию, что экономит время и снижает сложность кода. Вы можете вызвать функцию с несколькими операторами `spread`, в таком случае она примет все элементы массива за входные данные. Пример:

```
function addFourNumbers(x, y, z, a) {
  console.log(x + y + z + a);
}
let arr = [5, 9];
let arr2 = [6, 7];
addFourNumbers(...arr, ...arr2);
```

На экран будет выведено значение `27`, как при вызове следующей функции:

```
addFourNumbers(5, 9, 6, 7);
```

Параметр rest

Параметр `rest` подобен оператору `spread`. У него тот же синтаксис, но `rest` используется внутри списка параметров функции. Вспомните, что происходит, если мы отправляем аргумент слишком много раз:

```
function someFunction(param1, param2) {
  console.log(param1, param2);
}
someFunction("hi", "there!", "How are you?");
```

Вот именно: ничего особенного. JavaScript просто притворится, что мы отправили только два аргумента, и зафиксирует: `hi there!`. Если мы используем параметр `rest`, он позволит отправить любое количество аргументов и преобразовать их в массив параметров. Вот пример:

```
function someFunction(param1, ...param2) {
  console.log(param1, param2);
}
someFunction("hi", "there!", "How are you?");
```

Результат будет следующим:

```
hi [ 'there!', 'How are you?' ]
```

Как видите, второй параметр был заменен на массив, содержащий второй и третий аргументы. Это очень полезно, когда вы не знаете, какое количество аргументов будет получено. Использование параметра `rest` позволяет обработать это переменное число аргументов, например, с помощью цикла.

Возврат значений функций

До сих пор мы упускали из вида крайне важную опцию, благодаря которой функции так полезны: возвращаемое значение. Функции могут возвращать результат, когда мы указываем возвращаемое значение. Это значение можно сохранить в переменной. Мы уже делали это раньше — помните `prompt()`?

```
let favoriteSubject = prompt("What is your favorite subject?");
```

Мы записываем результат функции `prompt()` в переменную `favoriteSubject`. В данном случае это будет то значение, которое введет пользователь. Посмотрим, что будет происходить, когда мы сохраним результат функции `addTwoNumbers()` и выведем переменную.

```
let result = addTwoNumbers(4, 5);
console.log(result);
```

Может быть, вы догадались (а может, и нет) — при выполнении функции в консоли появится следующее:

```
9
undefined
```

Значение 9 выведено на экран, потому что addTwoNumbers() содержит в себе оператор console.log(). Строка console.log(result) выводит значение undefined, потому что она не получила значения результата для сохранения данных. Получается, наша функция addTwoNumbers() не отправляет никаких данных после выполнения. Поскольку JavaScript не любит сбоить и создавать проблемы, переменной будет присвоено значение undefined. Чтобы изменить это, мы можем переписать функцию addTwoNumbers() и вернуть значение вместо его вывода на экран. Так намного эффективнее, ведь мы можем сохранить результат этой функции и продолжить работу с ним в остальной части кода:

```
function addTwoNumbers(x, y) {
  return x + y;
}
```

Оператор return прекращает выполнение функции и отправляет обратно любое значение, стоящее после return. Если это выражение подобно приведенному выше, оно вычислит результат операции, а затем вернет его в то место кода, где оно было вызвано (в нашем случае переменная result):

```
let result = addTwoNumbers(4, 5);
console.log(result);
```

С этими дополнениями фрагмент кода выведет 9 на экран.

Как вы думаете, что делает следующий код?

```
let resultsArr = [];

for(let i = 0; i < 10; i ++){
  let result = addTwoNumbers(i, 2*i);
  resultsArr.push(result);
}

console.log(resultsArr);
```

Он выводит в консоль массив всех результатов. Функция была вызвана в цикле. В первой итерации i равно 0. Следовательно, результат равен 0. В последней итерации значение i становится 9, следовательно, последнее значение массива — 27. Ниже представлен результат:

```
[
   0,  3,  6,  9, 12,
  15, 18, 21, 24, 27
]
```

Практическое занятие 6.4

Измените калькулятор, созданный в практическом занятии 6.2, так, чтобы результаты не выводились на экран, а возвращались. Далее вызовите функцию в цикле десять или более раз и сохраните итог в массиве. Как только цикл будет завершен, выведите массив результатов на экран.

1. Создайте пустой массив для хранения значений, которые будут рассчитаны в цикле.
2. Создайте цикл, который выполняется десять раз, каждый раз увеличивая счетчик на 1 и генерируя два значения на каждой итерации. Чтобы получить первое значение, умножьте значение счетчика на 5. Чтобы получить второе значение, умножьте значение счетчика само на себя.
3. Создайте функцию, которая возвращает значение двух параметров, переданных в функцию при ее вызове. Сложите значения и верните результат.
4. Внутри цикла вызовите функцию вычисления, передав в нее два значения в качестве аргументов и сохранив возвращенный результат в переменной ответа.
5. Не выходя из цикла, по мере его прохождения помещайте значения результатов в массив.
6. После завершения цикла выведите массив на экран.
7. Вы должны увидеть значения `[0, 6, 14, 24, 36, 50, 66, 84, 104, 126]`.

Возврат результата с помощью стрелочных функций

Результат однострочной стрелочной функции можно отправить без ключевого слова `return`. Поэтому, если мы хотим переписать функцию, мы можем сделать из нее стрелочную функцию:

```
let addTwoNumbers = (x, y) => x + y;
```

Можно вызвать ее и сохранить результат следующим образом:

```
let result = addTwoNumbers(12, 15);
console.log(result);
```

На экран выводится `27`. Если функция многострочная, потребуется использовать ключевое слово `return`, как мы это делали в предыдущем подразделе:

```
let addTwoNumbers = (x, y) => {
  console.log("Adding...");
  return x + y;
}
```

Область видимости переменных в функциях

В этом разделе мы обсудим тему, которая зачастую считается сложной, — область видимости. Данное понятие определяет, в каком месте кода можно получить доступ к переменной. Когда переменная *в зоне видимости*, к ней можно получить доступ. Когда переменная *вне зоны видимости*, вы не сможете получить к ней доступ. Рассмотрим эту тему для локальных и глобальных переменных.

Локальные переменные в функциях

Локальные переменные находятся только в области действия функции, в которой они были объявлены. Данное утверждение применимо к переменным `let` и `var`. Между ними существует разница, далее мы также коснемся этого вопроса. Параметры функции (не используют `let` или `var`) — это тоже локальные переменные. Возможно, все это выглядит расплывчато, но в следующем фрагменте кода мы продемонстрируем, что имеем в виду:

```
function testAvailability(x) {
  console.log("Available here:", x);
}

testAvailability("Hi!");
console.log("Not available here:", x);
```

Результат вывода на экран:

```
Available here: Hi!
ReferenceError: x is not defined
```

Вызванная внутри функции переменная `x` логируется. Вне функции оператор терпит неудачу, потому что `x` — локальная переменная функции `testAvailability()`. Этот пример показывает, что параметры функции недоступны за ее пределами.

Находясь вне функции, переменные уходят из области видимости, а будучи внутри функции, они находятся также в области видимости. Посмотрим на переменную, объявленную внутри функции:

```
function testAvailability() {
  let y = "Local variable!";
  console.log("Available here:", y);
}

testAvailability();
console.log("Not available here:", y);
```

На экране мы увидим следующее:

```
Available here: Local variable!
ReferenceError: y is not defined
```

Переменные, объявленные внутри функции, также недоступны вне функции.

Начинающих может сбить с толку объединение локальных переменных и `return`. Говорим сразу: локальные переменные, объявленные внутри функции, недоступны вне функции, но с помощью `return` вы можете сделать доступными их значения (вернуть эти значения). Ключевое слово здесь — *значения*! Вы не можете вернуть сами переменные. Но вместо этого можете захватить нужное значение и присвоить его другой переменной, например, так:

```
function testAvailability() {
  let y = "I'll return";
  console.log("Available here:", y);
  return y;
}

let z = testAvailability();
console.log("Outside the function:", z);
console.log("Not available here:", y);
```

Таким образом, возвращаемое значение `I'll return`, присвоенное локальной переменной `y`, было возвращено и сохранено в переменной `z`.

Переменную z можно тоже назвать y, но тогда возникнет путаница, ведь это все равно будет другая переменная.

Результат выполнения данного фрагмента будет следующим:

```
Available here: I'll return
Outside the function: I'll return
ReferenceError: y is not defined
```

Сравнение переменных let и var

Разница между переменными `let` и `var` заключается в следующем: `var` видима в пределах функции, как мы и говорили выше, а `let` фактически видима в пределах блока кода. Область блока кода определяется фигурными скобками `{ }`. Поэтому на всем протяжении кода между этими скобками переменная `let` будет доступна для использования.

Посмотрим на это различие в действии:

```
function doingStuff() {
  if (true) {
    var x = "local";
  }
  console.log(x);
}

doingStuff();
```

Результат работы фрагмента будет следующим:

```
local
```

Когда мы используем `var`, переменная становится видимой в пределах функции и доступной в любом месте блока функции (даже если до этого ее объявили как `undefined`). Таким образом, после выполнения блока `if` доступ к переменной `x` все еще сохраняется.

А вот что происходит с `let`:

```
function doingStuff() {
  if (true) {
    let x = "local";
  }
  console.log(x);
}

doingStuff();
```

В результате вы получите:

```
ReferenceError: x is not defined
```

В этом случае отобразилась ошибка `x is not defined`. Поскольку видимость `let` ограничена блочной областью, `x` выходит из области видимости, когда блок `if` заканчивается. Доступа к переменной больше нет.

Окончательное различие между `let` и `var` заключается в порядке объявления в скрипте. Попробуйте использовать значение `x`, прежде чем объявить переменную с помощью `let`:

```
function doingStuff() {
  if (true) {
    console.log(x);
    let x = "local";
  }
}

doingStuff();
```

Мы получим ошибку `ReferenceError` — `x` не инициализирована. А все потому, что переменные, которые были объявлены с помощью `let`, но не были определены, не будут доступны даже в пределах одного и того же блока. Как думаете, что произойдет при объявлении таким же образом переменной `var`?

```
function doingStuff() {
  if (true) {
    console.log(x);
```

```
    var x = "local";
  }
}

doingStuff();
```

На сей раз ошибки не будет. Когда мы используем переменную `var` перед оператором `define`, мы получаем ее со значением `undefined`. Это связано с явлением, которое называется *hoisting* — «поднятие». Другими словами, переменной `var`, если она используется до того момента, как она была объявлена, присваивается значение `undefined` вместо вызова ошибки `ReferenceError`.

Поднятие, а также то, как при необходимости свести на нет последствия его применения, являются более сложными темами, которые мы рассмотрим в главе 12.

Область видимости константы

Видимость константы ограничена пределами блока, как у переменной `let`. Поэтому правила для определения ее области видимости аналогичны правилам работы с `let`. Вот вам пример:

```
function doingStuff() {
  if (true) {
    const X = "local";
  }
  console.log(X);
}

doingStuff();
```

В результате вы получите:

```
ReferenceError: X is not defined
```

Использование переменной `const` до определения ее значения так же, как и в случае переменной `let`, вызовет ошибку `ReferenceError`.

Глобальные переменные

Как вы уже могли догадаться, глобальные переменные — это переменные, объявленные вне функции или какого-либо другого блока кода. Они доступны в той области (функции или блоке), в которой были определены, а также в любых «нижних» областях. Если глобальная переменная определена вне функции, она будет доступна не просто внутри функции, а в любых функциях или других блоках кода этой функции. Таким образом, глобальная переменная, определенная на верхнем уровне вашей программы, будет доступна во всей вашей программе.

Рассмотрим следующий пример:

```
let globalVar = "Accessible everywhere!";
console.log("Outside function:", globalVar);

function creatingNewScope(x) {
  console.log("Access to global vars inside function." , globalVar);
}

creatingNewScope("some parameter");

console.log("Still available:", globalVar);
```

Результат вывода на экран:

```
Outside function: Accessible everywhere!
Access to global vars inside function. Accessible everywhere!
Still available: Accessible everywhere!
```

Как видите, глобальные переменные доступны отовсюду, потому что не были объявлены в блоке. После того как они определены, они *всегда* находятся в области видимости — неважно, где вы будете их использовать. Однако вы можете скрывать их доступность внутри функции, задавая новую переменную с тем же именем в ограниченной области. Подобное можно провернуть для `let`, `var` и `const`. (Значения переменной `const` это не изменит: вы создаете новую `const`, которая заменит значение первой переменной во внутреннем цикле.) В одной области видимости не получится задать две переменные `let` или две переменные `const` с одним и тем же именем. Так можно с переменной `var`, но лучше избегайте подобных действий, чтобы не создавать путаницу.

Если вы создадите переменную с таким же именем внутри функции, значение этой переменной будет использоваться всякий раз, когда вы будете ссылаться на имя переменной в рамках этой конкретной функции. Например:

```
let x = "global";

function doingStuff() {
  let x = "local";
  console.log(x);
}

doingStuff();
console.log(x);
```

Результат вывода на экран:

```
local
global
```

Как видите, значение x внутри функции doingStuff() имеет статус local. Тем не менее вне функции значение остается global. Поэтому с особой осторожностью относитесь к смешиванию имен в локальной и глобальной областях — а лучше избегайте этого.

То же самое касается имен параметров. Если название параметра совпадает с именем глобальной переменной, в коде будет применяться значение параметра:

```
let x = "global";

function doingStuff(x) {
  console.log(x);
}

doingStuff("param");
```

Как результат, в консоли выводится param.

Тем не менее слишком сильно полагаться на глобальные переменные может быть опасным. Вы почувствуете это по мере увеличения размера ваших приложений. Как мы только что видели, локальные переменные переписывают значение глобальных переменных. Поэтому лучше всего работать с локальными переменными в функциях, так у вас будет больше контроля над тем, что вы создаете. На данном этапе обучения подобные утверждения могут выглядеть немного расплывчато, но в реальной жизни, когда размер кода станет больше и будет задействовано больше строк и файлов кода, вы все поймете.

Есть еще один очень важный нюанс, касающийся областей видимости. Начнем с примера. Сможете ли вы разобраться, что должно получиться в итоге?

```
function confuseReader() {
  x = "Guess my scope...";
  console.log("Inside the function:", x);
}

confuseReader();
console.log("Outside of function:", x);
```

Ваш ответ готов? Результат будет выглядеть следующим образом:

```
Inside the function: Guess my scope...
Outside the function: Guess my scope...
```

Не закрывайте книгу, сейчас мы все объясним. Давайте посмотрим внимательно: переменная x в функции определена без использования ключевых слов let или var. Ранее она объявлена не была — перед нами весь код программы. JavaScript не видит let или var и решает: «Это, должно быть, глобальная переменная». Несмотря на

то что x определяется внутри функции, объявленная внутри функции переменная получает глобальную область видимости и все равно может быть доступна за пределами функции.

Стоит подчеркнуть, что это ужасная практика. Если вам нужна глобальная переменная, лучше объявите ее в начале файла.

Немедленно вызываемое функциональное выражение

Immediately Invoked Function Expression (немедленно вызываемое функциональное выражение), или *IIFE*, — это функция, которая выполняется сразу же после того, как была определена. Она анонимная, у нее нет имени, и она самоисполняющаяся.

Данная функция может быть полезна, когда вы хотите что-то инициализировать с ее помощью. Она также используется во многих проектных шаблонах, например, чтобы задать различные частные и общедоступные переменные и функции.

При работе с немедленно вызываемым функциональным выражением важно учитывать аспект доступа к функциям и переменным. Если у вас есть IIFE в области видимости верхнего уровня, то все, что в ней находится, недоступно извне, даже когда запрос поступает из того же верхнего уровня.

Вот как определяется IIFE:

```
(function () {
  console.log("IIFE!");
})();
```

Сама функция заключена в круглые скобки, что позволяет ей создавать экземпляр функции. Отсутствие этих скобок привело бы к ошибке, потому что у данной функции нет имени (однако это можно обойти, назначив функцию переменной, где выходные данные могут быть возвращены переменной).

Оператор `();` выполняет неназванную функцию — это должно быть сделано сразу после ее объявления. Если бы вашей функции требовался параметр, вы бы передали его как раз в заключительных скобках.

Вы также можете комбинировать IIFE с другими шаблонами функций. Например, можете использовать здесь стрелочную функцию, чтобы сделать запись еще более краткой:

```
(()=>{
    console.log("run right away");
})();
```

И снова мы применили здесь `();`, чтобы вызвать созданную функцию.

Практическое занятие 6.5

Используйте IIFE, чтобы создать несколько немедленно вызываемых функций и посмотреть, как они влияют на область видения.

1. Создайте переменную `let` и присвойте ей значение `1000`.
2. Создайте функцию IIFE и в пределах ее области действия назначьте новое значение переменной с тем же именем. С помощью функции выведите локальное значение на экран.
3. Создайте немедленно вызываемое функциональное выражение, присвоив его новой переменной `result`, и задайте другое значение переменной с тем же именем в этой области видимости. Верните локальное значение переменной `result` и вызовите функцию. Выведите значение переменной `result` вместе с именем переменной, которое вы использовали. Какое значение она содержит сейчас?
4. Наконец, создайте анонимную функцию с параметром. Добавьте логику, которая будет присваивать переданное значение тому же имени переменной (как в предыдущих шагах) и выводить его как часть строкового выражения. Вызовите функцию и введите желаемое значение в круглых скобках.

Рекурсивные функции

В некоторых случаях вам понадобится вызвать функцию изнутри самой этой функции. Подобный ход может быть прекрасным решением довольно сложных проблем. Однако есть кое-что, что следует иметь в виду. Как думаете, что делает следующий код?

```
function getRecursive(nr) {
  console.log(nr);
  getRecursive(--nr);
}

getRecursive(3);
```

Код выводит значение `3`, после чего выполняется декремент — и функция продолжает работу. Почему она не останавливается? Ну, мы ей не сказали, когда она должна прекратиться. Посмотрите на улучшенную версию:

```
function getRecursive(nr) {
  console.log(nr);
  if (nr > 0) {
    getRecursive(--nr);
  }
}

getRecursive(3);
```

Функция будет вызывать сама себя, пока значение параметра больше `0`. После этого она остановится.

Когда мы вызываем функцию рекурсивно, при каждом исполнении она становится на одну функцию глубже. Первый вызов функции будет исполнен последним. Для нашей функции это выглядит так:

- рекурсивная функция `getRecursive(3)` получена;
  - рекурсивная функция `getRecursive(2)` получена;
    - рекурсивная функция `getRecursive(1)` получена;
      - рекурсивная функция `getRecursive(0)` получена;
      - выполнение `getRecursive(0)` завершено;
    - выполнение `getRecursive(1)` завершено;
  - выполнение `getRecursive(2)` завершено;
- выполнение `getRecursive(3)` завершено.

Следующая рекурсивная функция продемонстрирует данную схему:

```
function logRecursive(nr) {
  console.log("Started function:", nr);
  if (nr > 0) {
    logRecursive(nr - 1);
  } else {
     console.log("done with recursion");
  }
  console.log("Ended function:", nr);
}

logRecursive(3);
```

В консоли мы получим такой результат:

```
Started function: 3
Started function: 2
Started function: 1
Started function: 0
done with recursion
Ended function: 0
Ended function: 1
Ended function: 2
Ended function: 3
```

Для некоторых ситуаций рекурсивные функции будут отличным решением. Когда в цикле возникает необходимость вызывать одну и ту же функцию снова и снова, вам, вероятно, стоит подумать о рекурсии. Рекурсия также будет полезна при поиске чего-либо. Вместо того чтобы перебирать все внутри одной и той же функции, вы можете разделить функцию и многократно вызывать ее в теле функции.

Однако следует иметь в виду, что в целом производительность рекурсии немного хуже, чем у обычной итерации цикла. Поэтому, если использование рекурсии приведет к возникновению узких мест, которые действительно рискуют замедлить работу вашего приложения, стоит поискать другой подход.

Взгляните на вычисление факториала с использованием рекурсивных функций в следующем упражнении.

Практическое занятие 6.6

Распространенной проблемой, которую мы можем решить с помощью рекурсии, является вычисление факториала.

Кратко освежим математические знания о факториалах.

Факториал числа — это произведение всех положительных целых чисел больше 0 вплоть до самого числа. Небольшой пример: факториал семи равен $7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$. Записывается как 7!.

Как рекурсивные функции помогут нам найти факториал? В этом упражнении мы будем использовать рекурсию для вычисления результата факториала числового значения, заданного в качестве аргумента функции. Мы собираемся вызывать функцию с понижением значения числа, пока не достигнем `0`.

1. Создайте функцию, которая содержит условие, проверяющее, равно ли значение аргумента `0`.
2. Если параметр равен `0`, функция должна возвращать значение `1`. В противном случае она должна возвращать значение аргумента, умноженное на значение, возвращаемое самой функцией, вычитая единицу из значения предоставленного аргумента. Это приведет к запуску блока кода, который будет работать до тех пор, пока значение не станет равным `0`.
3. Вызовите функцию, предоставив аргумент любого числа, факториал которого вы хотите узнать. Код должен работать с любым числом, первоначально переданным в функцию, уменьшая значение до `0` и выводя результаты вычисления на экран. Вы также можете включить команду `console.log()` для вывода на экран текущего значения аргумента функции в момент ее вызова.
4. Измените число, чтобы увидеть, как оно влияет на результаты работы функции.

Вложенные функции

Как и в случае с циклами, оператором `if`, да и всеми другими блоками, мы можем создавать функции внутри функций. Такие функции называются *вложенными*:

```
function doOuterFunctionStuff(nr) {
  console.log("Outer function");
  doInnerFunctionStuff(nr);
  function doInnerFunctionStuff(x) {
    console.log(x + 7);
    console.log("I can access outer variables:", nr);
  }
}

doOuterFunctionStuff(2);
```

Результат вывода на экран:

```
Outer function
9
I can access outer variables: 2
```

Как видите, внешняя функция вызывает вложенную функцию. У вложенной функции есть доступ к переменной родительской функции. Но есть одна загвоздка. Переменные, определенные во внутренней функции, имеют область видимости в функции. Это означает, что переменные доступны внутри функции, в которой определены; функция, в свою очередь, является внутренней. Таким образом, мы получим ошибку `ReferenceError`:

```
function doOuterFunctionStuff(nr) {
  doInnerFunctionStuff(nr);
  function doInnerFunctionStuff(x) {
    let z = 10;
  }
  console.log("Not accessible:", z);
}

doOuterFunctionStuff(2);
```

Как думаете, что делает следующий код?

```
function doOuterFunctionStuff(nr) {
  doInnerFunctionStuff(nr);
  function doInnerFunctionStuff(x) {
    let z = 10;
  }
}

doInnerFunctionStuff(3);
```

Мы также получим ошибку `ReferenceError`. Теперь `doInnerFunctionStuff()` определена внутри внешней функции, это означает, что она доступна только в области функции `doOuterFunctionStuff()`. Область видимости вне этой функции заканчивается.

Практическое занятие 6.7

Создайте цикл с обратным отсчетом, начинающимся с динамического значения `10`.

1. Задайте переменную `start` со значением `10`, которая будет использована в качестве начального значения цикла.
2. Создайте функцию, которая получает один аргумент — значение обратного отсчета.
3. В рамках функции выведите текущее значение обратного отсчета на экран.
4. Добавьте условие, чтобы проверить, является ли значение меньше `1`. Если да, верните функцию.
5. Добавьте условие, чтобы проверить, является ли значение обратного отсчета не меньше `1`, затем продолжайте цикл, вызывая функцию внутри себя.
6. Убедитесь, что вы добавили оператор декремента в обратный отсчет, чтобы предыдущее условие в конечном счете стало истинным и цикл завершился. Во время каждого цикла значение будет уменьшаться, пока не достигнет `0`.
7. Создайте второй обратный отсчет с условием, при котором значение должно быть больше `0`. Если условие верно, уменьшите значение обратного отсчета на 1.
8. Используйте `return` для возврата функции, которая будет вызываться снова и снова, пока условие не изменится на противоположное.
9. Когда вы отправляете новое значение обратного отсчета в качестве аргумента в функцию, убедитесь, что из цикла есть выход, используя ключевое слово `return` и условие, которое продолжает цикл, если оно удовлетворено.

Анонимные функции

До сих пор мы давали имена всем функциям. Но можно также создавать функции без названий, если хранить их внутри переменных. Такие функции называются *анонимными*. Вот неанонимная функция:

```
function doingStuffAnonymously() {
  console.log("Not so secret though.");
}
```

А вот что получится, если предыдущую функцию заменить на анонимную:

```
function () {
  console.log("Not so secret though.");
};
```

Как видите, функция не носит названия. Она анонимна. У вас может возникнуть вопрос: как же вызвать эту функцию? Ответ вам наверняка не понравится!

Чтобы вызвать анонимную функцию, нам придется сохранить ее в переменной. Можно это сделать следующим образом:

```
let functionVariable = function () {
  console.log("Not so secret though.");
};
```

Анонимную функцию можно вызвать, используя имя переменной:

```
functionVariable();
```

На экране мы увидим `Not so secret though.`.

Хотя данная конструкция JavaScript может показаться немного бесполезной, на самом деле она очень мощная. Хранение функций внутри переменных позволяет делать очень интересные вещи, типа передачи функций в качестве параметров. Эта концепция добавляет еще один абстрактный слой к кодированию. Она называется *обратным вызовом*, и мы обсудим ее в следующем разделе.

Практическое занятие 6.8

1. Задайте имя переменной и присвойте ей функцию. Создайте выражение функции с одним параметром, которое выводит предоставленный аргумент на экран.
2. Введите аргумент в функцию.
3. Создайте ту же самую функцию с помощью обычного ее объявления.

Функции обратного вызова

Вот пример передачи функции в качестве аргумента другой функции:

```
function doFlexibleStuff(executeStuff) {
  executeStuff();
  console.log("Inside doFlexibleStuffFunction.");
}
```

Если мы вызовем новую функцию с помощью нашей ранее созданной анонимной функции `functionVariable`, например, так:

```
doFlexibleStuff(functionVariable);
```

получим следующий результат:

```
Not so secret though.
Inside doFlexibleStuffFunction.
```

Но мы также можем вызвать ее с помощью другой функции. `doFlexibleStuff` отлично справится с данной задачей. Насколько это круто?

```
let anotherFunctionVariable = function() {
  console.log("Another anonymous function implementation.");
}

doFlexibleStuff(anotherFunctionVariable);
```

В результате вы получите:

```
Another anonymous function implementation.
Inside doFlexibleStuffFunction.
```

Что же произошло? Мы создали функцию и сохранили ее в переменной `anotherFunctionVariable`. Затем мы отправили ее в качестве параметра в нашу функцию `doFlexibleStuff()`. Эта функция просто будет выполнять любую функцию, которую вы в нее передадите.

Вы можете задаться вопросом, почему авторы так увлечены концепцией обратного вызова. Вероятно, в примерах, встречавшихся выше, мы вас не убедили. Позже мы перейдем к асинхронным функциям, и обратный вызов окажет нам серьезную помощь. Чтобы по-прежнему удовлетворять вашу потребность в более конкретных примерах, приведем вам следующий.

В JavaScript, как вы уже знаете, существует множество встроенных функций. Функция `setTimeout()` одна из них. Это особенная функция, которая выполняет определенную операцию после того, как подождет какое-то определенное время. По-видимому, это она ответственна за ужасную работу многих веб-страниц, но она однозначно не виновата в том, что ее неправильно поняли и использовали.

Следующий фрагмент кода действительно стоит попытаться понять:

```
let youGotThis = function () {
  console.log("You're doing really well, keep coding!");
};

setTimeout(youGotThis, 1000);
```

Программа должна ожидать 1000 миллисекунд (одну секунду), а после выводить на экран:

```
You're doing really well, keep coding!
```

Если требуется более мощный стимул, можно использовать функцию `setInterval()`. Она работает очень похоже на `setTimeout()`, только выполняет указанную операцию не один раз, а несколько с заданным интервалом:

```
setInterval(youGotThis, 1000);
```

В этом случае функция будет выводить в консоль наше ободряющее сообщение каждую секунду, пока вы не завершите работу программы.

Концепция функции, выполняющей функцию после того, как была вызвана сама собой, очень полезна для управления асинхронным выполнением программы.

Проекты текущей главы

Создание рекурсивной функции

Создайте рекурсивную функцию, которая считает до 10. Вызовите функцию с разными начальными значениями в качестве аргументов, которые передаются в функцию. Функция должна выполняться до тех пор, пока значение не станет больше 10.

Использование функции setTimeout()

Используйте стрелочный формат для создания функций, которые будут выводить значения `one` и `two` на экран. Создайте третью функцию, которая будет выводить в консоль значение `three` и после этого вызывать две предыдущие функции.

Создайте четвертую функцию, которая выводит на экран слово `four`, и примените `setTimeout()` для немедленного вызова сначала первой функции, а после нее — третьей функции.

Что у вас получилось? Попробуйте заставить функцию выводить на экран:

```
Four
Three
One
Two
One
```

Вопросы для самопроверки

1. Какое значение будет выведено на экран?

```
let val = 10;
function tester(val){
    val += 10;
    if(val < 100){
        return tester(val);
    }
    return val;
}
tester(val);
console.log(val);
```

2. Что выведет в консоль следующий код?

```
let testFunction = function(){
    console.log("Hello");
}();
```

3. Какой результат появится на экране?

```
(function () {
    console.log("Welcome");
})();
(function () {
    let firstName = "Laurence";
})();
let result = (function () {
    let firstName = "Laurence";
    return firstName;
})();
console.log(result);
(function (firstName) {
    console.log("My Name is " + firstName);
})("Laurence");
```

4. Какое значение будет выведено на экран?

```
let test2 = (num) => num + 5;
console.log(test2(14));
```

5. Какой результат получится после выполнения данного кода?

```
var addFive1 = function addFive1(num) {
return num + 2;
};
let addFive2 = (num) => num + 2;
console.log(addFive1(14));
```

Резюме

В этой главе мы рассмотрели функции. Это отличный строительный блок JavaScript, который мы можем использовать для повторного запуска конкретного фрагмента кода. Функциям можно задавать параметры, чтобы изменять код в зависимости от аргументов, с которыми вызывается функция. Функции могут возвращать результат. Это осуществляется с помощью ключевого слова `return`. Также можно добавлять `return` в том месте, где мы вызываем функцию. Результат функции можно сохранить в переменной или использовать повторно, например, для другой функции.

Далее вы познакомились с областью видимости переменных. Область видимости включает в себя места, откуда доступны переменные. По умолчанию переменные `let` и `const` доступны внутри блока, в котором были определены (а также внутри вложенных блоков). А переменная `var` доступна везде, начиная с места, где была определена.

Мы также можем использовать рекурсивные функции для элегантного решения тех проблем, к которым рекурсивный подход применяется и в обычной жизни (например, для вычисления факториала). Следующей нашей темой были вложенные функции. В них нет ничего особенного: это просто функции внутри функций. Обычно их считают не очень красивыми, но анонимные и стрелочные функции встречаются не так уж и редко. Анонимные функции — это функции без имени, а стрелочные функции — их частный случай; там мы используем стрелку для разделения параметров и тела.

В следующей главе мы рассмотрим классы — еще одну мощную программную конструкцию.

7 Классы

В этой главе мы обсудим классы в JavaScript. Вы уже встречались с объектами JavaScript; классы — это схема, или шаблон, для их создания. Так что большинство обсуждаемых далее вещей должны звучать достаточно знакомо и не революционно.

Классы обеспечивают объектно-ориентированное программирование — одну из наиболее важных концепций разработки. Данный подход значительно снизил сложность приложений и повысил удобство их обслуживания.

Объектно-ориентированное программирование и классы имеют огромное значение для информатики в целом. Но в JavaScript они не обязательно будут работать так же, как в других языках программирования. В отличие от других систем классы в JavaScript — это нечто особенное. На самом деле здесь классы считаются неким особым видом функции. Это означает, что фактически они задают альтернативный синтаксис для определения объектов с помощью функции конструктора. В этой главе мы узнаем, что такое классы и как их можно создавать и использовать.

Мы рассмотрим следующие темы:

- объектно-ориентированное программирование;
- классы и объекты;
- классы;
- наследование;
- прототипы.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Объектно-ориентированное программирование

Прежде чем мы начнем погружаться прямиком в увлекательные классы, давайте кратко поговорим об *объектно-ориентированном программировании (ООП)*. ООП — очень важная парадигма программирования, в которой код структуриру-

ется в виде объектов, что приводит к более удобному обслуживанию и многократному использованию кода. Работа с ООП учит вас по-настоящему продумывать всевозможные ключевые моменты в объектах, объединив свойства таким образом, чтобы их можно было сгруппировать в схеме, называемой *классом*, который, в свою очередь, может наследовать свойства родительского класса.

Например, когда мы берем животное, мы можем описать его определенные свойства: вид, вес, рост, максимальную скорость, цвет и многое другое. Если после этого нам нужно будет охарактеризовать конкретный вид рыбы, мы можем рассмотреть все свойства «животного», добавив туда также несколько свойств, специфичных для рыбы. То же самое справедливо и для собак: мы применяем все свойства «животного» и добавляем к ним несколько характеристик, специфичных для собак. Таким образом, у нас есть повторно используемый код нашего класса «животное». И если вдруг мы забыли какое-то свойство, очень важное для многих животных в приложении, останется только добавить его в класс «животные».

Данный подход очень важен для Java.NET и других классических объектно-ориентированных способов написания кода. JavaScript не обязательно вращается вокруг объектов. Они нам понадобятся, и мы будем их использовать, но они, так сказать, не являются звездами нашего кода.

Классы и объекты

Кратко напомним: объекты представляют собой набор свойств и методов. Мы рассматривали их в главе 3. Свойства объекта должны иметь разумные имена. Так, например, если у нас есть объект `person` (человек), у него должны быть свойства с названиями `age` (возраст) и `lastName` (фамилия) и заданными значениями. Перед вами пример объекта:

```
let dog = { dogName: "JavaScript",
            weight: 2.4,
            color: "brown",
            breed: "chihuahua"
          };
```

Классы в JavaScript инкапсулируют данные и функции, которые являются частью этого класса. Если вы создадите класс, с его помощью вы сможете позже работать над объектами, используя следующий синтаксис:

```
class ClassName {
  constructor(prop1, prop2) {
    this.prop1 = prop1;
    this.prop2 = prop2;
  }
}

let obj = new ClassName("arg1", "arg2");
```

Код определяет класс с именем `ClassName`, объявляет переменную `obj` и делает ее новым экземпляром объекта. Приводятся два аргумента, они будут использоваться конструктором для инициализации свойств. Как вы заметили, параметры для конструктора и свойства класса (`prop1` и `prop2`) носят одинаковые названия. Свойства класса можно распознать по ключевому слову `this` перед ними. Ключевое слово `this` относится к объекту, которому принадлежит, поэтому является первым свойством экземпляра `ClassName`.

Помните, мы говорили, что классы — это на самом деле просто особая функция. Мы могли бы создать объект с помощью специальной функции, типа этой:

```
function Dog(dogName, weight, color, breed) {
  this.dogName = dogName;
  this.weight = weight;
  this.color = color;
  this.breed = breed;
}

let dog = new Dog("Jacky", 30, "brown", "labrador");
```

Пример с собакой также можно реализовать с использованием синтаксиса класса. Выглядит это так:

```
class Dog {
  constructor(dogName, weight, color, breed) {
    this.dogName = dogName;
    this.weight = weight;
    this.color = color;
    this.breed = breed;
  }
}

let dog = new Dog("JavaScript", 2.4, "brown", "chihuahua");
```

В результате получается объект с теми же свойствами. Мы сможем это увидеть, если проведем логирование следующим образом:

```
console.log(dog.dogName, "is a", dog.breed, "and weighs", dog.weight, "kg.");
```

Результат вывода на экран:

```
JavaScript is a chihuahua and weighs 2.4 kg.
```

В следующем разделе мы погрузимся во все детали классов.

Классы

Вы можете задаться вопросом: если классы делают то же самое, что и простое определение объекта, зачем они вообще нужны? Ответ заключается в том, что классы, по сути, являются схемами для объектов. Если нужно создать 20 записей собак,

придется писать гораздо меньше кода, когда у нас есть класс dog. При описывании каждого объекта по отдельности нам придется каждый раз указывать имена всех свойств. И можно легко сделать опечатку или неправильно прописать свойство. Классы пригодятся в подобных ситуациях.

Как показано в предыдущем разделе, мы используем ключевое слово class, чтобы сказать JavaScript, что хотим создать класс. Далее мы присваиваем имя классу. По общепринятым правилам имена классов начинаются с заглавной буквы.

Давайте рассмотрим различные элементы классов.

Метод constructor

Особый метод constructor используется для инициализации объектов с помощью классов. В классе может быть только один метод constructor. Он содержит свойства, которые будут установлены при инициализации класса.

Ниже вы можете рассмотреть пример использования метода constructor при создании класса Person:

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}
```

Под внешней оболочкой JavaScript инициализирует специальную функцию, основанную на методе constructor. Эта функция получает имя класса и создает объект с заданными свойствами. С помощью данной специальной функции вы можете прописывать экземпляры (объекты) класса.

А теперь перед вами код, создающий новый объект на основе класса Person:

```
let p = new Person("Maaike", "van Putten");
```

Именно слово new дает понять JavaScript, что требуется искать специальную функцию конструктора в классе Person и создавать новый объект. Конструктор вызывается и возвращает экземпляр объекта person с заданными свойствами. Этот объект сохраняется в переменной p.

Если использовать новую переменную p для логирования, можно увидеть, что свойства действительно установлены:

```
console.log("Hi", p.firstname);
```

Так будет выглядеть результат:

```
Hi Maaike
```

Как думаете, что произойдет, если создать класс без каких-либо свойств? Давайте выясним:

```
let p = new Person("Maaike");
```

Многие языки программирования выдали бы ошибку, но не JavaScript. Он просто присвоит оставшимся свойствам значение `undefined`. Давайте понаблюдаем за тем, что происходит, логируя результат:

```
console.log("Hi", p.firstname, p.lastname);
```

В консоли отобразится следующая запись:

```
Hi Maaike undefined
```

В методе `constructor` вы можете задать значения по умолчанию. Реализовать это можно следующим образом:

```
constructor(firstname, lastname = "Doe") {
    this.firstname = firstname;
    this.lastname = lastname;
  }
```

В результате на экран будет выведено не `Hi Maaike undefined`, а `Hi Maaike Doe`.

Практическое занятие 7.1

Выполните следующие действия, чтобы создать класс `person` и вывести на экран экземпляры имен друзей.

1. Создайте класс `Person`, включающий конструктор для `firstname` и `lastname`.
2. Создайте переменную и присвойте значение новому объекту `Person`, используя имя и фамилию вашего первого друга.
3. Теперь добавьте вторую переменную с именем второго друга, используя его имя и фамилию.
4. Выведите на экран обе записи с приветствием `hello`.

Методы

В классе можно указывать функции. Благодаря этому наш объект может начать выполнять действия, используя собственные свойства, — например, выводить имя на экран. Функции в классе называются методами. Для определения таких методов не используется ключевое слово `function`. Мы сразу начинаем писать имя:

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  greet() {
    console.log("Hi there! I'm", this.firstname);
  }
}
```

Вызвать метод `greet` для объекта `Person` можно следующим образом:

```
let p = new Person("Maaike", "van Putten");
p.greet();
```

Полученный результат:

```
Hi there! I'm Maaike
```

В классе можно указывать любое количество методов. В данном примере мы используем свойство `firstname`. Мы вызываем его с помощью `this.property`. Для человека с другим значением `firstname`, например `Rob`, на экран будет выведено следующее:

```
Hi there! I'm Rob
```

Методы, как и функции, могут принимать параметры и возвращать результаты:

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  greet() {
    console.log("Hi there!");
  }

  compliment(name, object) {
    return "That's a wonderful " + object + ", " + name;
  }
}
```

Функция `compliment` сама не выводит никаких значений, поэтому мы логируем ее:

```
let compliment = p.compliment("Harry", "hat");
console.log(compliment);
```

Результат будет следующим:

```
That's a wonderful hat, Harry
```

В данном случае мы отправляем параметры методу, потому что обычно не хвалим сами себя (отличное предложение, Майке!). Однако всякий раз, когда метод не требует внешнего ввода, а только свойств объекта, никакие параметры не будут работать, и метод может использовать именно свойства объекта. Выполним упражнение, а затем перейдем к использованию свойств классов вне класса.

Практическое занятие 7.2

Получите полное имя своего друга.

1. Используйте класс `Person` из практического занятия 7.1, добавив метод с названием `fullname`, который будет возвращать совокупное значение `firstname` и `lastname`.
2. Создайте значения для `person1` и `person2`, используя фамилии и имена друзей.
3. Используя метод `fullname` внутри класса, верните полное имя одного или обоих человек.

Свойства

Свойства (иногда называемые полями) содержат данные класса. В конструкторах мы уже видели один тип свойств:

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}
```

В данном случае класс `Person` получает два свойства от конструктора: `firstname` и `lastname`. Свойства можно добавлять и удалять, как мы проделывали это с объектами. К свойствам можно получить доступ за пределами класса, мы видели подобное, когда логировали значения вне класса, получая доступ к свойствам в экземпляре:

```
let p = new Person("Maaike", "van Putten");
console.log("Hi", p.firstname);
```

Прямой доступ к свойствам лучше не предоставлять. Мы хотим, чтобы класс контролировал значения свойств, по нескольким причинам — например, нам необходимо убедиться, что свойство имеет требуемое значение. Допустим, вы хотите подтвердить, что возраст пользователя больше 18 лет. Можно достичь этого, сделав невозможным прямой доступ к свойству за пределами класса.

Вот как добавить свойства, недоступные извне. Ставим перед ними символ `#`:

```
class Person {
  #firstname;
  #lastname;
  constructor(firstname, lastname) {
    this.#firstname = firstname;
    this.#lastname = lastname;
  }
}
```

В данный момент к свойствам `firstname` и `lastname` доступа извне нет. Если попробуем ввести:

```
let p = new Person("Maria", "Saga");
console.log(p.firstname);
```

то получим:

```
undefined
```

Если бы мы хотели убедиться, что можем создавать объекты только с именами, начинающимися на букву M, мы могли бы немного изменить конструктор:

```
constructor(firstname, lastname) {
    if(firstname.startsWith("M")){
      this.#firstname = firstname;
    } else {
      this.#firstname = "M" + firstname;
    }
    this.#lastname = lastname;
  }
```

Теперь при попытке ввести значение `firstname`, которое не начинается на M, код добавит `M` в начало. Так, например, в консоли значение следующего имени будет отображено как `Mkay`:

```
let p = new Person("kay", "Moon");
```

Это очень простой пример проверки. На данный момент мы вообще не можем получить доступ к свойству извне класса после конструктора. Оно доступно только изнутри класса. Вот тут-то и вступают в игру геттеры и сеттеры.

Геттеры и сеттеры

Геттеры и сеттеры — это особые свойства, которые можно использовать для получения данных из класса и записи полей данных в классе. Геттеры и сеттеры являются вычисляемыми свойствами. Можно сказать, что они даже больше похожи на свойства, чем на функции. Мы называем их *аксессорами* (accessors). Геттеры и сеттеры выглядят как функции, потому что после них ставят скобки (`()`), но функциями они не являются!

Аксессоры начинаются с ключевых слов `get` и `set`. Хорошей практикой считается максимально закрывать поля и предоставлять доступ к ним с помощью геттеров и сеттеров. Благодаря этому свойства не могут быть заданы извне без контроля самого объекта. Подобный принцип называется *инкапсуляцией*. Класс *инкапсулирует* (помещает) данные и объект под контроль собственных полей.

Вот как это реализовать:

```
class Person {
  #firstname;
  #lastname;
  constructor(firstname, lastname) {
    this.#firstname = firstname;
    this.#lastname = lastname;
  }

  get firstname() {
    return this.#firstname;
  }

  set firstname(firstname) {
    this.#firstname = firstname;
  }

  get lastname() {
    return this.#lastname;
  }

  set lastname(lastname) {
    this.#lastname = lastname;
  }
}
```

Геттер используется для получения свойства. Поэтому он не берет никаких параметров, а просто возвращает свойство. Сеттер действует наоборот: принимает параметр, присваивает новое значение свойству и ничего не возвращает. Сеттер может включать в себя больше логики, например проводить какую-то проверку, как мы увидим ниже. Геттер можно использовать вне объекта, как если бы он был

свойством. Свойства больше не доступны напрямую извне класса, но к ним можно получить доступ через геттер, возвращающий значение, и через сеттер, присваивающий значение. Вот как можно это реализовать вне экземпляра класса:

```
let p = new Person("Maria", "Saga");
console.log(p.firstname);
```

Результат вывода на экран:

```
Maria
```

Мы создали новый объект `Person` с именем `Maria` и фамилией `Saga`. На выходе отображается имя, что стало возможным только благодаря наличию у нас средства доступа — геттера. Мы также можем задать значение чему-то другому, используя сеттер. Давайте обновим значение имени так, что теперь это будет не `Maria`, а `Adnane`:

```
p.firstname = "Adnane";
```

На данный момент в сеттере не происходит ничего особенного. Мы могли бы выполнить такую же проверку, как и в предыдущем конструкторе, например:

```
set firstname(firstname) {
    if(firstname.startsWith("M")){
      this.#firstname = firstname;
    } else {
      this.#firstname = "M" + firstname;
    }
  }
```

Теперь код проверит, начинается ли `firstname` с буквы M. Если да, то значение `firstname` обновится до значения параметра. Если нет, код добавит M перед параметром.

Обратите внимание, что мы не обращаемся к `firstname` так, как если бы это была функция. Если вы допишете круглые скобки `()`, то получите сообщение об ошибке, говорящей, что это не функция.

Наследование

Наследование — одна из ключевых концепций ООП. Согласно ей классы могут иметь дочерние классы, которые наследуют свойства и методы родительского класса. Например, если вам понадобятся все виды транспортных средств в приложении, вы можете задать класс с названием `Vehicle`, в котором укажете некоторые общие свойства и методы объектов. Для продолжения можете создавать дочерние классы, основанные на классе `Vehicle`, например `boat`, `car`, `bicycle` и `motorcycle`.

Вот сильно упрощенная версия класса `Vehicle`:

```
class Vehicle {
  constructor(color, currentSpeed, maxSpeed) {
    this.color = color;
    this.currentSpeed = currentSpeed;
    this.maxSpeed = maxSpeed;
  }

  move() {
    console.log("moving at", this.currentSpeed);
  }

  accelerate(amount) {
    this.currentSpeed += amount;
  }
}
```

Здесь представлены два метода в классе `Vehicle`: `move` и `accelerate`. Это может быть класс `Motorcyle`, наследуемый от данного класса с использованием ключевого слова `extends`:

```
class Motorcycle extends Vehicle {
  constructor(color, currentSpeed, maxSpeed, fuel) {
    super(color, currentSpeed, maxSpeed);
    this.fuel = fuel;
  }
  doWheelie() {
    console.log("Driving on one wheel!");
  }
}
```

С помощью ключевого слова `extends` мы указываем, что определенный класс является дочерним по отношению к другому. В данном случае `Motorcycle` — дочерний класс `Vehicle`. Это значит, что у нас будет доступ к свойствам и методам класса `Vehicle` в классе `Motorcycle`. Сюда же мы добавили особый метод `doWheelie()` для описания езды на заднем колесе. Включать данную возможность в класс `Vehicle` не было смысла, потому что она специфична для определенных транспортных средств.

Слово `super` в конструкторе вызывает конструктор родителя — в данном случае конструктор `Vehicle`. Это гарантирует, что поля от родителя также заданы и что методы доступны без необходимости делать что-либо еще: они наследуются автоматически. Вызов `super()` не является обязательным, но вы должны использовать это ключевое слово при нахождении в классе, который наследуется от другого класса — иначе получите `ReferenceError`.

В классе `Motorcycle` у нас есть доступ к полям класса `Vehicle`, поэтому следующая конструкция будет работать:

```
let motor = new Motorcycle("Black", 0, 250, "gasoline");
console.log(motor.color);
motor.accelerate(50);
motor.move();
```

В результате вы увидите:

```
Black
moving at 50
```

Мы не можем получить доступ к каким-либо конкретным свойствам или методам класса `Motorcycle` в классе `Vehicle`: не все транспортные средства являются мотоциклами, поэтому мы не можем быть уверены, что при описании нашего конкретного объекта понадобятся дочерние свойства или методы.

Сейчас мы не используем никаких геттеров и сеттеров, но определенно могли бы. Если в родительском классе есть геттеры и сеттеры, они также наследуются дочерним классом. Следовательно, мы могли бы влиять на то, какие свойства могут быть извлечены и изменены (и как) за пределами нашего класса. Обычно это хорошая практика.

Прототипы

Прототипы — это механизмы JavaScript, которые делают возможным существование объектов.

Если при создании класса никакие детали специально не уточняются, объекты наследуются от прототипа `Object.prototype`. Среди всех встроенных классов JavaScript, которые мы можем применять, этот довольно сложный. Нет необходимости изучать, как он реализован. Его можно считать базовым объектом, который всегда находится на вершине дерева наследования, а значит, всегда присутствует в наших объектах.

Для всех классов доступно свойство `prototype`, и оно всегда называется «прототип». Вызвать его можно так:

```
ClassName.prototype
```

Давайте приведем пример того, как добавить функцию в класс, используя свойство `prototype`. Для этого мы будем использовать класс `Person`:

```
class Person {
  constructor(firstname, lastname) {
```

```
    this.firstname = firstname;
    this.lastname = lastname;
  }

  greet() {
    console.log("Hi there!");
  }
}
```

И вот как добавить функцию в этот класс, используя `prototype`:

```
Person.prototype.introduce = function () {
  console.log("Hi, I'm", this.firstname);
};
```

Свойство `prototype` содержит в себе все методы и свойства объекта. Таким образом, добавление функции в `prototype` — это добавление функции в класс. Вы можете использовать `prototype`, чтобы добавлять свойства или методы объекту (как мы сделали выше с помощью функции `introduce`). Данный способ применим и для свойств:

```
Person.prototype.favoriteColor = "green";
```

После их можно вызывать из экземпляра класса `Person`:

```
let p = new Person("Maria", "Saga");
console.log(p.favoriteColor);
p.introduce();
```

Мы получим следующий результат:

```
green Hi, I'm Maria
```

Это выглядит точно так же, как если бы вы определили класс с любимым цветом, имеющим значение по умолчанию, и функцией `introduce`. Они были добавлены в класс и теперь доступны для всех экземпляров, существующих и будущих.

Следовательно, методы и свойства, определенные через `prototype`, точно такие же, как если бы они были определены в классе. Это означает, что, перезаписывая их для определенного экземпляра, вы не перезаписываете методы и свойства всех экземпляров. Например, если бы у нас был второй объект `Person`, он мог бы перезаписать значение `favoriteColor`, и это не изменило бы значение любимого цвета для объекта со значением `firstname`, равным `Maria`.

Прототипы не следует применять, если у вас есть контроль над кодом класса и вы хотите изменить его навсегда, — в таком случае просто измените класс. Однако с помощью прототипов вы можете расширить существующие объекты, в том числе с применением условий. Важно также знать, что встроенные объекты JavaScript имеют прототипы и наследуют от `Object.prototype`. Не меняйте данный прототип, иначе это повлияет на работу JavaScript.

Практическое занятие 7.3

Напишите класс, содержащий свойства для разных животных — в том числе звуки, издаваемые каждым видом — и создайте два (или более) объекта.

1. Создайте метод, который выводит на экран название этого животного и звук, который оно издает.
2. Добавьте прототип с другим действием для животного.
3. Выведите весь объект животного на экран.

Проекты текущей главы

Приложение для контроля сотрудников

Создайте класс для контроля сотрудников компании.

1. Используйте имена, фамилии и количество отработанных лет в качестве значений в конструкторе.
2. Создайте записи о двух или более сотрудниках со значениями их имен, фамилий и количества лет, которые они проработали в компании. Добавьте этих людей в массив.
3. Создайте прототип, чтобы возвращать данные об имени и фамилии человека и о том, как долго он проработал в компании.
4. Повторите содержимое массива для вывода результатов на экран, добавив немного текста, чтобы на выходе получилось полноценное предложение.

Расчет стоимости заказов

Создайте класс, который позволит рассчитать совокупную цену ряда товаров и взаимодействовать с ней, чтобы узнать общую стоимость различных заказов.

1. Создайте класс, содержащий цены двух пунктов прейскуранта в качестве объявлений частных полей.
2. Используйте конструктор в классе, чтобы получить значения аргументов (сколько покупается каждого товара).
3. Создайте метод для расчета и возврата общей стоимости в зависимости от того, сколько товаров выбирает пользователь.
4. Используйте геттер, чтобы получить значение, выводимое методом расчета.
5. Создайте два или три объекта с различными комбинациями пунктов прейскуранта и выведите общую стоимость на экран.

Вопросы для самопроверки

1. Какое ключевое слово используется для создания классов?
2. Как бы вы написали класс для имени и фамилии человека, содержащий `first` и `last` в качестве начальных свойств?
3. Как называется концепция, при которой один объект приобретает свойства и поведение другого объекта?
4. Какие из следующих утверждений в отношении метода `constructor` справедливы:
   - выполняется автоматически при создании нового объекта;
   - следует добавлять только последним;
   - обязан включать в себя ключевое слово `constructor`;
   - используется для инициализации свойств объектов;
   - может быть использован для нескольких значений?
5. Устраните ошибки в следующем коде, чтобы прототип выводил имя и фамилию экземпляра `Person` на экран. Каким должен быть корректный синтаксис для прототипа `Person`?

```
function Person(first,last) {
  this.first = first;
  this.last = last;
}
// Что указывается далее: А, Б или В?
const friend1 = new Person("Laurence", "Svekis");
console.log(friend1.getName());
```

1)

```
Person.prototype.getName = (first,last) {
  return this.first + " " + this.last;
};
```

2)

```
Person.prototype.getName = function getName() {
  return this.first + " " + this.last;
};
```

3)

```
Person.prototype = function getName() {
  return this.first + " " + this.last;
};
```

Резюме

В этой главе мы познакомили вас с концепцией ООП, согласно которой код структурируется таким образом, чтобы объекты были основными участниками логики. Классы — это оболочки объектов. Мы можем легко задать шаблон для объекта и создать экземпляр, используя ключевое слово `new`.

Затем вы узнали, что классы могут наследовать друг друга с помощью ключевого слова `extends`. Классы, которые являются дочерними по отношению к другому классу, расширяя набор свойств, должны вызывать конструктор этого класса с помощью функции `super()` — так они автоматически получают доступ ко всем свойствам и методам родительского класса. Данный подход позволяет применять код много раз и делает его легко поддерживаемым.

Наконец, вы встретились с прототипами. Это встроенная концепция JavaScript, которая делает возможным существование классов. Добавляя свойства и методы в класс с помощью `prototype`, мы можем изменять схему этого класса.

В следующей главе мы рассмотрим некоторые встроенные методы JavaScript, которые можно использовать для управления и усложнения вашего кода.

8 Встроенные методы JavaScript

Мы рассмотрели большинство основных строительных блоков в JavaScript. Пришло время взглянуть на некоторые мощные встроенные методы, с которыми вы пока не знакомы, но которые могут облегчить вашу жизнь. Встроенные методы — это функционал JavaScript, который мы получаем «из коробки». Мы можем использовать эти методы сразу, без необходимости их предварительного кодирования. Вы уже видели такие конструкции — например, `console.log()` и `prompt()`.

Многие встроенные методы также относятся к встроенным классам. Эти классы и их методы могут быть использованы в любое время, поскольку JavaScript уже определил их. Эти очень распространенные и нужные элементы существуют для нашего удобства, например классы `Date`, `Array` и `Object`.

Использование встроенных возможностей JavaScript повышает эффективность кода, экономит время и соответствует различным передовым практикам разработки. Мы рассмотрим некоторые распространенные сферы применения данного функционала: обработку текста, математические вычисления, работу со значениями даты и времени, взаимодействия и построение надежного кода. Вот темы, которые мы изучим в текущей главе:

- глобальные методы JavaScript;
- строчные методы;
- математические методы;
- методы работы с датами;
- методы работы с массивами;
- числовые методы.

Решения упражнений и проектов, а также вопросы для самопроверки находятся в приложении.

Введение во встроенные методы JavaScript

Мы уже рассмотрели множество встроенных методов JavaScript. Любой метод, который мы не определяли сами, является встроенным. Это `console.log()`, `Math.random()`, `prompt()` и многое другое, например методы для работы с массивами. Разница между методом и функцией заключается в том, что функция определяется в любом месте скрипта, а метод — внутри класса. Таким образом, методы в значительной степени являются функциями классов и экземпляров.

Методы также часто могут быть связаны в цепочку, но это справедливо только для методов, возвращающих результат. Следующий метод как раз такой:

```
let s = "Hello";
console.log(
  s.concat(" there!")
  .toUpperCase()
  .replace("THERE", "you")
  .concat(" You're amazing!")
);
```

В первой строке мы объявляем переменную `s` и записываем в нее значение `Hello`. Затем мы хотим что-то логировать. Данный код был разделен на разные строки для удобства чтения, но на самом деле это одно выражение. Сначала мы применим метод `concat()` к переменной `s`, который добавляет строчное значение к уже записанному. После этой операции значение примет вид `Hello there!`. Затем мы запишем строку всеми прописными с помощью следующего метода. Значение примет вид `HELLO THERE!`. Далее мы заменим `THERE` на `you`. Значение примет вид `HELLO you!`. После чего мы снова добавим к нему строку — и наконец значение будет выведено на экран:

```
HELLO you! You're amazing!
```

В этом примере нам нужно записать или сохранить выходные данные, потому что исходное строковое значение не будет обновлено простым вызовом методов для строки.

Глобальные методы

Глобальные методы JavaScript можно использовать без ссылки на встроенный объект, частью которого они являются. Мы можем просто взять имя метода, как если бы это была функция, определенная внутри области, где мы находимся, без указания объекта перед ней. Например, вместо того, чтобы писать:

```
let x = 7;
console.log(Number.isNaN(x));
```

можно просто задать:

```
console.log(isNaN(x));
```

`Number` можно убрать, потому что `isNaN` доступен глобально, без ссылки на класс, к которому он принадлежит (в этом экземпляре — класс `Number`). В таком случае оба оператора `console.log` зафиксируют значение `false` (фактически они делают одно и то же), потому что `isNaN` возвращает значение `true`, когда значение не является числом. А `7` — это число.

JavaScript был создан для того, чтобы методы были доступны напрямую, поэтому для достижения цели необходимо было какое-то волшебство. Создатели JavaScript выбрали, по их собственному мнению, наиболее распространенные методы, поэтому причины, по которым некоторые из них доступны как глобальные, а другие — нет, могут показаться немного произвольными. Это просто выбор нескольких великолепных разработчиков, сделанный в определенный момент времени.

Ниже мы рассмотрим наиболее распространенные глобальные методы. Начнем с декодирования и кодирования URI, методов `escape` и `unescape`; затем будем сравнивать числа и, наконец, рассмотрим метод `eval`.

Декодирование и кодирование URI

Кодирование — простая конвертация из одной формы в другую. Далее мы будем иметь дело с процентным кодированием, также называемым *кодированием URL*. Но прежде, чем мы начнем, стоит устранить некоторую путаницу в понятиях URI и URL. *URI (uniform resource identifier)* — идентификатор определенного ресурса. *URL (uniform resource locator)* — это подкатегория URI, которая является не только идентификатором, но также содержит информацию о том, как получить доступ к ресурсу (location).

Давайте поговорим о кодировании и декодировании URI (а также URL, раз это подгруппа). Вам однозначно понадобится данная тема, например, при отправке переменных по URL-адресу с использованием метода `get` в форме. Переменные, которые вы отправляете по URL-адресу, называются *параметрами запроса*.

Использовать пробелы в URL-адресе нельзя, поэтому, если вы их где-то поставите, они будут декодированы как `%20`. В итоге URL может выглядеть так:

```
www.example.com/submit?name=maaike%20van%20putten&love=coding
```

Все знаки можно перевести в `%`-ный формат. Однако в большинстве случаев в этом нет необходимости. URI могут содержать некоторое число буквенно-цифровых символов, а вот специальные символы подлежат кодированию. Пример до использования кодирования:

```
https://www.example.com/submit?name=maaike van putten
```

Тот же адрес URL, но с применением кодирования:

```
https://www.example.com/submit?name=maaike%20van%20putten
```

Существует две пары методов кодирования и декодирования. Мы обсудим эти методы и варианты их применения. У вас не может быть URI с пробелами, поэтому использование этих методов имеет решающее значение для работы с переменными, содержащими пробелы.

Методы decodeUri() и encodeUri()

`decodeUri()` и `encodeUri()` — это на самом деле не «кодирование» и «декодирование», а, скорее, «исправление» неработающих URI (как в предыдущем примере с пробелами). Эта пара методов действительно полезна при исправлении неработающих URI и декодировании их обратно в строку. Рассмотрим их в действии:

```
let uri = "https://www.example.com/submit?name=maaike van putten";
let encoded_uri = encodeURI(uri);
console.log("Encoded:", encoded_uri);
let decoded_uri = decodeURI(encoded_uri);
console.log("Decoded:", decoded_uri);
```

Результат будет следующим:

```
Encoded: https://www.example.com/submit?name=maaike%20van%20putten
Decoded: https://www.example.com/submit?name=maaike van putten
```

Как видите, код заменил пробелы в закодированной версии и снова удалил их в декодированной. Все остальные символы остаются неизменными — кодирование и декодирование не учитывает специальные символы, поэтому оставляет их в URI. Можно также использовать двоеточия, вопросительные знаки, знаки равенства, слеши и амперсанды.

Описанный выше способ подходит для исправления неработающих URI, но бесполезен, когда необходимо закодировать строки, содержащие любой из этих символов: `/ , ? : @ & = + $ #`. Они могут быть использованы как составная часть URI и потому пропускаются. Вот где пригодятся следующие два встроенных метода.

Методы decodeUriComponent() и encodeUriComponent()

Итак, методы `decodeURI()` и `encodeURI()` очень полезны при исправлении некорректных URI. Но они не помогут, когда вы хотите закодировать или декодировать строку, содержащую символ со специальным значением (такой как `=` или `&`). Возьмем следующий пример:

```
https://www.example.com/submit?name=this&that=some thing&code=love
```

Согласны, значение странное, но оно хорошо демонстрирует описываемую проблему. Использование `encodeURI` в этом случае даст нам:

```
https://www.example.com/submit?name=this&that=some%20thing&code=love
```

В соответствии со стандартами URI здесь на самом деле есть три переменные:

- `name` (со значением `this`);
- `that` (со значением `some thing`);
- `code` (со значением `love`).

Однако мы намеревались отправить только одну переменную `name`, содержащую значение `this&that=some thing&code=love`.

В таком случае потребуются `decodeUriComponent()` и `encodeUriComponent()` — с их помощью `=` и `&` заносятся в закодированную переменную. Пока все идет не так, и это вызовет проблемы при интерпретации параметров запроса (переменных после `?`). Мы хотели отправить только один параметр — `name`. Но вместо этого отправили три.

Рассмотрим другой случай. Вот что сделал бы пример из предыдущего раздела с кодировкой компонента:

```
let uri = "https://www.example.com/submit?name=maaike van putten";
let encoded_uri = encodeURIComponent(uri);
console.log("Encoded:", encoded_uri);
let decoded_uri = decodeURIComponent(encoded_uri);
console.log("Decoded:", decoded_uri);
```

Вывод будет следующим:

```
Encoded: https%3A%2F%2Fwww.example.com%2Fsubmit%3Fname%3Dmaaike%20 van%20putten
Decoded: https://www.example.com/submit?name=maaike van putten
```

Очевидно, вы бы не хотели видеть подобное в качестве URI, но методы `component` правда полезны, например, для кодирования переменной URL. Если переменная URL будет содержать специальный незакодированный символ, например `=` или `&`, это изменит значение и нарушит URI.

Кодирование с помощью escape() и unescape()

Это глобальные методы, доступные для выполнения чего-то подобного кодированию (`escape`) и декодированию (`unescape`). Настоятельно не рекомендуем их использовать, возможно, они вовсе исчезнут из будущих версий JavaScript или перестанут поддерживаться браузерами — по веским причинам.

Практическое занятие 8.1

Выведите в консоль результат работы `decodeURIComponent()` для строки `How's%20it%20going%3F`. Закодируйте строку `How's it going?`. Создайте веб-URL и закодируйте URI.

1. Добавьте строки в качестве переменных в код JavaScript.
2. Используйте `encodeURIComponent()` и `decodeURIComponent()` и выведите результаты их работы на экран.
3. Создайте веб-URI с параметрами `http://www.basescripts.com?=Hello World";`.
4. Закодируйте и выведите веб-URI на экран.

Парсинг чисел

Есть несколько способов сравнить строки с цифрами. Во многих ситуациях вам придется перевести строку в число, например, при чтении полей ввода с HTML-страницы. Со строками проводить вычисления нельзя, но с числами можно. В зависимости от того, что именно вам нужно сделать, вам понадобится один из этих методов.

Создание целых чисел с помощью parseInt()

Переменная типа `string` может быть переведена в `integer` с помощью метода `parseInt()`. Данный метод является частью класса `Number`, но, поскольку он глобальный, он может быть вызван без добавления слова `Number` в начале. Посмотрим на него в действии:

```
let str_int = "6";
let int_int = parseInt(str_int);
console.log("Type of ", int_int, "is", typeof int_int);
```

Мы начинаем со строки, содержащей `6`. Затем преобразуем эту строку в целое число, используя метод `parseInt`. При логировании результата на экране появится:

```
Type of 6 is number
```

Как видите, тип данных изменился со `string` на `number`. На этом этапе вы можете задаться вопросом: а что, если `parseInt()` попытается сравнить другие типы чисел,

например строковые версии чисел с плавающей точкой или двоичные числа. Что произойдет, если мы сделаем так?

```
let str_float = "7.6";
let int_float = parseInt(str_float);
console.log("Type of", int_float, "is", typeof int_float);

let str_binary = "0b101";
let int_binary = parseInt(str_binary);
console.log("Type of", int_binary, "is", typeof int_binary);
```

Результат будет следующим:

```
Type of 7 is number
Type of 0 is number
```

Прослеживаете логику? Прежде всего, JavaScript не любит сбоить и выдавать ошибки, поэтому пытается заставить работать код в меру своих возможностей. Метод `parseInt()` при столкновении с нечисловым символом просто останавливает анализ синтаксиса. Это заданное поведение, и вам нужно иметь его в виду. В первом случае `parseInt()` прекратил работу, как только дошел до точки, поэтому результат равен `7`. А в случае с двоичным числом метод остановил анализ, как только достиг `b`, и результат принял значение `0`. Теперь вы, вероятно, сможете понять, что делает следующий код:

```
let str_nan = "hello!";
let int_nan = parseInt(str_nan);
console.log("Type of", int_nan, "is", typeof int_nan);
```

Поскольку первый символ не является числовым, JavaScript преобразует эту строку в `NaN`. Вот результат, который вы увидите на экране:

```
Type of NaN is number
```

Так что `parseInt()` может быть немного причудливым, но все равно он очень полезен. В реальном мире этот метод часто используется, чтобы объединить пользовательский ввод с веб-страниц и вычисления.

Создание чисел с плавающей точкой с помощью parseFloat()

Для сравнения и перевода строковых значений в числа с плавающей точкой мы можем использовать `parseFloat()`. Он работает по тем же принципам, только еще понимает десятичные числа и не прекращает анализ синтаксиса, когда находит первую точку:

```
let str_float = "7.6";
let float_float = parseFloat(str_float);
console.log("Type of", float_float, "is", typeof float_float);
```

Результат будет следующим:

```
Type of 7.6 is number
```

При использовании `parseInt()` в консоли отображалось значение `7`, потому что этот метод останавливает работу, как только находит нечисловой символ. `parseFloat()`, в свою очередь, может работать с одной точкой в числе, а сами числа после этого интерпретирует как десятичные дроби. Можете ли вы догадаться, что происходит, когда он находит вторую точку?

```
let str_version_nr = "2.3.4";
let float_version_nr = parseFloat(str_version_nr);
console.log("Type of", float_version_nr, "is", typeof float_version_nr);
```

Результат будет следующим:

```
Type of 2.3 is number
```

Стратегия действия схожа с функцией `parseInt()`. Как только метод обнаруживает символ, который не может интерпретировать (в данном случае вторую точку), он прекращает анализ синтаксиса и просто возвращает результат до текущей позиции. Здесь следует отметить еще одну вещь. Для целых чисел метод не будет добавлять `.0`, поэтому `6` не станет `6.0`. Следующий пример:

```
let str_int = "6";
let float_int = parseFloat(str_int);
console.log("Type of", float_int, "is", typeof float_int);
```

выдаст результат:

```
Type of 6 is number
```

И последнее: поведение двоичных чисел и строк одинаково. Метод прекратит анализ, как только столкнется с символом, который не сможет интерпретировать:

```
let str_binary = "0b101";
let float_binary = parseFloat(str_binary);
console.log("Type of", float_binary, "is", typeof float_binary);

let str_nan = "hello!";
let float_nan = parseFloat(str_nan);
console.log("Type of", float_nan, "is", typeof float_nan);
```

Результат будет следующим:

```
Type of 0 is number
Type of NaN is number
```

`parseFloat()` стоит применять везде, где требуются десятичные числа. Однако он не будет взаимодействовать с двоичными, шестнадцатеричными и восьмеричными значениями, поэтому для работы с этими значениями или целыми числами вам придется использовать `parseInt()`.

Исполнение кода JavaScript с помощью eval()

Этот глобальный метод вызывает операторы JavaScript в виде аргумента. Иными словами, он просто выполняет любой код JavaScript, как если бы этот код был написан здесь целиком вместо `eval()`. Для работы с внедренным кодом JavaScript данное действие может быть удобно, но оно также сопряжено с большими рисками. Мы разберемся с этими рисками позже. Давайте сначала рассмотрим пример:

```
<html>
  <body>
    <input onchange="go(this)"></input>
    <script>
      function go(e) {
        eval(e.value);
      }
    </script>
  </body>
</html>
```

Это простая HTML-страница с полем ввода на ней.

Вы изучите HTML более подробно в главе 9.

Все, что вы напишете в поле ввода, будет выполнено. Если бы там оказалось:

```
document.body.style.backgroundColor = "pink";
```

фон сайта изменился бы на розовый. Выглядит забавно, правда? Однако мы не устанем повторять, что вы должны использовать `eval()` очень осторожно. Не зря многие разработчики считают, что с таким же успехом данный метод можно было бы назвать *evil* — «зло». Можете объяснить, почему так происходит?

Ответ — безопасность! Да, вы собираетесь выполнить внешний код — и это худшее, что можно сделать в большинстве ситуаций, ведь такой код может быть вредоносным. Известный и уважаемый фонд *OWASP (Open Web Application Security Project)* каждые три года обновляет список угроз безопасности. Внедрение кода входит в топ-10 угроз безопасности с их первого ранкинга и все еще находится там. Запуск кода на стороне сервера может привести к сбою сервера и выходу вашего веб-сайта из строя или даже более страшным вещам. Почти всегда можно найти решение лучше, чем `eval()`. Помимо рисков для безопасности, использование данного ме-

тода ужасно с точки зрения производительности. Именно по этим причинам вы, возможно, захотите избежать его применения.

Последнее замечание по этому поводу. Вы можете использовать `eval()` в особых случаях, если четко осознаете, что делаете. Несмотря ни на что, это «зло» обладает большой силой. В некоторых случаях его функционал может быть крайне полезен, например, когда вы создаете механизмы шаблонов, свой собственный интерпретатор и другие основные инструменты JavaScript. Поэтому помните про риски и тщательно контролируйте доступ к этому методу. И последний бонусный совет: когда вы почувствуете, что вам действительно нужно использовать `eval()`, покопайтесь сначала в интернете. Скорее всего, вы найдете альтернативу получше.

Методы работы с массивами

Вы уже знакомы с массивами — они могут содержать множество элементов. Также вы изучили довольно много встроенных методов для массивов, таких как `shift()` и `push()`. Рассмотрим еще несколько.

Выполнение определенного действия для каждого элемента

Есть причина, по которой мы начинаем с этого метода. Возможно, вы сейчас подумали о циклах, но, чтобы вызывать функцию для каждого элемента массива, можно использовать встроенный метод. Это метод `forEach()`. Мы упоминали его в главе 6 и теперь готовы рассмотреть более подробно. В качестве входных данных метод принимает функцию, которая должна быть выполнена для каждого элемента. Ниже представлен пример:

```
let arr = ["grapefruit", 4, "hello", 5.6, true];

function printStuff(element, index) {
  console.log("Printing stuff:", element, "on array position:", index);
}

arr.forEach(printStuff);
```

Данный фрагмент кода выведет на экран:

```
Printing stuff: grapefruit on array position: 0
Printing stuff: 4 on array position: 1
Printing stuff: hello on array position: 2
Printing stuff: 5.6 on array position: 3
Printing stuff: true on array position: 4
```

Как видите, метод вызвал функцию `printStuff()` для каждого элемента массива. Мы также можем использовать значение индекса — это наш второй параметр. Нам не придется контролировать выполнение цикла, и мы не застрянем в определенной точке. Нужно будет только указать, какую функцию необходимо выполнить для каждого элемента, и функция примет значение элементов. Такой способ применяется часто, особенно в более функциональном стиле программирования, когда множество методов связываются в цепь (например, для обработки данных).

Фильтрация массива

К массиву можно применять встроенный метод `filter()`, чтобы изменять в нем значения. Метод фильтрации принимает функцию в качестве аргумента, и эта функция возвращает логическое значение. Если логическое значение `true`, элемент попадает в отфильтрованный массив. Если `false` — пропускается. Ниже вы можете увидеть, как это работает:

```
let arr = ["squirrel", 5, "Tjed", new Date(), true];

function checkString(element, index) {
  return typeof element === "string";
}

let filterArr = arr.filter(checkString);
console.log(filterArr);
```

На экран будет выведено:

```
[ 'squirrel', 'Tjed' ]
```

Важно понимать, что исходный массив не меняется: метод `filter()` возвращает новый массив, содержащий элементы, прошедшие условия фильтра. Эти данные фиксируются в переменной `filterArr`.

Проверка условия для всех элементов

Чтобы узнать, есть ли какое-то общее логически верное условие для всех элементов, можно использовать метод `every()`. Если условие есть, метод `every()` вернет значение `true`, в ином случае — `false`. Мы используем функцию `checkString()` и массив из предыдущего примера:

```
console.log(arr.every(checkString));
```

Эта операция логирует значение `false`, поскольку не все элементы массива имеют тип `string`.

Замена части массива другой частью массива

Для замены части массива другой частью массива можно применять метод `copyWithin()`. Давайте сначала зададим три аргумента. Первый — это целевая позиция, в которую копируются значения. Второй — указание позиции, с которой надо начинать копирование. И третий — конец копируемой последовательности (данный индекс не включен в область копирования). Здесь мы собираемся переписать значение позиции 0 любым значением, находящимся в позиции 3:

```
arr = ["grapefruit", 4, "hello", 5.6, true];
arr.copyWithin(0, 3, 4);
```

Массив `arr` станет таким:

```
[ 5.6, 4, 'hello', 5.6, true ]
```

Если мы укажем диапазон копирования длиной `2`, то первые два элемента после начальной позиции будут переписаны:

```
arr = ["grapefruit", 4, "hello", 5.6, true];
arr.copyWithin(0, 3, 5);
```

и массив `arr` примет вид:

```
[ 5.6, true, 'hello', 5.6, true ]
```

Мы можем не указывать конечное значение — тогда диапазон копирования продлится до конца строки:

```
let arr = ["grapefruit", 4, "hello", 5.6, true, false];
arr.copyWithin(0, 3);
console.log(arr);
```

Результат будет следующим:

```
[ 5.6, true, false, 5.6, true, false ]
```

Важно помнить, что эта функция меняет *содержание* исходного массива, но не его длину.

Сопоставление значений массива

Иногда требуется изменить все значения в массиве. Это можно осуществить с помощью метода `map()`. Данный метод вернет новый массив с новыми значениями. Чтобы создать эти значения, необходимы соответствующие условия. Их можно

задать с помощью стрелочной функции. Она будет применена к каждому элементу массива, например:

```
let arr = [1, 2, 3, 4];
let mapped_arr = arr.map(x => x + 1);
console.log(mapped_arr);
```

Вот как выглядит массив после сопоставления:

```
[ 2, 3, 4, 5 ]
```

Метод `map()` использовал стрелочную функцию для создания нового массива, в котором все исходные значения были увеличены на 1.

Поиск последнего вхождения элемента в массиве

Вы уже знаете, что последнее встречающееся значение можно найти с помощью `indexOf()`. А последнее вхождение переменной в массиве можно обнаружить благодаря методу `lastIndexOf()`, как мы уже делали со `string`.

Данный метод вернет значение индекса элемента с требуемым значением, если найдет его:

```
let bb = ["so", "bye", "bye", "love"];
console.log(bb.lastIndexOf("bye"));
```

На экране появится `2` — именно значение с индексом 2 содержит последнюю встречающуюся переменную `bye`. Как думаете, какой результат мы получим, если запросим последний индекс несуществующего элемента?

```
let bb = ["so", "bye", "bye", "love"];
console.log(bb.lastIndexOf("hi"));
```

Правильно (надеемся)! Это `-1`.

Практическое занятие 8.2

Удалите дубликаты значений из массива с помощью `filter()` и `indexOf()`. Начальный массив:

```
["Laurence", "Mike", "Larry", "Kim", "Joanne", "Laurence", "Mike",
"Laurence", "Mike", "Laurence", "Mike"]
```

Используя метод `filter()`, создайте новый массив и включите в него элементы, которые проходят реализованное функцией условие проверки.

Конечный результат должен быть таким:

```
[ 'Laurence', 'Mike', 'Larry', 'Kim', 'Joanne' ]
```

Выполните следующие шаги.

1. Создайте массив имен. Убедитесь, что ввели повторяющиеся значения, — это необходимо для выполнения условий упражнения.
2. Используя метод `filter()`, назначьте результирующие значения каждого элемента из массива в качестве аргументов в анонимной функции. Используя значения, индекс и аргументы массива, верните отфильтрованный результат. Вы можете временно установить возвращаемое значение `true` — при этом будет создан новый массив со всеми результатами в исходном массиве.
3. Добавьте внутри функции вызов `console.log`, который выведет индекс текущего элемента в массиве. Также добавьте значение, с помощью которого вы увидите значение элемента, имеющего текущий номер индекса, и индекс первого подходящего результата.
4. С помощью `indexOf()` текущее значение возвращает индекс элемента и проверяет, соответствует ли условие исходному индексу. Условие будет истинным только для первого совпадения, поэтому все последующие дубликаты не будут соответствовать условию и не добавятся в новый массив — значение `false` не позволит сделать это. Таким образом, все дубликаты будут ложными, раз `indexof()` получает только первое совпадение в массиве.
5. Выведите новый массив уникальных значений на экран.

Практическое занятие 8.3

Обновите содержимое массива с помощью метода `map()`. Выполните следующие шаги.

1. Создайте массив чисел.
2. Верните обновленный массив, в котором все значения будут умножены на 2. Используйте для этого метод `map()` и анонимную функцию. Выведите результаты на экран.
3. В качестве альтернативного метода примените в одной строке кода стрелочную функцию для умножения каждого элемента массива на 2 и метод `map()`.
4. Выведите результаты на экран.

Строчные методы

Вы уже работали со строками и правдоподобно сталкивались с некоторыми методами взаимодействия с переменными типа `string`. Есть несколько вопросов, которые мы еще не рассматривали детально, — давайте обсудим их в этом разделе.

Объединение строк

Чтобы объединить строки, можно использовать метод `concat()`. Он не меняет исходные строки, но возвращает результат объединения в виде строки. Необходимо сохранить результат в новой переменной, иначе он будет утерян:

```
let s1 = "Hello ";
let s2 = "JavaScript";
let result = s1.concat(s2);
console.log(result);
```

В консоли появится следующая запись:

```
Hello JavaScript
```

Преобразование строки в массив

Строку можно преобразовать в массив с помощью метода `split()`. И снова нам необходимо зафиксировать результат, поскольку данный метод не изменяет исходные строки. Возьмем итог выполнения предыдущего примера: `Hello JavaScript`. Сообщите методу `split`, на каком значении он должен разделять данные. Каждый раз, когда он столкнется с этим значением, он будет создавать новый элемент массива:

```
let result = "Hello JavaScript";
let arr_result = result.split(" ");
console.log(arr_result);
```

Результат будет следующим:

```
[ 'Hello', 'JavaScript' ]
```

Как видите, метод создает массив всех элементов, отделенных пробелом. Элементы можно разделять любым знаком, например запятой:

```
let favoriteFruits = "strawberry,watermelon,grapefruit";
let arr_fruits = favoriteFruits.split(",");
console.log(arr_fruits);
```

Результат будет следующим:

```
[ 'strawberry', 'watermelon', 'grapefruit' ]
```

Теперь программа создала массив из трех элементов. Вы можете разделить что угодно, и строка, с которой вы работаете, не будет учитываться в результате.

Преобразование массива в строку

Массив можно преобразовать в строку, используя метод `join()`. Перед вами простой пример:

```
let letters = ["a", "b", "c"];
let x = letters.join();
console.log(x);
```

Результат будет следующим:

```
a,b,c
```

Тип переменной `x` — `string`. Если вы хотите использовать что-то вместо запятой, укажите это следующим образом:

```
let letters = ["a", "b", "c"];
let x = letters.join('-');
console.log(x);
```

В таком случае вместо запятой появится другой символ: `-`. И вот как будет выглядеть запись в консоли:

```
a-b-c
```

Данный метод можно отлично совмещать с методом `split()`, рассмотренным в предыдущем подразделе. Напомним: он делает обратное, преобразуя строку в массив.

Работа со свойствами index и position

Возможность узнать значение индекса определенной подстроки в вашей строке очень полезна. Например, когда вы ищете определенное слово в файле с записанным пользовательским вводом, чтобы создать подстроку, начинающуюся с этого индекса. Вот как находится индекс строки. Метод `indexOf()` вернет отдельное значение — индекс первого символа подстроки:

```
let poem = "Roses are red, violets are blue, if I can do JS, then you can too!";
let index_re = poem.indexOf("re");
console.log(index_re);
```

Результат вывода на экран — 7. Он обусловлен тем, что первое значение re встречается в are, и индекс первого символа в этом re — 7. Когда индекс не найден, будет возвращено значение -1:

```
let indexNotFound = poem.indexOf("python");
console.log(indexNotFound);
```

Значение -1 на экране означает, что подстрока, которую мы ищем, не встречается в целевой строке. Часто вам понадобится написать if, чтобы увидеть, действительно ли значение равно -1, прежде чем разбираться с результатом. Например:

```
if(poem.indexOf("python") != -1) {
  // некий код
}
```

Альтернативным способом поиска определенной подстроки в строке является использование метода search():

```
let searchStr = "When I see my fellow, I say hello";
let pos = searchStr.search("lo");
console.log(pos);
```

Результат будет 17 — это значение индекса lo в слове fellow. Точно так же, как indexOf(), если значение не будет найдено, метод search() выдаст -1. Как в следующем примере:

```
let notFound = searchStr.search("JavaScript");
console.log(notFound);
```

search() в качестве входных данных будет принимать формат регулярного выражения, тогда как indexOf() просто принимает строку. indexOf() работает быстрее, чем search(), но если вы просто хотите найти строку, используйте indexOf(). Если же вам требуется шаблон строки, стоит применить метод search().

Регулярное выражение — это специальный синтаксис для определения строковых шаблонов, с помощью которого вы можете заменить все встречающиеся при поиске значения. Мы будем изучать это в главе 12.

Идем дальше. Как помним, метод indexOf() возвращает индекс первого встречающегося значения. Существует также аналогичный метод lastIndexOf() — он возвращает индекс той позиции, где аргумент строки встречается последний раз. Не найдя совпадений, метод возвращает значение -1. Например:

```
let lastIndex_re = poem.lastIndexOf("re");
console.log(lastIndex_re);
```

Данная операция возвращает значение 24. Это последний раз, когда re встречается в нашем стихотворении — во втором слове are.

Иногда понадобится сделать обратное: не искать, под каким индексом встречается строка, а узнать, какой символ находится под определенным индексом. Здесь пригодится метод charAt(index), который в качестве аргумента берет указанную позицию индекса:

```
let pos1 = poem.charAt(10);
console.log(pos1);
```

В консоли выводится значение r, потому что символ с индексом 10 — это r в слове red. Если вы запрашиваете позицию индекса, который находится вне диапазона строки, он вернет пустую строку:

```
let pos2 = poem.charAt(1000);
console.log(pos2);
```

На экране появится пустая строка. Если вы запросите тип переменной pos2, получите значение string.

Создание подстрок

Создавать подстроки можно с помощью метода slice(start, end). Он не изменяет исходную строку, но возвращает новую строку, состоящую из значений выбранной подстроки. Метод принимает два параметра: первый — индекс, с которого начнется работа метода, и второй — конечный. Если вы опустите второй индекс, метод будет выполняться с начального индекса до конца строки. Конечный индекс не включается в подстроку. Вот пример:

```
let str = "Create a substring";
let substr1 = str.slice(5);
let substr2 = str.slice(0,3);
console.log("1:", substr1);
console.log("2:", substr2);
```

Результат будет следующим:

```
1: e a substring
2: Cre
```

Первая подстрока содержит только один аргумент, поэтому начинается с индекса 5 (в котором записано значение e) и принимает все значения строки до конца. Вторая содержит два аргумента — 0 и 3 (C с индексом 0 и a с индексом 3). Поскольку последний индекс не включен в подстроку, метод вернет только Cre.

Замена фрагментов строки

Метод `replace(old, new)` применяется для замены фрагмента строки. Он получает два аргумента. Первый — переменная типа `string`, которую необходимо найти в строке. Второй — новое значение для замены. Перед вами пример:

```
let hi = "Hi buddy";
let new_hi = hi.replace("buddy", "Pascal");
console.log(new_hi);
```

На экран будет выведено `Hi Pascal`. Если вы не зафиксируете результат, он исчезнет, потому что исходная строка не меняется. В случае, когда переменная, на которую вы ориентируетесь, в исходной строке отсутствует, замена не выполняется и возвращается исходная строка:

```
let new_hi2 = hi.replace("not there", "never there");
console.log(new_hi2);
```

На экране появится `Hi buddy`.

Последнее замечание: по умолчанию изменяется только первое встречающееся совпадение. Поэтому следующий пример заменит только первую запись `hello` в строке:

```
let s3 = "hello hello";
let new_s3 = s3.replace("hello", "oh");
console.log(new_s3);
```

В консоли отобразится `oh hello`. Если требуется заменить все встречающиеся значения, применяется метод `replaceAll()`:

```
let s3 = "hello hello";
let new_s3 = s3.replaceAll("hello", "oh");
console.log(new_s3);
```

На экране появится надпись `oh oh`.

Верхний и нижний регистры

Изменить буквы в строке можно с помощью `toUpperCase()` и `toLowerCase()` — встроенных методов для работы с переменными типа `string`. И снова нам необходимо зафиксировать результат (данный метод не изменяет исходную строку).

```
let low_bye = "bye!";
let up_bye = low_bye.toUpperCase();
console.log(up_bye);
```

Результат будет таким:

```
BYE!
```

`toUpperCase()` переводит все буквы в верхний регистр. Обратное можно сделать с помощью `toLowerCase()`:

```
let caps = "HI HOW ARE YOU?";
let fixed_caps = caps.toLowerCase();
console.log(fixed_caps);
```

Итог будет следующим:

```
hi how are you?
```

Немного усложним задачу и скажем, чтобы первое слово предложения было написано с большой буквы. Это можно сделать, объединив некоторые уже знакомые вам методы:

```
let caps = "HI HOW ARE YOU?";
let fixed_caps = caps.toLowerCase();
let first_capital = fixed_caps.charAt(0).toUpperCase().concat(fixed_
caps.slice(1));
console.log(first_capital);
```

Здесь мы комбинируем методы. Сначала мы получаем первую букву, используя `fixed_caps` и `charAt(0)`, после чего меняем ее регистр на верхний, вызвав `toUpperCase()`. Затем нам нужна остальная часть строки, получить ее мы можем путем объединения `slice(1)`.

Начало и конец строки

Иногда требуется узнать, с чего строка начинается и чем заканчивается. Как вы уже догадались, для этого тоже существуют встроенные методы. Можем представить, как тяжко вам дается эта глава, поэтому вот вам небольшое поощрение — и в то же время пример:

```
let encouragement = "You are doing great, keep up the good work!";
let bool_start = encouragement.startsWith("You");
console.log(bool_start);
```

На экране появится значение `true` — предложение действительно начинается со слова `You`. Обратите внимание на то, что метод чувствителен к регистру, поэтому следующий код выдаст результат `false`:

```
let bool_start2 = encouragement.startsWith("you");
console.log(bool_start2);
```

Если вас не волнует регистр букв, можете использовать ранее обсуждавшийся метод `toLowerCase()`:

```
let bool_start3 = encouragement.toLowerCase().startsWith("you");
console.log(bool_start3);
```

В данном случае мы сначала преобразуем всю строку в нижний регистр, и теперь уверены, что работаем только со строчными символами. Однако помните, что такая обработка огромных по объему строк влияет на производительность.

Опять же более удобной альтернативой является использование регулярных выражений. Уже взволнованы приближением главы 12?

В завершение данного раздела мы можем сделать то же самое для проверки окончания строки. Рассмотрим это в действии:

```
let bool_end = encouragement.endsWith("Something else");
console.log(bool_end);
```

Поскольку строка не заканчивается на `Something else`, будет возвращено значение `false`.

Практическое занятие 8.4

Используя методы управления строками, создайте функцию, которая будет возвращать строку, где каждое слово будет написано с большой буквы. Иными словами, вы должны преобразовать предложение `thIs will be capiTalized for each word` в `This Will Be Capitalized For Each Word`.

1. Создайте строку из нескольких слов, содержащих в себе буквы разного регистра.
2. Создайте функцию, получающую строку в качестве аргумента. Эта строка будет значением, которым мы будем управлять.
3. Преобразуйте в функции написание всех букв в нижний регистр.
4. Создайте пустой массив — в нем будут храниться значения слов, написанных с заглавной буквы.
5. Преобразуйте целую фразу в отдельные слова в массиве, используя метод `split()`.
6. Переберите циклом все слова нового массива, чтобы обращаться независимо к любому из них. Можно для этого применить `forEach()`.

7. Отделите первую букву каждого слова с помощью `slice()` и преобразуйте ее в верхний регистр. Снова примените `slice()`, чтобы получить оставшееся слово без первой буквы. Затем соедините части слова, чтобы образовать слово, которое теперь пишется с заглавной буквы.
8. Добавьте новое слово с заглавной буквы в созданный вами пустой массив. К концу цикла вы должны получить массив всех слов как отдельных элементов массива.
9. Возьмите массив обновленных слов и, используя метод `join()`, преобразуйте их обратно в строку с пробелами между каждым словом.
10. Верните значение обновленной строки, которое затем можно вывести на экран.

Практическое занятие 8.5

Используя строчный метод `replace()`, замените гласные буквы в строке цифрами. Можете начать со строки:

```
I love JavaScript
```

и преобразовать ее в нечто подобное:

```
2 l3v1 j0v0scr2pt
```

Выполните следующие шаги.

1. Создайте строку и конвертируйте ее в нижний регистр.
2. Создайте массив гласных: a, e, i, o, u.
3. Пройдитесь по всем буквам, которые уже есть в массиве, и выводите текущую букву на экран, чтобы видеть, какие буквы в итоге будут преобразованы.
4. Используйте `replaceAll()` в цикле и обновите каждую подстроку гласной значением индекса буквы из массива гласных.

Метод replace() заменит только первое найденное значение. Метод replaceAll() обновит все совпадения.

5. Как только цикл завершится, выведите новую строку на экран.

Числовые методы

Перейдем к встроенным методам работы с объектами `Number`, с некоторыми из них вы уже встречались на страницах книги. Эти методы настолько популярны, что некоторые из них были превращены в глобальные.

Проверка на принадлежность числовому типу данных

Эта операция может быть выполнена с помощью `isNaN()`, мы уже рассматривали данный способ, когда говорили о глобальных методах. Метод можно применять без ключевого слова `Number`. Метод `isNaN()` возвращает значение `true`, если значение *не* является числовым. Но часто вам понадобится обратный результат. Получить его можно с помощью приставки `!`:

```
let x = 34;
console.log(isNaN(x));
console.log(!isNaN(x));
let str = "hi";
console.log(isNaN(str));
```

На экран будет выведено:

```
false
true
true
```

Так как `x` является числом, `isNaN` примет значение `false`. Но далее этот результат изменится на противоположный — `true`, поскольку в коде мы использовали приставку `!`. Строка `hi` не является числовым значением, поэтому в консоли появится `true`. А что будет здесь?

```
let str1 = "5";
console.log(isNaN(str1));
```

Здесь происходит что-то странное. Хотя `5` заключено в кавычки, JavaScript все еще видит, что это число, и выдает значение `false`. Уверены, вы бы хотели, чтобы ваш партнер, семья и коллеги были такими же понимающими и прощающими, как JavaScript.

Проверка на конечность значения

Возможно, вы сможете угадать название метода `Number`, который проверяет, является ли значение конечным. Он очень популярен и также был превращен в глобальный, его название `isFinite()`. Метод возвращает значение `false` для `NaN`, `Infinity` и `undefined`, и `true` — для всех остальных типов.

```
let x = 3;
let str = "finite";
console.log(isFinite(x));
console.log(isFinite(str));
console.log(isFinite(Infinity));
console.log(isFinite(10 / 0));
```

Результат будет следующим:

```
true
false
false
false
```

Только `x` является конечным числом в этом списке. Остальные — нет. Строка не является числом и, следовательно, не является конечной. `Infinity` также не конечное значение, а `10`, деленное на `0`, возвращает значение `Infinity` (не ошибку).

Проверка целых чисел

Да, вы наверняка догадались, что это делается с помощью `isInteger()`. В отличие от `isNaN()` и `isFinite()`, `isInteger()` не является глобальным методом, поэтому для его применения ссылаться на объект `Number` обязательно. Он действительно делает то, что вы думаете: возвращает `true`, если значение является целым числом, и `false`, если это не так:

```
let x = 3;
let str = "integer";
console.log(Number.isInteger(x));
console.log(Number.isInteger(str));
console.log(Number.isInteger(Infinity));
console.log(Number.isInteger(2.4));
```

Поскольку единственное целое число в списке — это `x`, результат будет следующим:

```
true
false
false
false
```

Указание количества знаков после запятой

Сообщить JavaScript о том, сколько знаков после запятой необходимо использовать, можно с помощью метода `toFixed()`. Этой возможностью он отличается от методов округления `Math`. Метод не изменяет исходное значение, поэтому нам придется сохранить результат в переменной:

```
let x = 1.23456;
let newX = x.toFixed(2);
```

Данный фрагмент кода оставит только два знака после запятой, поэтому значение `newX` будет равно `1.23`. Метод округляет числа по обычным математическим принципам. Чтобы продемонстрировать это, добавим еще один знак после точки:

```
let x = 1.23456;
let newX = x.toFixed(3);
console.log(x, newX);
```

Результат будет таким: `1.23456 1.235`.

Указание необходимой точности

Существует также метод, позволяющий вам указать нужное количество отображаемых знаков для чисел с плавающей точкой. Этим опять же он отличается от методов округления в классе `Math`. Здесь JavaScript просматривает все число, учитывая также цифры перед точкой:

```
let x = 1.23456;
let newX = x.toPrecision(2);
```

Значение `newX` будет равным `1.2`. Отображаемое значение также может быть округлено:

```
let x = 1.23456;
let newX = x.toPrecision(4);
console.log(newX);
```

Результат будет равен `1.235`.

Продолжим — теперь мы поговорим о некоторых полезных математических методах.

Математические методы

Объект `Math` включает в себя множество методов, которые мы можем использовать для выполнения вычислений и операций с числами. Далее мы рассмотрим наиболее важные из них. Вы можете увидеть все доступные методы, если работаете в редакторе, который во время набора кода отображает подсказки и дает возможность выбора.

Нахождение наибольшего и наименьшего числа

Для поиска наибольшего числа среди аргументов существует встроенный метод `max()`. Посмотрим на него в действии:

```
let highest = Math.max(2, 56, 12, 1, 233, 4);
console.log(highest);
```

В консоли появится значение 233 — именно это число наибольшее среди остальных. Аналогичным образом можно найти наименьшее число:

```
let lowest = Math.min(2, 56, 12, 1, 233, 4);
console.log(lowest);
```

В итоге получим 1 — это наименьшее число. Если попробовать применить данные методы к нечисловым аргументам, результатом будет NaN:

```
let highestOfWords = Math.max("hi", 3, "bye");
console.log(highestOfWords);
```

Метод не дает 3 в качестве выходного значения. Он анализирует все значения и, поскольку не может определить, является ли hi больше или меньше 3, возвращает NaN.

Квадратный корень и возведение в степень

Для вычисления квадратного корня из определенного числа используется метод sqrt():

```
let result = Math.sqrt(64);
console.log(result);
```

Результатом будет 8, поскольку именно этому значению равняется квадратный корень из 64. Данный метод работает в точности по таким же законам, как и школьная математика. Чтобы возвести число в определенную степень (основание$^{\text{степень}}$, например, 2^3), можно использовать функцию pow(основание, степень). Например, так:

```
let result2 = Math.pow(5, 3);
console.log(result2);
```

Здесь мы возводим 5 в степень 3 (5^3) — следовательно, результат будет 125 (произведение $5 \times 5 \times 5$).

Преобразование десятичных дробей в целые числа

Есть несколько способов преобразовать десятичные дроби в целые числа. Иногда необходимо округлить число. Это можно осуществить с помощью метода round():

```
let x = 6.78;
let y = 5.34;

console.log("X:", x, "becomes", Math.round(x));
console.log("Y:", y, "becomes", Math.round(y));
```

Результат будет следующим:

```
X: 6.78 becomes 7
Y: 5.34 becomes 5
```

Как видите, здесь осуществляется обычное округление. Но встречаются случаи, когда округлить в большую сторону необходимо любое значение. Например, если мы считаем нужное нам количество досок и приходим к значению `1.1`, то понимаем, что одной доски будет явно недостаточно для завершения работы. Нам понадобится две. В таком случае можно использовать метод `ceil()`:

```
console.log("X:", x, "becomes", Math.ceil(x));
console.log("Y:", y, "becomes", Math.ceil(y));
```

В консоли появится следующая запись:

```
X: 6.78 becomes 7
Y: 5.34 becomes 6
```

Метод `ceil()` всегда округляет до первого найденного целого числа. Вспомните: мы использовали его раньше, когда генерировали случайные числа! Но будьте внимательны к отрицательным числам, потому что число `-5` больше, чем `-6`. Ниже вы можете увидеть, как это работает:

```
let negativeX = -x;
let negativeY = -y;
console.log("negativeX:", negativeX, "becomes", Math.ceil(negativeX));
console.log("negativeY:", negativeY, "becomes", Math.ceil(negativeY));
```

На экране отобразится следующий результат:

```
negativeX: -6.78 becomes -6
negativeY: -5.34 becomes -5
```

Метод `floor()` — полная противоположность методу `ceil()`. Он делает округление в обратную сторону:

```
console.log("X:", x, "becomes", Math.floor(x));
console.log("Y:", y, "becomes", Math.floor(y));
```

Результат будет следующим:

```
X: 6.78 becomes 6
Y: 5.34 becomes 5
```

Будьте внимательны к отрицательным числам — результат работы может показаться нелогичным:

```
console.log("negativeX:", negativeX, "becomes", Math.floor(negativeX));
console.log("negativeY:", negativeY, "becomes", Math.floor(negativeY));
```

В консоли увидим такую запись:

```
negativeX: -6.78 becomes -7
negativeY: -5.34 becomes -6
```

И еще один, последний, метод `trunc()`. Для положительных чисел он дает точно такой же результат, что и `floor()`, но приходит к нему другим путем. По сути, это не округление в меньшую сторону, а просто возврат целой части:

```
console.log("X:", x, "becomes", Math.trunc(x));
console.log("Y:", y, "becomes", Math.trunc(y));
```

Результат будет следующим:

```
X: 6.78 becomes 6
Y: 5.34 becomes 5
```

Мы поймем разницу, если применим `trunc()` к отрицательным числам:

```
negativeX: -6.78 becomes -6
negativeY: -5.34 becomes –5
```

Поэтому всякий раз, когда нужно округлить число в меньшую сторону, вам придется использовать `floor()`; а если понадобится целая часть числа, берите `trunc()`.

Показатель степени и логарифм

Показатель степени — это число, до которого повышается основание. В математике мы часто используем *e* (число Эйлера). В JavaScript это делает метод `exp()`. Он возвращает число, до которого *e* должно быть увеличено, чтобы получить входные данные. Мы можем использовать `exp()` — встроенный метод `Math` для вычисления экспоненты, и метод `log()`, чтобы вычислить натуральный логарифм. Рассмотрим следующий пример:

```
let x = 2;
let exp = Math.exp(x);
console.log("Exp:", exp);
let log = Math.log(exp);
console.log("Log:", log);
```

Результат будет следующим:

```
Exp: 7.38905609893065
Log: 2
```

Не волнуйтесь, если на данный момент не можете уловить математическую суть. Вы все поймете, когда начнете использовать это в программировании.

Практическое занятие 8.6

Поэкспериментируйте с объектом `Math`, выполнив следующие действия.

1. Выведите значение `PI` на экран с помощью `Math`.
2. Используя `Math` для числа `5.7`, получите значение `ceil()`, значение `floor()`, округленное значение. Выведите результат на экран.
3. Выведите на экран случайное значение.
4. Используйте `Math.floor()` и `Math.random()` для получения числа в диапазоне от 0 до 10.
5. Используйте `Math.floor()` и `Math.random()` для получения числа в диапазоне от 1 до 10.
6. Используйте `Math.floor()` и `Math.random()` для получения числа в диапазоне от 1 до 100.
7. Создайте функцию для генерации случайного числа, используя параметры `min` и `max`. Запустите функцию 100 раз, каждый раз возвращая на экран случайное число в диапазоне от 1 до 100.

Метод работы с датами

Для работы с датами в JavaScript существует встроенный объект `Date`. Он включает в себя множество функций.

Генерирование дат

Есть несколько способов генерировать дату. Один из них — с помощью различных конструкторов. Рассмотрим несколько примеров.

```
let currentDateTime = new Date();
console.log(currentDateTime);
```

Код выводит текущую дату в следующем виде:

```
2021-06-05T14:21:45.625Z
```

Таким образом, мы используем не встроенный метод, а конструктор. Но существует также встроенный метод `now()`, который возвращает текущие дату и время аналогичным образом:

```
let now2 = Date.now();
console.log(now2);
```

При этом будет записано текущее время в секундах, которые прошли начиная с 1 января 1970 года (это условная дата, представляющая эпоху Unix):

```
1622902938507
```

Мы можем добавить 1000 миллисекунд к времени эпохи Unix:

```
let milliDate = new Date(1000);
console.log(milliDate);
```

Результат будет таким:

```
1970-01-01T00:00:01.000Z
```

JavaScript может конвертировать в даты многие строковые форматы. Всегда обращайте внимание на формат даты в интерпретаторе JavaScript — в каком порядке представлены дни и месяцы. В зависимости от региона они могут сильно отличаться:

```
let stringDate = new Date("Sat Jun 05 2021 12:40:12 GMT+0200");
console.log(stringDate);
```

Полученный результат:

```
2021-06-05T10:40:12.000Z
```

И наконец, с помощью конструктора вы можете указать определенную дату:

```
let specificDate = new Date(2022, 1, 10, 12, 10, 15, 100);
console.log(specificDate);
```

Результат будет следующим:

```
2022-02-10T12:10:15.100Z
```

Обратите внимание на важную деталь, а именно на второй параметр, обозначающий месяц: `0` — это январь, а `11` — это декабрь.

Методы получения и установки элементов даты

Теперь, когда мы увидели, как генерировать даты, перейдем к тому, как получать определенные части дат (дни недели, месяцы и т. д.). Эта операция может быть выполнена с помощью множества методов `get`. Какой из методов применять, зависит от требуемых сведений:

```
let d = new Date();
console.log("Day of week:", d.getDay());
console.log("Day of month:", d.getDate());
console.log("Month:", d.getMonth());
console.log("Year:", d.getFullYear());
console.log("Seconds:", d.getSeconds());
```

```
console.log("Milliseconds:", d.getMilliseconds());
console.log("Time:", d.getTime());
```

Результат будет следующим:

```
Day of week: 6
Day of month: 5
Month: 5
Year: 2021
Seconds: 24
Milliseconds: 171
Time: 1622903604171
```

Значение времени столь велико, потому что это число миллисекунд, прошедших с 1 января 1970 года. Изменить дату подобным образом можно с помощью метода `set`. Здесь важно отметить, что с его помощью меняется исходный объект даты:

```
d.setFullYear(2010);
console.log(d);
```

Мы изменили год в нашем объекте `date` на `2010`. Вывод на экран:

```
2010-06-05T14:29:51.481Z
```

Мы можем также изменить месяц. Добавим приведенный ниже фрагмент в наш код — он поменяет текущий месяц на октябрь. Имейте в виду, что, пока мы это делаем, код запускается снова и снова, поэтому минуты и меньшие единицы времени, раз мы их еще не установили сами, будут в примерах отличаться:

```
d.setMonth(9);
console.log(d);
```

Результат будет таким:

```
2010-10-05T14:30:39.501Z
```

Кажется странным: чтобы изменить день недели, мы должны вызвать метод `setDate()`, а не метод `setDay()`. Все потому, что метода `setDay()` не существует, ведь день недели вычисляется из определенной даты. Мы не можем изменить то, что 5 сентября 2021 года — это воскресенье. Хотя мы можем изменить количество дней в месяце:

```
d.setDate(10);
console.log(d);
```

В консоли отобразится запись:

```
2010-10-10T14:34:25.271Z
```

Можно изменить часы:

```
d.setHours(10);
console.log(d);
```

Получим:

```
2010-10-10T10:34:54.518Z
```

Помните, как сильно JavaScript не любит ломаться? Если вызвать метод `setHours()` со значением аргумента более 24, JavaScript, использовав оператор модуля, перейдет к следующей дате (по одной на каждые 24 часа). Все, что останется от `hours % 24`, будет значением часа.

Такая же процедура справедлива для минут, секунд и миллисекунд.

Функция `setTime()` фактически переопределяет полную дату с помощью вводимых пользователем временных параметров:

```
d.setTime(1622889770682);
console.log(d);
```

Результат будет следующим:

```
2021-06-05T10:42:50.682Z
```

Парсинг дат

Провести парсинг взятых из строк периодов дат можно с помощью встроенного метода `parse()`. Он принимает множество форматов, но, опять же, будьте предельно внимательны к порядку написания дней и месяцев.

```
let d1 = Date.parse("June 5, 2021");
console.log(d1);
```

В консоли отобразится следующий результат:

```
1622851200000
```

Как видите, значение оканчивается на множество нулей, потому что в нашей строке не было указания, как отобразить время: в секундах или более точно. Вот еще один пример, где мы задаем ту же дату в совершенно ином формате:

```
let d2 = Date.parse("6/5/2021");
console.log(d2);
```

Результат будет таким же:

```
1622851200000
```

Вывод данных для парсинга соответствует стандарту данных ISO. Довольно много форматов можно парсировать в строку, но будьте при этом осторожны. От точности реализации зависит ваш результат. Убедитесь, что вы знаете формат входящей

строки, чтобы не путать месяцы и дни, и способы выполнения операции — например, когда вам нужно преобразовать данные, поступающие из вашей собственной базы данных или средства выбора даты на веб-сайте.

Преобразование даты в строку

Дату можно конвертировать обратно в строку. Например, с помощью следующих методов.

```
console.log(d.toDateString());
```

Будет выведен результат, в котором день указан в буквенном виде:

```
Sat Jun 05 2021
```

Есть другой метод, который производит конвертацию по-другому:

```
console.log(d.toLocaleDateString());
```

Результат будет таким:

```
6/5/2021
```

Практическое занятие 8.7

Выведите на экран дату с указанием полного названия месяца. Преобразуя в массивы или из них, помните, что они начинаются с нуля.

1. Задайте объект, который может быть любой датой в будущем или прошлом. Выведите дату в консоль, чтобы посмотреть, как выглядит обычный вывод для объекта `date`.
2. Задайте массив, включающий названия всех месяцев года. Храните их в порядке следования, чтобы они соответствовали выводимой дате месяца.
3. Получите значение дня из объекта `date`, используя `getDate()`.
4. Получите значение года из объекта `date`, используя `getFullYear()`.
5. Получите значение месяца из объекта `date`, используя `getMonth()`.
6. Создайте переменную для хранения даты объекта `date` и выведите месяц, используя его номер в качестве индекса для имени месяца из массива. Из-за того, что массивы начинаются с нуля, а месяц возвращает значение в диапазоне от 1 до 12, из результата необходимо вычесть единицу.
7. Выведите результаты на экран.

Проекты текущей главы

Скремблер слов

Напишите функцию, которая возвращает значение слова и перемешивает порядок букв с помощью `Math.random()`.

1. Создайте строку, которая будет хранить выбранное вами слово.
2. Напишите функцию, которая может принимать параметр в виде значения строкового слова.
3. Так же, как и массивы, строки имеют длину по умолчанию. Можно использовать это значение для установки максимального числа итераций в цикле. Для его хранения по мере уменьшения значения с каждой итерацией цикла вам понадобится отдельная переменная.
4. Задайте пустую временную строковую переменную, которую можно использовать для хранения нового значения зашифрованного слова.
5. Создайте цикл `for`, который будет повторять количество букв в параметре строки, начиная с 0 и до тех пор, пока не достигнет исходного значения длины строки.
6. С помощью `Math.floor()` и `Math.random()`, умноженных на текущую длину строки, пропишите переменную, которая случайным образом выберет одну букву, используя ее индекс.
7. Добавьте эту букву в новую строку и удалите из исходной строки.
8. Выведите на экран вновь созданную строку из случайных букв и исходную строку, используя `console.log()`. Делайте так по мере продолжения цикла.
9. Обновите исходную строку, выбрав подстроку из значения с индексом и добавив ее к оставшемуся строковому значению с этим индексом плюс один. Выведите новое исходное строковое значение с удаленными символами.
10. Проходя через цикл, вы увидите уменьшение количества букв, оставшихся для работы, новую зашифрованную версию слова, когда она сформируется, и уменьшение количества букв в исходном слове.
11. Верните конечный результат и вызовите функцию с исходным строковым словом в качестве аргумента. Получившееся значение выведите в консоли.

Таймер обратного отсчета

Напишите таймер обратного отсчета, который выполнится в консоли и покажет общее количество миллисекунд, а также количество дней, часов, минут и секунд, оставшихся до достижения целевой даты.

1. Создайте конечную дату, к которой будет вестись отсчет. Переведите ее внутри строки в формат типа `date`.

2. Создайте функцию обратного отсчета, которая будет проводить парсинг `endDate()` и вычитать текущую дату из конечной. На экране появится общий результат в миллисекундах. С помощью `Date.parse()` вы можете преобразовать данное представление даты в количество миллисекунд, прошедших с 1 января 1970 года, 00:00:00 UTC.
3. Получив общее количество миллисекунд, чтобы рассчитать дни, часы, минуты и секунды, выполните следующие действия.
   - Чтобы получить дни, вы можете разделить количество миллисекунд в дате на нужное число, удалив остаток с помощью `Math.floor()`.
   - Для получения количества часов можно использовать модуль — тогда вы получите только часть, оставшуюся после удаления общего количества дней.
   - Чтобы получить минуты, используйте значение миллисекунд в минуте и, применив модуль, зафиксируйте остаток.
   - Сделайте то же самое для секунд, разделив число миллисекунд на 1000 и получив остаток. С помощью `Math.floor()` можно округлить значение в меньшую сторону, удалив все оставшиеся десятичные знаки, отображенные в меньших значениях.
4. Верните все значения внутри объекта с именами свойств, указывающими, к какой единице времени эти значения относятся.
5. Создайте функцию, которая с помощью `setTimeout()` каждую секунду (1000 миллисекунд) будет инициировать запуск функции `update()`. Функция `update()` создаст переменную, которая может временно хранить возвращаемые значения `countdown()`, и пустую переменную, которая будет использоваться для создания выходных значений.
6. Внутри той же функции, используя цикл `for`, получите все свойства и значения переменной объекта `temp`. По мере выполнения итераций по объекту обновляйте выходную строку, чтобы она содержала имя и значение свойства.
7. Выведите на экран результат в виде строки, используя `console.log()`.

Вопросы для самопроверки

1. Какой метод декодирует следующее:

```
var c = "http://www.google.com?id=1000&n=500";
var e = encodeURIComponent(c);
```

   1) `decodeURIComponent(e);`
   2) `e.decodeUriComponent();`
   3) `decoderURIComponent(c);`
   4) `decoderURIComponent(e)`?

2. Что выведет на экран следующий код?

```
const arr = ["hi","world","hello","hii","hi","hi World","Hi"];
console.log(arr.lastIndexOf("hi"));
```

3. Какой результат вы увидите на экране?

```
const arr = ["Hi","world","hello","Hii","hi","hi World","Hi"];
arr.copyWithin(0, 3, 5);
console.log(arr);
```

4. Что отобразится в консоли?

```
const arr = ["Hi","world","hello","Hii","hi","hi World","Hi"];
const arr2 = arr.filter((element,index)=>{
    const ele2 = element.substring(0, 2);
    return (ele2 == "hi");
});
console.log(arr2);
```

Резюме

В этой главе вы столкнулись с большим количеством встроенных методов, предоставленных нам самим JavaScript. Они полезны для решения многих популярных задач. Вы познакомились с наиболее часто применяемыми глобальными встроенными методами, которые настолько распространены, что их можно использовать без добавления объекта, которому они принадлежат.

Мы также обсудили методы массивов, строковые, числовые, математические методы и методы работы с датами. Когда вы лучше разберетесь в JavaScript, вы будете часто к ним прибегать и формировать цепочки методов (всякий раз, когда это будет давать результат).

Теперь, когда вы познакомились со многими функциями JavaScript, мы посвятим следующие пару глав тому, как они работают в HTML и взаимодействуют с браузером — оживим веб-страницы!

9 Объектная модель документа

Объектная модель документа (DOM) намного интереснее, чем может показаться на первый взгляд. Это фундаментальная концепция, с которой вам необходимо разобраться, прежде чем внедрять JavaScript на веб-страницы. Она берет HTML-страницу и превращает ее в разветвленную логическую схему. В этой главе мы познакомим вас с DOM. Если сталкиваетесь с HTML впервые, не переживайте. Первый раздел посвящен базе HTML, вы можете его пропустить, если уже владеете данной темой.

Как только мы убедимся, что находимся с вами на одной волне, перейдем к изучению *объектной модели браузера (BOM)*. BOM содержит все методы и свойства, которые позволят JavaScript взаимодействовать с браузером. Это информация, касающаяся ранее посещенных страниц, размера окна браузера, а также DOM.

DOM содержит в себе все HTML-элементы на странице. С помощью JavaScript мы можем выбирать части DOM и управлять ими, благодаря чему веб-страницы из статических превращаются в интерактивные. Да, владение навыками работы с DOM позволит вам создавать интерактивные веб-страницы!

Мы рассмотрим следующие темы:

- ускоренный курс HTML;
- введение в BOM;
- введение в DOM;
- типы элементов DOM;
- выбор элементов страницы.

Можем себе представить, как вам не терпится начать, поэтому давайте уже погрузимся в главу.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Ускоренный курс HTML

Hyper-Text Markup Language (HTML) — это язык, с помощью которого формируется содержание веб-страниц. Веб-браузеры понимают HTML-код и представляют его в формате, который мы привыкли видеть: в формате веб-страниц. Вот базовый пример HTML-кода:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Tab in the browser</title>
  </head>
  <body>
    <p>Hello web!</p>
  </body>
</html>
```

Вот как выглядит простая веб-страница (рис. 9.1).

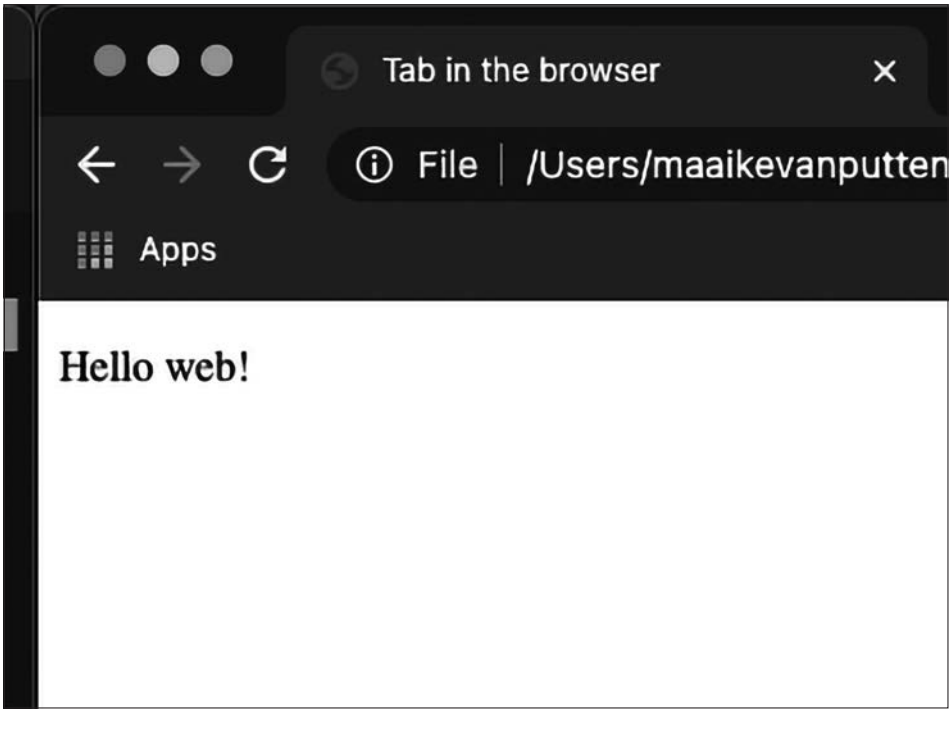


Рис. 9.1. Простая веб-страница

Код-HTML состоит из элементов. Данные элементы включают в себя теги и атрибуты, мы объясним эту фундаментальную концепцию в следующих разделах.

Элементы HTML

Как видите, HTML состоит из фрагментов кода (или, другими словами, элементов), размещенных между командами в `<угловых скобках>` — тегами. Любой открытый элемент должен быть закрыт. Мы открываем его с помощью такой комбинации — `<имя_элемента>`, а закрываем такой — `</имя_элемента>`.

Все, что находится между тегами, является частью элемента. Некоторые элементы все же закрывать не нужно, но вы с ними вряд ли столкнетесь. В предыдущем примере вы уже увидели несколько элементов, включая эти два: элемент с тегом `body` и внутренний элемент с тегом `p`:

```
<body>
  <p>Hello web!</p>
</body>
```

Итак, элементы могут включать в себя другие элементы. Элемент может быть закрыт, только если все внутренние элементы закрыты. Вот правильный способ:

```
<outer>
  <sub>
    <inner>
    </inner>
  </sub>
</outer>
```

А это неправильный:

```
<outer>
  <sub>
    <inner>
  </sub>
    </inner>
</outer>
```

Обратите внимание: это всего лишь выдуманные имена элементов. В последнем примере мы закрыли `sub` до того, как был закрыт элемент `inner`. Это неверное решение. Всегда сначала закрываются внутренние элементы, а за ними — внешние. Мы называем внутренние элементы дочерними, а внешние — родительскими. Вот небольшой фрагмент правильно выстроенного HTML-кода:

```
<body>
  <div>
    <p>
    </p>
  </div>
</body>
```

А это пример некорректного HTML-кода (там `div` закрыт раньше, чем внутренний элемент `p`):

```
<body>
  <div>
    <p>
  </div>
    </p>
</body>
```

Различные элементы представляют собой разные части макета. Тег `p`, который мы только что рассмотрели, описывает абзацы. Еще один распространенный элемент — `h1` — заголовок. Но гораздо важнее знать три основных элемента каждой HTML-страницы — это `html`, `head` и `body`.

Элемент `html` содержит в себе весь код веб-страницы, данный тег в ее структуре единственный. Это внешний элемент, и все остальные располагаются в его пределах. Он содержит в себе два других высших элемента: `head` и `body`. Если у вас когда-либо возникнут сомнения в порядке следования `head` и `body`, просто вспомните человеческое тело — голова (`head`) находится в верхней части тела (`body`).

В элемент `head` мы помещаем множество данных, предназначенных для браузера, а не для пользователя. Вы могли бы подумать об определенных метаданных, таких как скрипты JavaScript, включенные таблицы стилей, текст, который поисковая система должна использовать в качестве описания на странице результатов поиска. На самом деле, будучи разработчиками JavaScript, мы не будем много работать с `head`, разве что включать скрипты.

Вот пример базового элемента `head`:

```
<head>
    <title>This is the title of the browser tab</title>
    <meta name="description" content="This is the preview in google">
<script src="included.js"></script>
</head>
```

В элемент `body` включается основное содержимое, которое будет отображаться на веб-странице. Внутри `html` может содержаться только один `body`. Заголовки,

абзацы, изображения, списки, ссылки, кнопки и многое другое — все это вещи, с которыми мы можем столкнуться внутри `body`. У них есть свои собственные теги: например, `img` для изображений и `a` — для ссылок. Следующая таблица включает теги, которые могут использоваться в `body`, и это определенно не исчерпывающий список.

Тег открытия	Тег закрытия	Описание
`<p>`	`</p>`	Абзац
`<h1>`	`</h1>`	Заголовок первого уровня (заголовки более низких уровней описываются соответствующими тегами от `h2` до `h6`)
`<span>`	`</span>`	Универсальный встроенный контейнер для содержимого, которое необходимо разделить (например, для проектирования шаблонов)
`<a>`	`</a>`	Гиперссылка
`<button>`	`</button>`	Кнопка
`<table>`	`</table>`	Таблица
`<tr>`	`</tr>`	Строка таблицы (используется только внутри тега таблицы)
`<td>`	`</td>`	Ячейка таблицы (создается внутри строки)
`<ul>`	`</ul>`	Неупорядоченный маркированный список
`<ol>`	`</ol>`	Упорядоченный нумерованный список
`<li>`	`</li>`	Внутренние элементы упорядоченных и неупорядоченных списков
`<div>`	`</div>`	Раздел внутри HTML-страницы. Он часто используется в качестве контейнера для других стилей или разделов и легко применим в нетипичной верстке
`<form>`	`<form>`	Форма HTML
`<input>`	`</input>`	Поле для пользовательского ввода (текстовые поля, флажки, кнопки, пароли, цифры, выпадающие списки, переключатели и многое другое)
`<input />`	Нет	То же, что и `input`, но написанный без закрывающего тега: `/` внутри угловых скобок делает его самозакрывающимся. Подобное возможно только для некоторых элементов
` `	Нет	Конец строки (переход на новую строку). Он не нуждается в конечном теге и поэтому также является исключением из правила

Можете ли вы понять, что делает этот HTML-код?

```
<html>

<head>
    <title>Awesome</title>
</head>

<body>
    <h1>Language and awesomeness</h1>
    <table>
        <tr>
            <th>Language</th>
            <th>Awesomeness</th>
        </tr>
        <tr>
            <td>JavaScript</td>
            <td>100</td>
        </tr>
        <tr>
            <td>HTML</td>
            <td>100</td>
        </tr>
    </table>
</body>

</html>
```

Он создает веб-страницу. В заголовке вкладки мы видим надпись `Awesome`. На самой странице есть заголовок первого уровня с надписью `Language and awesomeness`. Затем идет таблица с тремя строками и двумя столбцами. Первая строка — головка таблицы с заголовками `Language` и `Awesomeness`. Вторая строка содержит значения `JavaScript` и `100`. А в третьей строке видим `HTML` и `100`.

Атрибуты HTML

Последняя часть, которую мы обсудим в этом разделе, — это атрибуты HTML. Атрибуты задают свойства элемента, в котором они указаны. Они находятся внутри этого элемента — им присваивается соответствующее значение со знаком равенства. Например, атрибутом `a` (гиперссылки) является `href`. Он указывает, куда должна вести ссылка:

```
<a href="https://google.com">Ask Google</a>
```

Этот код создает ссылку с названием `Ask Google`. При нажатии на нее вы будете перенаправлены на поисковую страницу Google, о чем можно судить по значению атрибута `href`. Таких атрибутов огромное множество, но пока вам достаточно просто знать, что они изменяют свойства элемента, в котором они указаны.

Основные атрибуты, наиболее важные для начала работы с HTML и JavaScript. Почему они важны, мы расскажем в следующей главе.

Название атрибута	Описание	К каким элементам может применяться
`id`	Присваивает элементу уникальный ID (например, задающий возраст)	Ко всем
`name`	Присваивает элементу пользовательское название	`input`, `button`, `form` и еще к таким, которых мы пока не встречали
`class`	Специальные метаданные, которые могут быть добавлены в элемент и вызвать определенные изменения в макете или другие JavaScript-манипуляции	Почти ко всем внутри `body`
`value`	Задает начальное значение элемента	`button`, `input`, `li` и еще к таким, которых мы пока не встречали
`style`	Задает определенные стили HTML-элемента, в котором указывается	Ко всем

С другими атрибутами мы познакомим вас по мере вашего продвижения в познании магии JavaScript.

Что ж, это был один из самых кратких курсов HTML. Существует множество замечательных ресурсов, где вы можете найти дополнительную информацию. Если вам на данном этапе нужны какие-то разъяснения, создайте и выполните следующий HTML-код и извлеките ответы оттуда!

```
<!DOCTYPE html >
<html>

<body>
    <a href="https://google.com">Ask google</a>
</body>

</html>
```

А мы продолжим — давайте теперь взглянем на BOM и его различные элементы.

BOM

BOM (Browser Object Model) — объектная модель браузера, иногда также называемая *браузерным объектом* `window`, является удивительным «волшебным» элементом, который позволяет вашему коду JavaScript взаимодействовать с браузером.

Объект `window` содержит все свойства, необходимые для описания окна браузера: например, размер окна и историю ранее посещенных веб-страниц. Данный объект

содержит в себе глобальные переменные и функции — все это мы рассмотрим при более детальном его изучении. Точная реализация BOM зависит от браузера и его версии, важно иметь это в виду.

Вот некоторые из важнейших объектов BOM, которые мы рассмотрим далее в этой главе:

- `history`;
- `navigator`;
- `location`.

Мы обязательно вернемся к DOM и рассмотрим его более подробно, но давайте сначала изучим BOM и просмотрим ее объекты с помощью команды `console.dir(window)`. Введем ее в консоль браузера, но прежде обсудим, как вообще туда попасть.

На контрольной панели браузера можно получить доступ к различным элементам HTML и JavaScript. Туда можно попасть разными способами — самыми распространенными являются нажатие клавиши F12 в открытом браузере или щелчок правой кнопкой мыши на веб-сайте, консоль которого вы хотите увидеть, и выбор пункта Inspect element (Проверить элемент) или Inspect (Проверить) на устройстве macOS. В результате появится боковая панель (или отдельное окно, если вы изменили настройки браузера) (рис. 9.2).

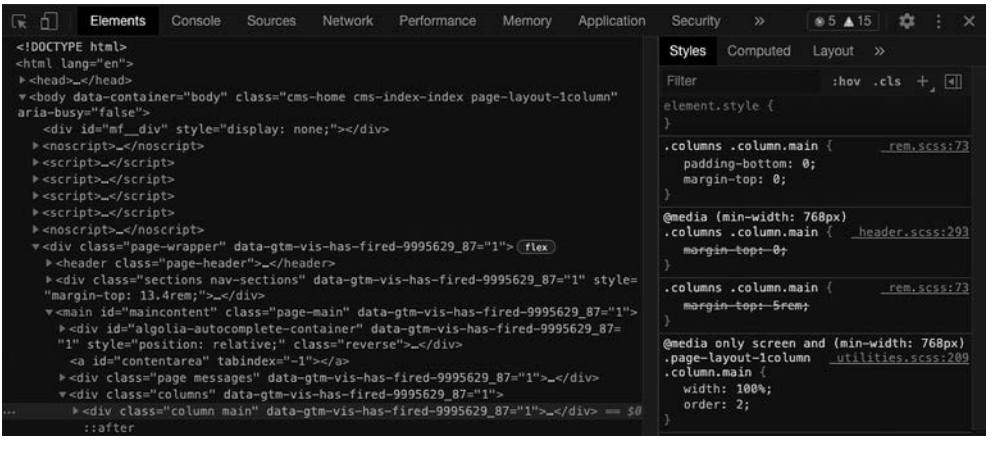


Рис. 9.2. Проверка страницы в браузере

Перейдите на вкладку Console (Консоль), которая располагается рядом со вкладкой Elements. Можете ввести следующую команду и нажать клавишу Enter, чтобы получить информацию об объекте в окне:

```
console.dir(window);
```

Данная команда приведет страницу к виду, похожему на представленный на рис. 9.3.

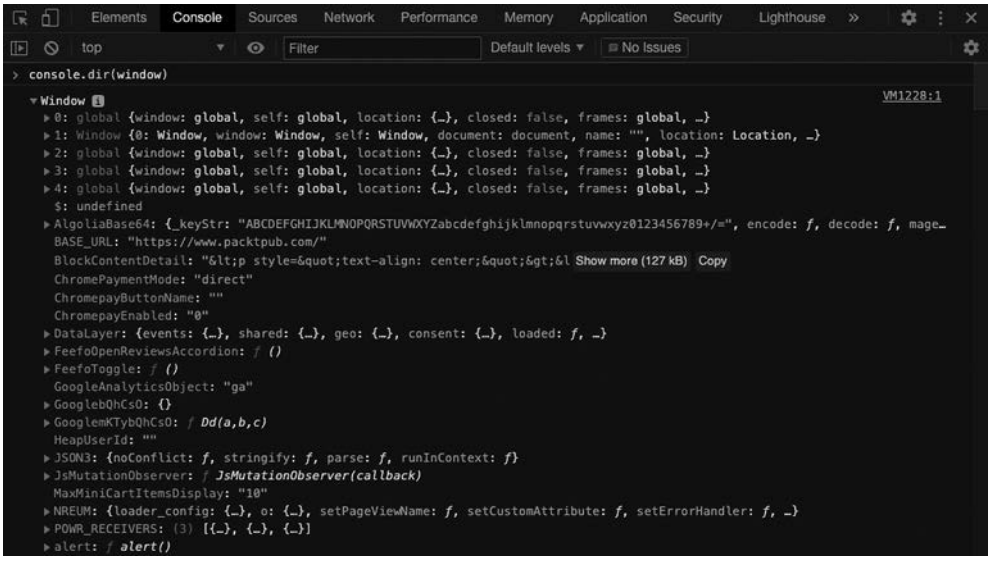


Рис. 9.3. Часть выводимых командой console.dir(window) данных об объекте в окне браузера

Метод `console.dir()` отображает список всех свойств выбранного объекта. Вы можете щелкнуть на маленьких треугольниках, чтобы открыть описание объекта и увидеть более подробную информацию о нем.

BOM содержит много других объектов. Мы можем получить к ним доступ так же, как и к другим объектам, уже знакомым нам. Так, можно получить длину объекта `history` (по крайней мере это рабочий способ для собственного браузера одного из авторов), обращаясь к объекту окна `history`, а затем к длине объекта `history`, например:

```
window.history.length;
```

Сначала сделаем упражнение, а потом изучим объект `history` подробнее.

Практическое занятие 9.1

1. Перейдите на сайт, который вы только что просматривали, и выполните команду `console.dir(window)`.
2. Можете найти объект `document`, вложенный в объект `window`? Сделайте это, перейдя вниз в корневом каталоге объекта `window` в консоли.
3. Получится ли у вас определить высоту и ширину вашего окна (в пикселях)? Вы можете вернуть внутреннее и внешнее окно.

Объект history

Объект браузера `window` содержит в себе объект `history` — его фактически можно указывать без префикса `window`, поскольку он задумывался как глобальный. Следовательно, получить его можно командой `console.dir(window.history)` или просто `console.dir(history)` (рис. 9.4).

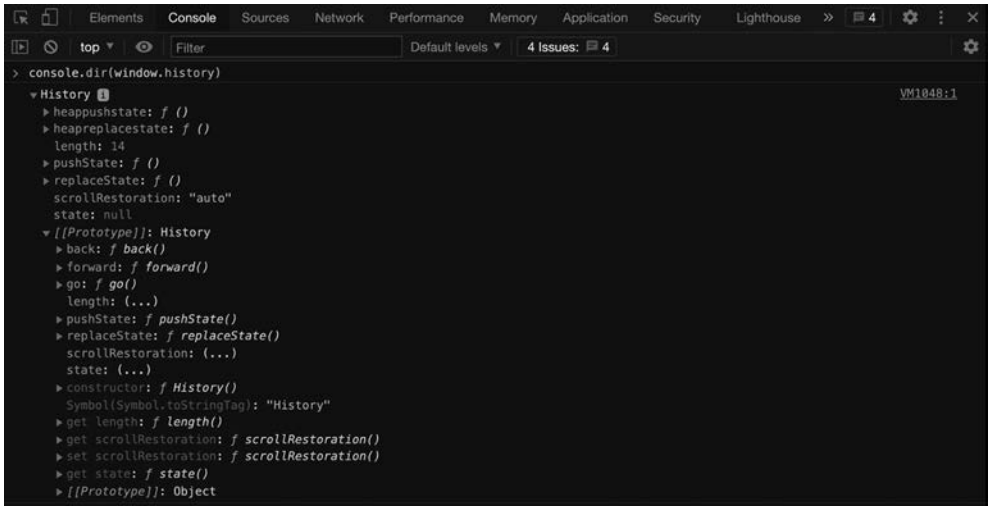


Рис. 9.4. Объект history

Объект `history` применятся для перехода на предыдущую страницу. В нем есть встроенная функция, называемая `go`. Что произойдет, когда вы выполните следующую команду?

```
window.history.go(-1);
```

А теперь попробуйте выполнить ее в консоли своего браузера (перед этим убедитесь, что вы посещали несколько страниц на текущей вкладке).

Объект navigator

В объекте `window`, который мы только что рассматривали, есть свойство `navigator`. Оно особенно интересно, поскольку содержит в себе информацию об используемом нами браузере — указывая, что это за браузер, какова его текущая версия и в какой операционной системе он функционирует.

Данное свойство может быть удобным при настройке сайта на работу в определенных операционных системах. Представьте себе кнопку загрузки, которая будет различаться для Windows, Linux и macOS.

Исследуем этот объект с помощью следующей команды в консоли:

```
console.dir(window.navigator);
```

Как вы заметили, мы добавили `window`, потому что объект `navigator` входит в объект `window`. Следовательно, перед нами свойство объекта `window`, которое мы определяем, используя точку между словами. Мы получаем доступ к объектам `window` так же, как и к любому другому объекту. Но объект `navigator` глобальный, поэтому мы можем обратиться к нему и без префикса:

```
console.dir(navigator);
```

Вот как может выглядеть объект `navigator` (рис. 9.5).

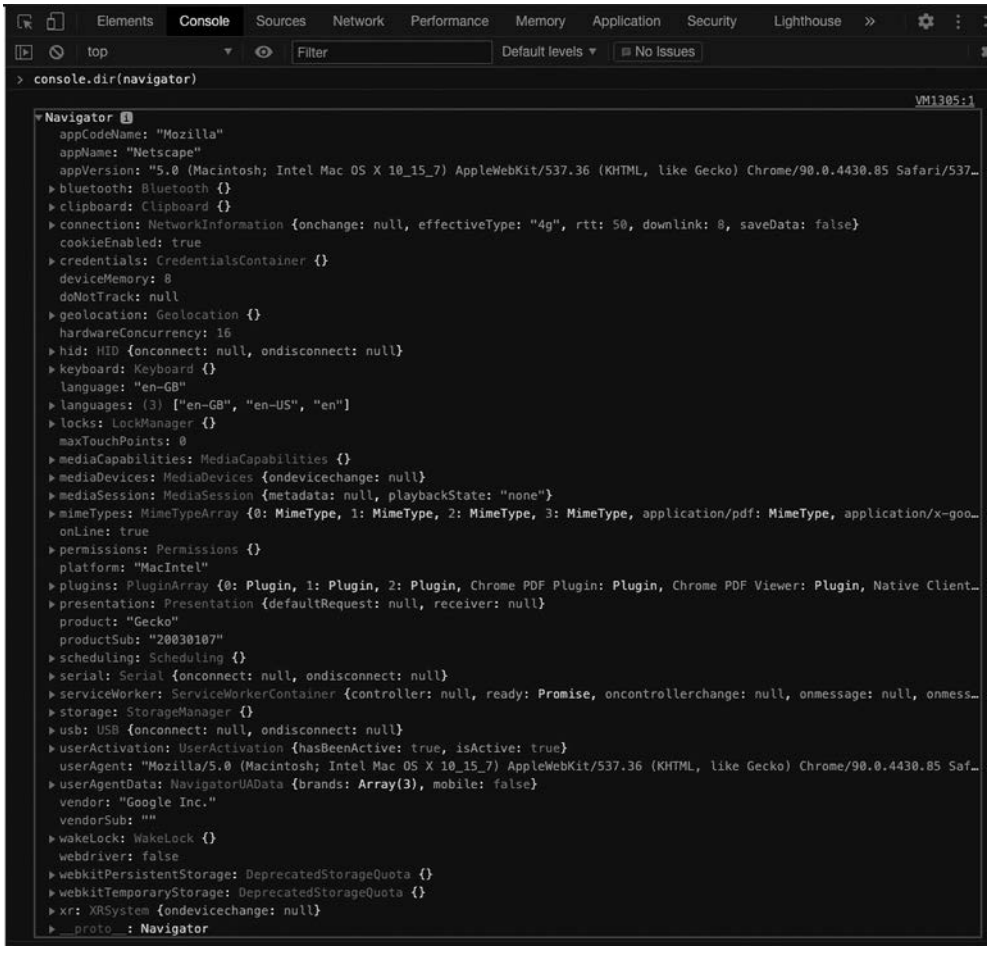


Рис. 9.5. Объект navigator

Объект location

Другое интересное свойство `window` — объект `location`. Он содержит URL текущей веб-страницы. Если переписать это свойство (или его часть), можно заставить браузер перейти на другую страницу! Как это сделать, зависит от браузера. Следующее упражнение поможет вам в этом разобраться.

Объект `location` состоит из нескольких свойств. Их можно увидеть с помощью команды `console.dir(window.location)` или `console.dir(location)`. Вот как выглядит результат (рис. 9.6).

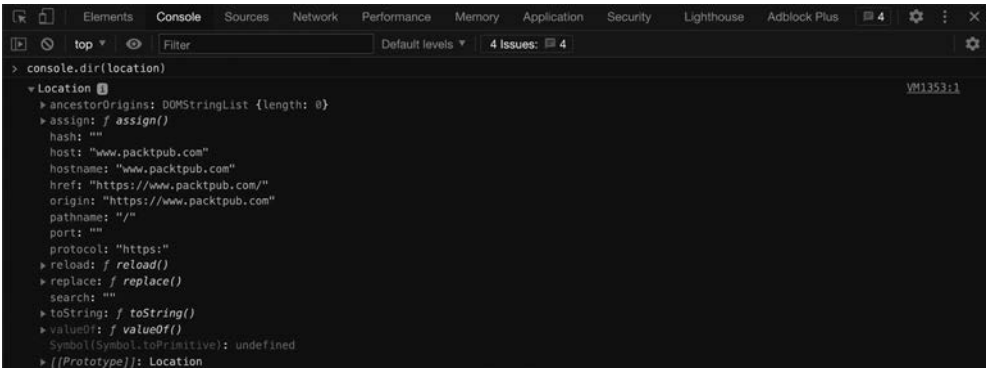


Рис. 9.6. Объект location

В `location`, как и в описываемых ранее объектах, содержится много других объектов. Мы можем получить доступ к вложенным объектам и свойствам, используя точки (как делали до сих пор). Введем следующее:

```
location.ancestorOrigins.length;
```

Так мы получим значение длины объекта `ancestorOrigins`, который показывает, со сколькими контекстами просмотра связана наша страница. Это может быть полезно для определения того, оформлена ли веб-страница в неожиданном контексте. Однако не все браузеры содержат данный объект, опять же вид BOM и всех ее элементов в разных браузерах различается.

Следуйте пунктам практического занятия, чтобы сотворить такую же магию самостоятельно.

Практическое занятие 9.2

Просмотрите объект `window` и доберитесь до объекта `location`. После этого выведите на экран значения свойств `protocol` и `href` текущего файла.

DOM

На самом деле DOM не так уж трудно понять. Это способ отображения структуры HTML-документа в виде логической структуры. Подобное стало возможным благодаря соблюдению строгого правила, гласящего, что внутренние элементы закрываются перед закрытием внешних.

Перед вами фрагмент HTML:

```
<html>
  <head>
    <title>Tab in the browser</title>
  </head>
  <body>
    <h1>DOM</h1>
    <div>
      <p>Hello web!</p>
      <a href="https://google.com">Here's a link!</a>
    </div>
  </body>
</html>
```

И вот как это можно показать в виде дерева (рис. 9.7).

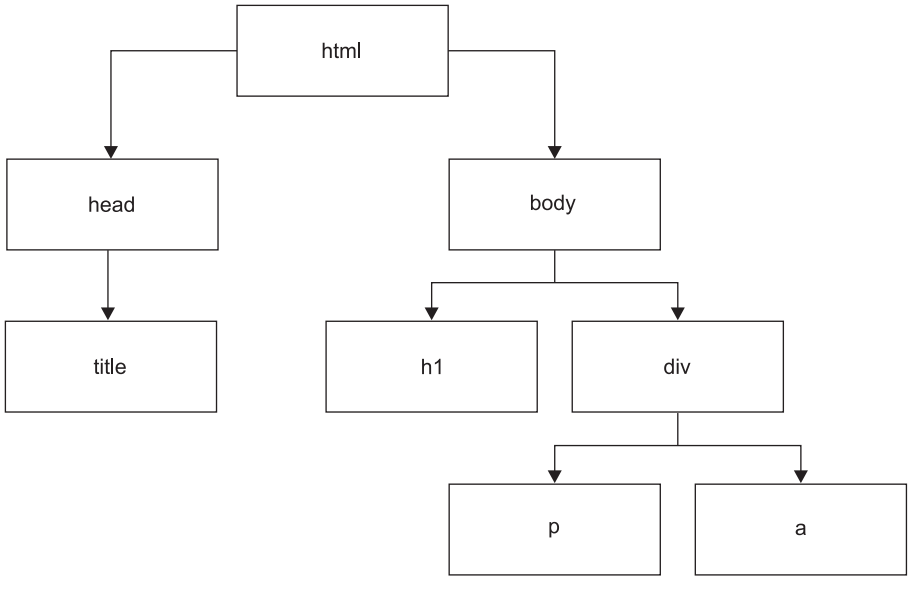


Рис. 9.7. Структура DOM для базовой веб-страницы

Как видите, внешние элементы `html` располагаются в вершине схемы. Следующие уровни, `head` и `body`, являются дочерними. У `head` только один дочерний элемент —

`title`. А `body` содержит два дочерних элемента: `h1` и `div`. `div`, в свою очередь, включает два элемента — `p` и `a`. Они обычно описывают абзацы и ссылки (или кнопки). Безусловно, сложные страницы имеют более сложную структуру, именно она со множеством дополнительных свойств формирует DOM веб-страницы.

DOM реального сайта не поместится на странице этой книги. Но умение рисовать подобные деревья в воображении очень поможет вам в будущем.

Дополнительные свойства DOM

Можно исследовать DOM аналогичным другим моделям способом. Выполним следующую команду в консоли веб-сайта (напомним: объект `document` глобальный, поэтому вызывать его через `window` нет необходимости):

```
console.dir(document);
```

Мы хотим отобразить объект `document`, который описывает DOM (рис. 9.8).

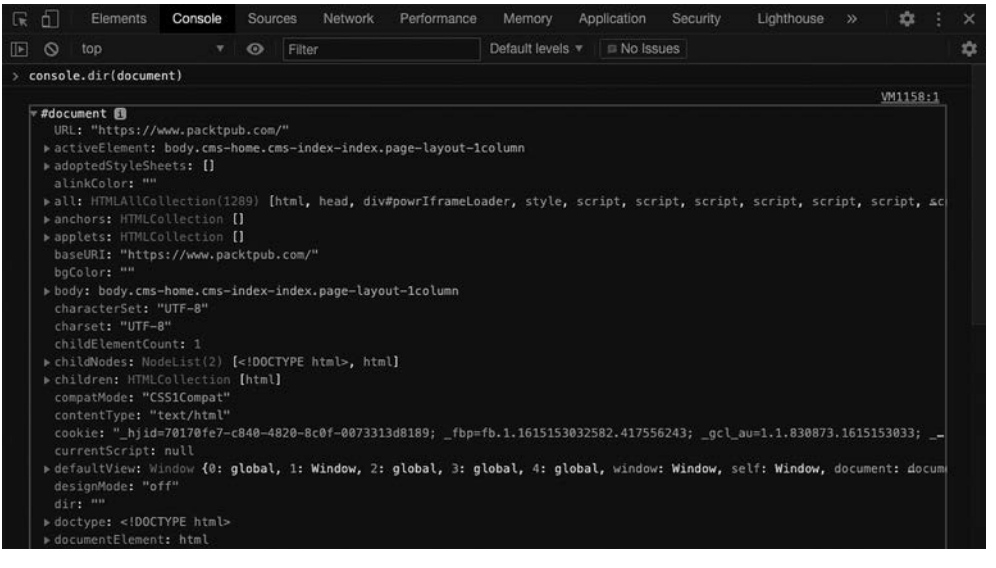


Рис. 9.8. DOM

Вам не нужно понимать каждую строку в представленном коде, но там можно увидеть многое, включая элементы HTML и код JavaScript.

Отлично, вы познакомились с основами BOM и ее дочерним объектом DOM, представляющим для нас на данном этапе наибольший интерес. Мы уже видели много свойств DOM, а поскольку DOM содержит в себе все элементы HTML-страницы, далее мы будем рассматривать их именно там. Зная основы DOM, а также владея

некоторым навыком в управлении и исследовании объектной модели, вы сможете сделать гораздо больше.

В следующей главе мы сосредоточимся на просмотре DOM, поиске в ней элементов и ее управлении. Код, который мы начнем писать, все более будет похож на код реальных проектов.

Выбор элементов страницы

Чтобы работать с элементами на странице, необходимо их найти. Объект `document` содержит в себе множество свойств и методов. Но прежде, чем изменить значение определенного абзаца, нужно получить его (мы называем это выбором абзаца). Только после этого можно приступать к работе со значениями.

Чтобы выбрать элементы страницы и далее управлять ими в JavaScript-коде, можно применить методы `querySelector()` или `querySelectorAll()`. Оба они могут использоваться для выбора элементов страницы по имени тега, идентификатору или классу.

Метод `document.querySelector()` вернет первый элемент документа, совпадающий с заданным значением. Если совпадений не существует, будет возвращен результат `null`. Чтобы вернулось множество совпадающих значений, необходимо применить метод `document.querySelectorAll()`.

Метод `querySelectorAll()` вернет статический список `NodeList`, в котором будут представлены названия элементов, соответствующих заданному значению. Следующий фрагмент HTML-кода демонстрирует применение методов `querySelector()` и `querySelectorAll()`.

```
<!doctype html>
<html>
  <head>
    <title>JS Tester</title>
  </head>
  <body>
    <h1 class="output">Hello World</h1>
    <div class="output">Test</div>
  </body>
</html>
```

С помощью `querySelector()` выберем элемент `h1`. Если таких значений в документе больше одного, метод захватит только первое.

```
const ele1 = document.querySelector("h1");
console.dir(ele1);
```

Для выбора множества элементов используйте `querySelectorAll()`: он вернет все элементы, которые совпадают с заданным значением, в виде массива. С по-

мощью следующего кода найдем экземпляры класса `output`, добавляя к имени класса точку.

```
const eles = document.querySelectorAll(".output");
console.log(eles);
```

Выбрав необходимые элементы, вы можете начать использовать динамические функции DOM и управлять элементами с помощью JavaScript. Содержимое можно изменять (так же, как и содержимое переменной), элементы можно удалять или добавлять, а стили можно корректировать. Все это делается с помощью JavaScript, и то, как пользователь взаимодействует со страницей, может повлиять на процесс.

Мы рассмотрели два распространенных метода выбора: `querySelector()` и `querySelectorAll()`. С их помощью можно выбрать не только описанные ранее элементы, но и, в принципе, любой нужный. Есть еще много других способов управления DOM, с ними вы столкнетесь в следующей главе.

Практическое занятие 9.3

Выберите какой-либо элемент страницы и обновите его содержимое. Измените его стиль и добавьте атрибуты.

Для этого создайте HTML-файл, содержащий элемент страницы с классом `output`:

```
<!DOCTYPE html >
<html>

<div class="output"></div>
    <script>
    </script>

</html>
```

Внутри тега `<script>` пропишите следующие изменения.

1. Выберите элемент страницы как объект JavaScript.
2. Обновите его свойство `textContent`.
3. Добавьте класс `red`, используя метод объекта `classList.add`.
4. Обновите `id` до значения `tester`.
5. Через объект `style` добавьте элементу страницы свойство `backgroundColor` с атрибутом `red`.
6. Получите URL-адрес документа с помощью `document.URL` и обновите текст элемента, чтобы он содержал значение URL-документа. Выведите значение на экран, чтобы убедиться, что значение верное.

Проект текущей главы

Управление элементами HTML с помощью JavaScript

Возьмите HTML-код:

```
<div class="output">
    <h1>Hello</h1>
    <div>Test</div>
    <ul>
        <li id="one">One</li>
        <li class="red">Two</li>
    </ul>
    <div>Test</div>
</div>
```

Осуществите следующие шаги (и поэкспериментируйте дальше), чтобы закрепить способы, которыми можно управлять HTML-кодом с помощью кода JavaScript.

1. Выберите элемент, используя класс `output`.
2. Создайте другой объект JavaScript с названием `mainList` и выберите тег `ul` (находится внутри элемента `output`). Измените `id` тега на `mainList`.
3. Проведите поиск `tagName` в каждом `div` и выведите полученные значения на экран в виде массива.
4. С помощью цикла `for` установите идентификатор каждого тега `div` в виде `id` с числом, соответствующим порядку вывода значений. Все еще используя цикл, измените цвет содержимого каждого элемента в `output` на красный или синий.

Вопросы для самопроверки

1. Перейдите на свой любимый сайт и откройте консоль браузера. Наберите `document.body`. Что вы видите на экране?
2. Как мы знаем, при работе с объектами можно записать значение свойства и задать новое значение с помощью оператора присваивания. Обновите свойство `textContent` в объекте `document.body` на веб-странице таким образом, чтобы оно содержало строку `Hello World`.
3. Перечислите свойства и значения объектов BOM. Попробуйте прописать их в объекте `document`.
4. Сделайте то же самое для объекта `window`.
5. Создайте HTML-файл, в котором был бы тег `h1`. Выберите `h1` и, используя JavaScript, присвойте ему значение переменной. Измените свойство `textContent` переменной на `Hello World`.

Резюме

Эту главу мы начали с изучения основ HTML. Вы узнали, что HTML состоит из элементов, которые могут содержать в себе другие, вложенные элементы. Каждый элемент определяется тегами, указывающими на его тип. У элемента могут быть атрибуты, изменяющие его или добавляющие к нему метаданные, эти атрибуты могут быть использованы JavaScript.

Затем мы рассмотрели BOM, описывающую окно браузера, которое используется для веб-страницы и содержит такие объекты, как `history`, `location`, `navigator` и `document`. Объект `document` называется DOM, скорее всего, с ним вы будете встречаться постоянно. Именно он содержит HTML-элементы веб-страницы.

Мы также начали рассматривать, как можно выбирать элементы документа и с их помощью управлять веб-страницей, с этим вопросом мы продолжим разбираться в следующей главе.

10 Управление динамическими элементами с помощью DOM

В этой главе ваши усилия в изучении сложных концепций будут вознаграждены. Мы сделаем шаг вперед и узнаем, как с помощью JavaScript управлять элементами DOM на странице. Сначала научимся ориентироваться в DOM и выбирать желаемые элементы. Изучим способы добавления и изменения значений и атрибутов, а также добавим новые элементы в DOM.

Вы также узнаете, как добавлять стили элементов, которые помогают этим элементам исчезать или появляться. Познакомитесь с событиями и прослушивателями событий. Начав с самого простого, к концу этой главы вы сможете различными способами управлять веб-страницами и получите все необходимые знания для создания базовых веб-приложений. С получением такого навыка ваши возможности станут неограниченными.

Мы рассмотрим следующие темы:

- базовое перемещение в DOM;
- доступ к элементам DOM;
- обработчик щелчка кнопкой мыши на элементе;
- ключевое слово `this` и DOM;
- управление стилем элемента;
- изменение классов элементов;
- управление атрибутами;
- прослушиватели событий;
- создание новых элементов.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Мы уже знаем многое о DOM. Чтобы взаимодействовать с веб-страницей и делать ее динамической, мы должны подключить наши JavaScript-навыки.

Базовое перемещение в DOM

Используя объект `document`, знакомый из предыдущей главы, мы можем перемещаться по DOM. Он содержит в себе весь код HTML и является представлением веб-страницы. Прохождение по этим данным может привести вас к элементу, которым нужно управлять.

Далее мы рассмотрим не самый распространенный способ решения этой задачи, но по крайней мере он поможет вам понять, как все работает. Согласитесь, иногда нам действительно нужны такие методы. Есть и другие способы — и они будут раскрыты в текущей главе.

Даже для простого HTML-фрагмента существует несколько способов пройти через DOM. Давайте отправимся на поиск сокровищ. Начнем с небольшого HTML-фрагмента:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Let's find the treasure</h1>
    <div id="forest">
      <div id="tree1">
        <div id="squirrel"></div>
        <div id="flower"></div>
      </div>
      <div id="tree2">
        <div id="shrubbery">
          <div id="treasure"></div>
        </div>
        <div id="mushroom">
          <div id="bug"></div>
        </div>
      </div>
    </div>
  </body>
</html>
```

Чтобы найти сокровище, мы хотим исследовать DOM этого фрагмента. Можно сделать это, войдя в объект `document` и двигаясь от этого места дальше. Проще всего выполнить данное упражнение в консоли браузера: таким образом, вы будете четко понимать, где находитесь.

Можем начать с использования свойства `body` объекта `document`. В нем содержится все, что находится в элементе `body`. Введем в консоль:

```
console.dir(document.body);
```

Должен вернуться очень длинный объект. Добраться от этого объекта до нашего сокровища можно несколькими способами. Прежде всего давайте обсудим потомков и свойство `childNodes` (дочерние узлы).

childNodes — более полное явление, нежели потомки. Потомки просто содержат все HTML-элементы (по сути, узлы). childNodes содержит еще текстовые узлы и комментарии. Для работы с потомками тем не менее вы можете использовать идентификаторы, и это упрощает их применение.

Чтобы добраться до сокровища с помощью потомков, вам придется использовать:

```
console.dir(document.body.children.forest.children.tree2.children.
shrubbery.children.treasure);
```

Как видим, для каждого выбранного нами элемента необходимо снова определить дочерние элементы. Сначала мы получаем потомков из тела. Затем выбираем из дочерних элементов `forest`. После этого снова обращаемся к дочерним элементам `forest` и из них выбираем `tree2`. Далее опять получаем дочерние элементы `tree2` — из них нам требуется `shrubbery`. И наконец, обращаемся к дочерним элементам `shrubbery` и выбираем `treasure` (сокровище).

Чтобы получить сокровище с помощью `childNodes`, нужно очень много пользоваться консолью, потому что текст и узлы тоже находятся там. `childNodes` является массивом, поэтому для получения правильных дочерних элементов придется указывать точные значения индекса. Но есть и одно преимущество: такой способ намного короче, ведь вам не понадобится отдельно выбирать имя.

```
console.dir(document.body.childNodes[3].childNodes[3].childNodes[1].
childNodes[1]);
```

Вы также можете комбинировать фрагменты:

```
console.dir(document.body.childNodes[3].childNodes[3].childNodes[1].
children.treasure);
```

Существует много способов перемещения по документу. Вы можете использовать тот или иной в зависимости от ваших потребностей. Для задач, требующих перемещения по DOM, обычно действует следующая логика: это работает — значит, это хорошее решение.

До сих пор мы рассматривали перемещение по DOM вниз, но можно двигаться в обратную сторону. Каждый элемент имеет связь со своим родителем, поэтому, чтобы двигаться вверх, мы можем использовать свойство `parentElement`. Например, если взять образец HTML с сокровищем и написать в консоли:

```
document.body.children.forest.children.tree2.parentElement;
```

код вернет нас к уровню `forest`, родительскому для `tree2`. Такой способ может быть очень полезен, в частности, в сочетании с такими функциями, как `getElementById()`, позже мы рассмотрим ее более подробно.

Перемещаться можно не только вверх и вниз, но и в стороны. Например, если выбрать `tree2` таким образом:

```
document.body.children.forest.children.tree2;
```

можно переместиться к `tree1`, используя:

```
document.body.children.forest.children.tree2.previousElementSibling;
```

В свою очередь, от `tree1` можно переместиться к `tree2` так:

```
document.body.children.forest.children.tree1.nextElementSibling;
```

В качестве альтернативы `nextElementSibling`, который возвращает значение следующего узла, являющегося элементом, можно применять `nextSibling`, который вернет следующий узел независимо от его типа.

Практическое занятие 10.1

В этом упражнении поэкспериментируйте с перемещением в иерархии DOM. Можете использовать следующий HTML-образец веб-сайта:

```
<!doctype html>
<html><head><title>Sample Webpage</title></head>
<body>
    <div class="main">
        <div>
            <ul >
                <li>One</li>
                <li>Two</li>
                <li>Three</li>
            </ul>
        </div>
        <div>blue</div>
        <div>green</div>
        <div>yellow</div>
        <div>Purple</div>
    </div>
</body>
</html>
```

Выполните следующие шаги.

1. Напишите указанный образец веб-страницы (или перейдите на любимый веб-сайт). Откройте тело документа в консоли с помощью `console.dir(document)`.
2. Выберите несколько дочерних элементов в свойстве `body.children`. Проверьте, насколько они совпадают с содержимым страницы.
3. Перейдите к следующим узлам или элементам и выведите их на экран.

Выбор элементов в качестве объектов

Теперь мы знаем, как перемещаться в DOM и вносить изменения в элементы. Вместо прописывания `console.dir()` мы можем просто задать путь к необходимому элементу. Сейчас мы можем рассматривать его как объект JavaScript и изменять любые его свойства. Возьмем простую HTML-страницу.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Welcome page</h1>
    <p id="greeting">
      Hi!
    </p>
  </body>
</html>
```

Мы можем переместиться, например, к элементу `p`, используя такой код:

```
document.body.children.greeting;
```

Теперь мы можем напрямую управлять свойствами элемента и самим элементом!

Изменение свойства innerText

Свойство `innerText` описывает текст между открывающим и закрывающим тегами:

```
<element>here</element>
```

Полученное значение `here` будет представлено на странице в виде обычного текста. Например, если перейти в консоль и написать:

```
document.body.children.greeting.innerText = "Bye!";
```

сообщение, отображаемое на странице, незамедлительно изменится с `Hi!` на `Bye!`. Поскольку `innerText` возвращает содержимое элемента в виде простого текста, в нашем случае проблем не возникнет: ведь у нас действительно между тегами помещен только текст. Однако, если внутри необходимого нам элемента есть какой-либо HTML-код (или вы хотите его добавить), данный метод использовать нельзя, ведь он интерпретирует HTML как обычную строку. Так, если выполнить:

```
document.body.children.greeting.innerText = "<p>Bye!</p>";
```

на экране появится `<p>Bye!</p>` — HTML-код в том виде, как если бы это была текстовая строка. Чтобы обойти проблему, необходимо использовать `innerHTML`.

Изменение свойства innerHTML

Если вы не работаете только с текстом или, возможно, хотите указать какое-либо HTML-форматирование для выбранного вами элемента, используйте свойство

innerHTML. Оно не просто обрабатывает обычный текст, но и выполняет роль внутреннего HTML-элемента:

```
document.body.children.greeting.innerHTML = "<b>Bye!</b>";
```

Представленный код выведет на экран выделенную полужирным надпись **Bye!**. Он примет во внимание тег <b>, распознав, что это именно тег, а не строковое значение.

Мы обещали показать вам более удобный, чем перемещение по DOM, способ доступа к элементам. Давайте посмотрим, как его получить.

Доступ к элементам DOM

Элементы DOM можно выбрать несколькими методами. Как только вы получите эти элементы, вы сможете их изменять. Далее мы обсудим получение элементов по идентификатору, названию тега, имени класса и с помощью CSS-селектора.

Вместо того чтобы шаг за шагом проходить через данные, как мы делали ранее, мы собираемся использовать встроенные методы, способные перемещаться по DOM и возвращать элементы, соответствующие заданным параметрам.

В качестве примера возьмем следующий HTML-фрагмент:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Just an example</h1>
    <div id="one" class="example">Hi!</div>
    <div id="two" class="example">Hi!</div>
    <div id="three" class="something">Hi!</div>
  </body>
</html>
```

Сначала рассмотрим доступ к элементам по идентификатору.

Доступ к элементам по идентификатору

Получить доступ к элементам по идентификатору можно с помощью метода getElementById() — он вернет один элемент с указанным ID. Идентификаторы должны быть уникальными, только тогда вы получите из HTML-документа единственный результат. Правил присваивания ID не так уж много: название должно состоять минимум из одного символа, в нем не должно быть пробелов и, как в случае с переменными, лучше, чтобы оно было описательным и не содержало специальных символов.

Если нам нужен элемент с идентификатором `two`, можно использовать следующий синтаксис:

```
document.getElementById("two");
```

Данная операция вернет полный HTML-элемент:

```
<div id="two" class="example">Hi!</div>
```

Напоминаем: если у вас более одного элемента с одинаковыми идентификаторами, метод просто вернет вам первый, с которым столкнется. Избегайте таких ситуаций в своем коде.

Вот как можно оформить файл с JavaScript внутри HTML-страницы, вместо того чтобы постоянно что-то запрашивать в консоли браузера:

```
<html>
  <body>
    <h1 style="color:pink;">Just an example</h1>
    <div id="one" class="example">Hi!</div>
    <div id="two" class="example">Hi!</div>
    <div id="three" class="something">Hi!</div>
  </body>
  <script>
    console.log(document.getElementById("two"));
  </script>
</html>
```

На экран будет выведен весь HTML-фрагмент `div` с `id="two"`.

Практическое занятие 10.2

Попробуйте получить несколько элементов по их идентификаторам.

1. Создайте HTML-элемент и назначьте ему идентификатор.
2. Выберите элемент страницы, используя его идентификатор.
3. Выведите выбранный элемент страницы на экран.

Доступ к элементам по названию тега

Если мы запросим элемент по имени тега, то получим результат в виде массива. Так происходит, потому что один и тот же тег может описывать разные элементы. Получится собрание HTML-элементов — `HTMLCollection`, особый объект JavaScript. По сути, это просто список узлов.

Выполните следующую команду в консоли:

```
document.getElementsByTagName("div");
```

Она вернет результат:

```
HTMLCollection(3) [div#one.example, div#two.example, div#three.something,
one: div#one.example, two: div#two.example, three: div#three.something]
```

Как видите, возвращены все элементы DOM с тегом `div`. По синтаксису видны их идентификаторы и классы. Первыми идут объекты: `div` — это название; `#` определяет идентификатор, а `.` — класс. Если точек много, значит, классов тоже много. Теперь вы снова сможете просмотреть элементы (`namedItems`), на этот раз в качестве пары «ключ — значение» с их ID в качестве ключа.

Метод `item()` позволяет получить доступ к элементам, используя их индекс, например:

```
document.getElementsByTagName("div").item(1);
```

Результатом будет:

```
<div id="two" class="example">Hi!</div>
```

Также можно запросить элементы по их именам — с помощью `namedItem()`:

```
document.getElementsByTagName("div").namedItem("one");
```

В консоли отобразится:

```
<div id="one" class="example">Hi!</div>
```

Результат вернется в виде `HTMLCollection`, даже если совпадение будет всего одно. В нашем фрагменте всего один тег `h1`, продемонстрируем поведение кода на нем:

```
document.getElementsByTagName("h1");
```

Вывод на экран:

```
HTMLCollection [h1]
```

Поскольку у `h1` нет идентификатора и класса, мы увидим в записи только `h1`. И раз у него нет идентификатора, данный элемент не является `namedItem` и встречается только один раз.

Практическое занятие 10.3

Используйте JavaScript для выбора элементов страницы по имени тега.

1. Начните с создания простого HTML-файла.
2. Напишите три HTML-элемента с одним тегом.
3. Добавьте в каждый из них немного контента, чтобы различать их между собой.
4. Добавьте элемент `script` в ваш HTML-файл. С его помощью отберите элементы страницы по имени тега и сохраните их в переменной в виде массива.
5. Используя значение индекса, выберите средний элемент и выведите его на экран.

Доступ к элементам по названию класса

Аналогичную работу можно проводить с помощью имени класса. В нашем примере HTML содержится два разных класса: `example` и `something`. Если вы запросите элементы по имени класса, результат вернется в виде `HTMLCollection`. Следующий код получит элементы класса `example`:

```
document.getElementsByClassName("example");
```

В консоли отобразится:

```
HTMLCollection(2) [div#one.example, div#two.example, one: div#one. example,
two: div#two.example]
```

Как видите, код вернул только теги `div` с классом `example`, `div` с классом `something` остались нетронутыми.

Практическое занятие 10.4

Выберите все подходящие элементы страницы, используя имя класса элемента.

1. Создайте простой HTML-файл.
2. Добавьте в него три HTML-элемента одного класса. Пока класс этих элементов одинаков, их теги могут быть разными. Добавьте в каждый элемент немного контента, чтобы различать их между собой.

3. Добавьте `script` в ваш HTML-файл и с его помощью выберите необходимые элементы страницы по имени класса. Присвойте итоговые значения `HTMLCollection` переменной.
4. Как и для значений массива, для выбора отдельных элементов `HTMLCollection` можно использовать значение индекса. Начиная с индекса 0, выберите один из элементов страницы по имени класса и выведите этот элемент на экран.

Доступ к элементам с помощью CSS-селектора

Получить доступ к элементам можно также с помощью селектора CSS (каскадных таблиц стилей), используя `querySelector()` и `querySelectorAll()`. Далее мы представляем CSS-селектор в качестве аргумента: это фильтрует элементы в HTML-документе и возвращает только те, которые удовлетворяют селектору.

Селектор CSS может выглядеть несколько иначе, чем вы можете себе представлять. Вместо поиска определенного макета мы применяем тот же синтаксис, как если хотим определить макет для конкретных элементов. Мы еще не обсуждали эту тему, поэтому давайте сейчас ее кратко рассмотрим.

Если `p` задан в качестве CSS-селектора, это означает, что нам требуются все элементы с тегом `p`:

```
document.querySelectorAll("p");
```

Если укажем `p.example` — значит, мы ищем элементы с тегом `p` и классом `example`. Элементы с одним и тем же тегом могут иметь и другие классы — но они будут подходить условию, пока в них присутствует класс `example`. Можно задать значение `#one` — в результате будут выбраны все значения с идентификатором `one`.

Итог вы получите тот же, как и при использовании `getElementById()`. Какой из методов взять — дело вкуса. Когда вам всего-то нужно сделать выборку по идентификатору — обсудите этот вопрос с разработчиком, с которым вы сотрудничаете. `querySelector()` позволяет выполнять более сложные запросы, но `getElementById()`, по мнению некоторых программистов, более удобочитаем. Другие будут утверждать, что согласованности ради везде стоит использовать `querySelector()`. На данном этапе это не столь важно, но все же постарайтесь быть последовательными. Пока не придавайте всему этому большого значения. Вариантов использования CSS-селекторов в JavaScript крайне много — по мере надобности вы обязательно в них разберетесь. Вот как вы можете их применить.

Использование querySelector()

Данный вариант выберет первый элемент, соответствующий запросу. Используя HTML-фрагмент, представленный в начале раздела, введите в консоль следующее:

```
document.querySelector("div");
```

В ответ вы должны увидеть такую запись:

```
<div id="one" class="example">Hi!</div>
```

Вам вернулось первое повстречавшееся значение `div`. Можно также запросить элемент с классом `.something`. Если помните, выбирая классы, мы используем точечную нотацию:

```
document.querySelector(".something");
```

Результатом будет:

```
<div id="three" class="something">Hi!</div>
```

С этим методом вы можете использовать только допустимые CSS-селекторы: элементы, классы и идентификаторы.

Практическое занятие 10.5

Используйте `querySelector()` для выбора одного элемента.

1. Создайте еще один простой HTML-файл.
2. Создайте четыре HTML-элемента, задавая всем им один и тот же класс. Если в их атрибутах прописан один и тот же класс, названия тегов могут отличаться.
3. Добавьте в каждый элемент немного контента, чтобы различать их между собой.
4. Внутри `script` используйте `querySelector()`, чтобы выбрать первый встречающийся элемент с этим классом. Сохраните его в переменной. Если результатов совпадения будет больше одного, `querySelector()` вернет только первый результат.
5. Выведите итоговый элемент на экран.

Использование querySelectorAll()

Иногда только первого экземпляра недостаточно: нужно выбрать все элементы, соответствующие запросу (например, когда вы хотите получить данные из всех полей ввода или очистить их). Это осуществимо с помощью `querySelectorAll()`:

```
document.querySelectorAll("div");
```

Результатом будет:

```
NodeList(3) [div#one.example, div#two.example, div#three.something]
```

Как видите, перед нами объект типа `NodeList`. Он содержит в себе все узлы, которые отвечают требованиям CSS-селектора. Как и в случае с `HTMLCollection`, для получения значений по индексам здесь можно применить метод `item()`.

Практическое занятие 10.6

Используйте `querySelectorAll()` для выбора всех подходящих значений в HTML-файле.

1. Создайте HTML-файл и добавьте четыре элемента HTML, задавая всем один и тот же класс.
2. Добавьте в каждый элемент немного контента, чтобы различать их между собой.
3. Внутри `script` используйте `querySelectorAll()`, чтобы выбрать все встречающиеся элементы с этим классом. Сохраните их в переменной.
4. Выведите все элементы на экран: сначала в виде массива, а потом по очереди (применяя цикл).

Обработчик щелчка кнопкой мыши на элементе

HTML-элементы, к которым подключен JavaScript, могут выполнять какие-нибудь действия, если на них щелкнуть. Перед вами фрагмент, в котором связанная с элементом функция JavaScript указана в HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <div id="one" onclick="alert('Ouch! Stop it!')">Don't click here!
    </div>
  </body>
</html>
```

Всякий раз, когда вы щелкаете на тексте в `div`, появляется всплывающее окно с надписью `Ouch! Stop it!`. Здесь JavaScript указывается непосредственно после `onclick`, но если JavaScript есть на странице, можно также сослаться на функцию, которая находится в его коде:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function stop(){
        alert("Ouch! Stop it!");
      }
    </script>
    <div id="one" onclick="stop()">Don't click here!</div>
  </body>
</html>
```

Этот код делает то же самое, что и предыдущий. Можете себе вообразить, насколько данная практика будет хороша с объемными функциями. Таким же образом HTML может ссылаться на скрипты, которые загружаются на страницу.

Обработчик щелчков кнопкой мыши можно также добавить с помощью JavaScript. Мы выбираем HTML-элемент, к которому хотим написать обработчик, и указываем свойство `onclick`.

Вот фрагмент HTML-кода:

```
<!DOCTYPE html>
<html>
  <body>
    <div id="one">Don't click here!</div>
  </body>
</html>
```

На данный момент, если щелкнуть на элементе, ничего не произойдет. Если мы хотим динамически добавить к нему обработчик щелчков, можно выбрать этот элемент и указать его свойство через консоль:

```
document.getElementById("one").onclick = function () {
alert("Auch! Stop!");
}
```

Из-за того что мы добавили ее в консоль, при обновлении страницы данная функция исчезнет.

Ключевое слово this и DOM

Ключевое слово `this` всегда имеет относительное значение, зависящее от контекста, в котором оно находится. В DOM `this` относится к тому элементу DOM, которому оно принадлежит. Если мы определим `onclick`, чтобы отправить `this` в качестве аргумента, будет отправлен элемент с `onclick`.

Перед вами фрагмент HTML-кода, содержащий JavaScript в теге `script`:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function reveal(el){
        console.log(el);
      }
    </script>
    <button onclick="reveal(this)">Click here!</button>
  </body>
</html>
```

В результате мы получим:

```
<button onclick="reveal(this)">Click here!</button>
```

Как видите, логировался тот элемент, в котором находится функция, — элемент `button`.

Можно обратиться к родителю `this` с помощью такой функции:

```
function reveal(el){
    console.log(el.parentElement);
}
```

В примере выше `body` является родителем для `button`. Поэтому, если мы нажмем кнопку с новой функцией, в выводе получим:

```
<body>
    <script>
        function reveal(el.parentElement){
            console.log(el);
        }
    </script>
    <button onclick="reveal(this)">Click here!</button>
</body>
```

Таким же образом можно отобразить любое другое свойство элемента. Например, функция `console.log(el.innerText);` вывела бы значение `innerText`, как в пункте «Изменение свойства innerText» выше.

Итак, мы узнали, что ключевое слово `this` ссылается на элемент и из этого элемента мы можем исследовать DOM. Это может быть очень полезным, например, когда нужно получить значение поля ввода. Если вы отправите `this`, то сможете прочитать и изменить свойства элемента, который вызвал функцию.

Практическое занятие 10.7

Создайте в базовом HTML-документе кнопку и добавьте атрибут `onclick`. Пример покажет вам, как можно ссылаться на объект с помощью `this`.

1. Создайте функцию для обработки щелчков кнопкой мыши в вашем коде JavaScript. Можете назвать ее `message`.
2. Добавьте ее в параметры функции `onclick`, отправляя данные текущего элемента с помощью `this`.
3. Внутри функции `message` используйте `console.dir()` для вывода на экран данных элемента, который был отправлен в функцию, с помощью `onclick` и `this`.
4. Добавьте вторую кнопку на страницу, вызывающую при нажатии эту же функцию.
5. После нажатия кнопки вы должны увидеть на экране элемент, вызвавший щелчок. Все будет выглядеть примерно так, как показано на рис. 10.1.

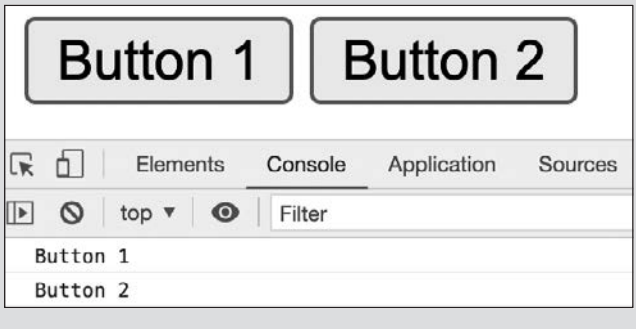


Рис. 10.1. Реализация атрибута onclick

Управление стилем элемента

После выбора нужного элемента в DOM мы можем изменить применяемый к нему стиль CSS. Это можно осуществить с помощью свойства `style`. Вот как это делается.

1. Выберите нужный элемент DOM.
2. Измените выбранный параметр свойства `style` этого элемента.

Мы собираемся сделать кнопку, которая будет управлять появлением и исчезновением строки текста. Чтобы спрятать что-либо с помощью CSS, можно установить значение `none` для свойства `display`, как это сделано ниже для элемента `p` (абзаца):

```
p {
  display: none;
}
```

Мы можем также вернуть текст назад:

```
p {
  display: block;
}
```

Данный стиль можно добавить и с помощью JavaScript. Перед вами небольшой фрагмент HTML- и JavaScript-кода, который управляет отображением части текста:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function toggleDisplay(){
        let p = document.getElementById("magic");
        if(p.style.display === "none") {
          p.style.display = "block";
        } else {
          p.style.display = "none";
        }
      }
    </script>
    <p id="magic">I might disappear and appear.</p>
    <button onclick="toggleDisplay()">Magic!</button>
  </body>
</html>
```

Как видите, в операторе `if` мы проверяем, скрыт ли текст в данный момент. Если скрыт, отображаем его. В обратном случае — прячем. Если вы нажмете кнопку, а текст в данный момент виден, он исчезнет. Если вы нажмете кнопку, а текст не виден, он появится.

С помощью этого свойства `style` можно делать всякие забавные вещи. Как думаете, что произойдет, когда вы нажмете на кнопку?

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function rainbowify(){
        let divs = document.getElementsByTagName("div");
        for(let i = 0; i < divs.length; i++) {
          divs[i].style.backgroundColor = divs[i].id;
        }
      }
    </script>
    <style>
      div {
        height: 30px;
        width: 30px;
        background-color: white;
      }
    </style>
    <div id="red"></div>
    <div id="orange"></div>
    <div id="yellow"></div>
    <div id="green"></div>
    <div id="blue"></div>
    <div id="indigo"></div>
    <div id="violet"></div>
    <button onclick="rainbowify()">Make me a rainbow</button>
  </body>
</html>
```

Вот что вы увидите, впервые открыв страницу (рис. 10.2).

И нажав кнопку (рис. 10.3).

Рис. 10.2. Кнопка, которая будет делать удивительные вещи, когда ее нажмут

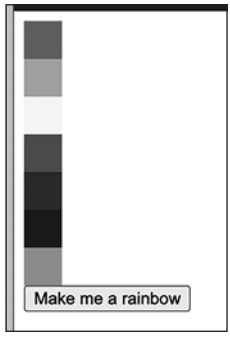


Рис. 10.3. Прекрасная радуга, созданная JavaScript нажатием кнопки

Рассмотрим скрипт, чтобы понять, как он работает. Прежде всего, в HTML-коде есть несколько тегов `div`, у каждого из которых — свой ID для определенного цвета. Прописан `style`, который задает следующие свойства по умолчанию для тегов `div`: размер — 30 на 30 пикселей, фон — белый.

При нажатии на кнопку выполняется JavaScript-функция `rainbowify()`. В ней происходят следующие процессы.

1. Все элементы `div` выбираются и сохраняются в массив `divs`.
2. Цикл перебирает массив `divs`.
3. Идентификатору элемента присваивается свойство `backgroundColor` — это проделывается для каждого элемента массива `divs`. Поскольку все идентификаторы представляют какой-то цвет, мы видим, как появляется радуга.

Представьте, сколько удовольствия можно получить, играя с этим кодом. С помощью всего нескольких строк вы можете заставить разные предметы появиться на экране.

Изменение классов элементов

HTML-элементы могут иметь классы — как мы видели, эти элементы можно выбрать как раз по имени их классов. Напомним: классы часто используются для создания определенного оформления элементов с помощью CSS.

Благодаря JavaScript можно изменять классы HTML-элементов и таким образом вызывать определенный макет оформления, связанный с этим классом в CSS. Далее мы рассмотрим добавление классов, их удаление и переключение.

Добавление классов в элементы

Тема может показаться немного мутной, поэтому рассмотрим пример. В нем мы собираемся написать класс к элементу, добавляющий оформление и заставляющий элемент исчезнуть.

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function disappear(){
        document.getElementById("shape").classList.add("hide");
      }
    </script>
    <style>
      .hide {
        display: none;
      }
```

```
      .square {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }

      .square.blue {
        background-color: blue;
      }
    </style>
    <div id="shape" class="square blue"></div>

    <button onclick="disappear()">Disappear!</button>
  </body>
</html>
```

Итак, у нас есть CSS, описанный в теге `style`. Для `display` в элементах с классом `hide` указан стиль `none`, это значит, что данные элементы скрыты. Элементы с классом `square` желтые, и их размер равен 100 на 100 пикселей. Но если в них прописаны классы `square` и `blue`, эти элементы будут голубыми.

Функция `disappear()` определена в `script` и вызывается нажатием кнопки Disappear!. Она изменяет класс, получая свойство `classList` с идентификатором `shape` (как видим, форма у нас задана квадратная). Добавляем класс `hide` к `classList`: элементы получают шаблон `display: none` — и мы их больше не видим.

Удаление классов из элементов

Классы можно также удалить. Например, если убрать класс `hide` из `classList`, мы снова увидим все элементы, потому что `display: none` больше не будет применяться.

В примере ниже мы удаляем другой класс. Можете, просто взглянув на код, понять, что произойдет при нажатии кнопки?

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function change(){
        document.getElementById("shape").classList.remove("blue");
      }
    </script>
    <style>
      .square {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }
```

```
      .square.blue {
        background-color: blue;
      }
    </style>
    <div id="shape" class="square blue"></div>

    <button onclick="change()">Change!</button>
  </body>
</html>
```

При нажатии кнопки вызывается функция, которая удаляет класс `blue`: из шаблона убирается фоновое значение голубого цвета — следовательно, квадрат становится желтым.

У вас может появиться вопрос: почему квадрат изначально был голубым, хотя ему было присвоено два CSS-значения `background-color`? Подобное происходит из-за наличия системы баллов. Чем более точно определен стиль, тем больше у него баллов. Например, записав два класса без пробела между ними, вы говорите, что конкретные свойства применяются к элементам с этими двумя классами, это более четко, чем указывать на один класс.

Ссылка на ID в CSS (#nameId) помогает набрать еще больше баллов и ставит подобный макет выше другого, оформленного на основе классов. Такая иерархия позволяет меньше дублировать код, но может привести к беспорядку, поэтому всегда старайтесь аккуратно комбинировать CSS и HTML, чтобы получить желаемый вид страницы.

Переключение классов

В некоторых случаях вы захотите добавить класс, если его еще в элементе нет, или удалить его, если такой там уже существует. Это называется переключением. Для переключения классов есть особый метод. Изменим наш первый пример и переключим класс `hide` таким образом, чтобы он появился со вторым нажатием кнопки, исчез с третьим и т. д. (Мы удалили класс `blue` ради сокращения кода — он все равно ничего в этом примере не делал, кроме заливки квадрата голубым цветом.)

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function changeVisibility(){
        document.getElementById("shape").classList.toggle("hide");
      }
    </script>
    <style>
      .hide {
```

```
        display: none;
      }
      .square {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }
    </style>
    <div id="shape" class="square"></div>

    <button onclick="changeVisibility()">Magic!</button>
  </body>
</html>
```

После нажатия кнопки Magic! в `classList` будет добавлен класс `hide`, если его в списке еще нет, и удален, если он там есть. Это дает вам возможность отслеживать результат каждого нажатия кнопки. Квадрат будет продолжать появляться и исчезать.

Управление атрибутами

Мы уже знаем, как изменять атрибуты стиля и класса, но для этого есть и более общий метод. Краткое напоминание: атрибуты — это части HTML-элементов, за которыми следуют знаки равенства. Например, данный HTML-код дает ссылку на поисковую страницу Google:

```
<a id="friend" class="fancy boxed" href="https://www.google.com">Ask my
friend here.</a>
```

Атрибуты в примере — `id`, `class` и `href`. Вам также уже встречались другие общие атрибуты — `src` и `style`, но существует и множество других.

Добавлять или изменять атрибуты элемента можно с помощью метода `setAttribute()`. Данный метод изменит HTML-код страницы. Если вы просмотрите HTML-код в браузере, вы легко обнаружите эти измененные атрибуты. С помощью данного метода можно вводить изменения прямо в консоли или написать другой HTML-файл. В следующем HTML-фрагменте это наглядно видно:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function changeAttr(){
        let el = document.getElementById("shape");
        el.setAttribute("style", "background-color:red;border:1px solid black");
        el.setAttribute("id", "new");
        el.setAttribute("class", "circle");
```

```
      }
    </script>
    <style>
      div {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }
      .circle {
        border-radius: 50%;
      }
    </style>
    <div id="shape" class="square"></div>

    <button onclick="changeAttr()">Change attributes...</button>
  </body>
</html>
```

Страница перед нажатием кнопки (рис. 10.4).

После нажатия кнопки код, описывающий `div`, принимает следующий вид:

```
<div id="new" class="circle" style="background-color:red;border:1px
solid black"></div>
```

Как видите, атрибуты изменились: `id` поменял значение с `shape` на `new`, значение `class` со `square` стало `circle`, а еще был добавлен `style`. Результат будет выглядеть так, как показано на рис. 10.5.

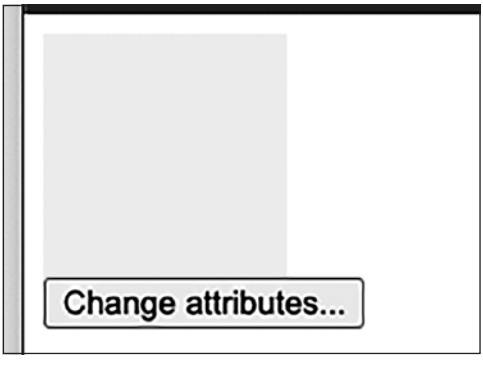


Рис. 10.4. Страница с желтым квадратом div

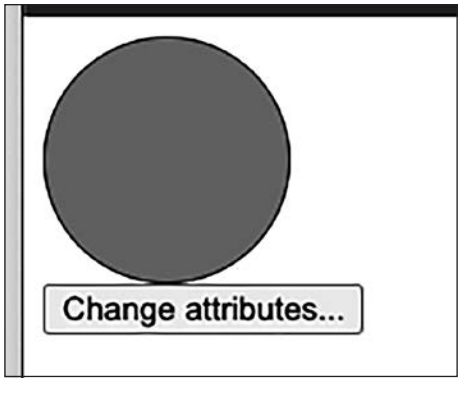


Рис. 10.5. Страница с красным кругом, заключенным в черную окружность

Это очень мощный инструмент, который очень по-разному можно использовать для взаимодействия с DOM. Представьте себе: с помощью этого инструмента

можно создавать изображения и даже открытки. На самом деле здесь может происходить много всего интересного.

Важно отметить, что JavaScript взаимодействует с DOM, а не с файлом HTML, следовательно, изменяется именно DOM. Если вы нажмете кнопку еще раз, в консоли появится сообщение об ошибке, потому что элемент с id="shape" в DOM не найден, следовательно, мы пытаемся вызвать метод со значением null.

Практическое занятие 10.8

В данном упражнении мы будем создавать пользовательские атрибуты. Используя данные из массива имен, программа обновит код элемента. Элементы внутри массива будут выведены на страницу в виде HTML-кода (рис. 10.6). Пользователь сможет щелкнуть на элементе — данное действие отобразит значения его атрибутов.

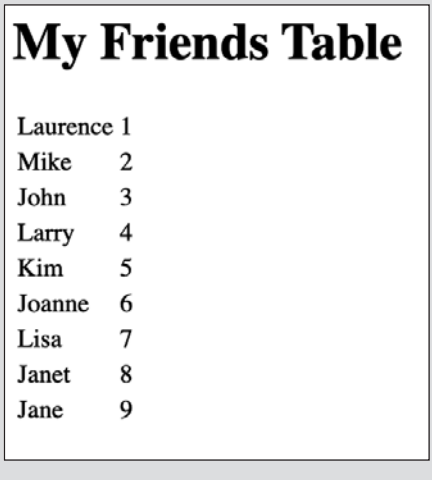


Рис. 10.6. Создание пользовательских атрибутов с массивом имен

Тем временем ваш уровень знаний в JavaScript продолжает расти, а программы в упражнениях становятся все длиннее и сложнее — так что отныне мы будем предоставлять HTML-шаблоны только в тех заданиях, где это действительно необходимо. Можете использовать следующую структуру и в качестве решения задачи дополнить ее содержимым элемента `script`:

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
    <div id="message"></div>
    <div id="output"></div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Создайте массив имен. Имен может быть сколько угодно — все строковые значения будут выведены на страницу в формате таблицы.
2. Выберите элемент страницы как объект JavaScript.
3. Добавьте функцию и вызовите ее в коде JavaScript. Функцию можно назвать `build()`, потому что она будет строить содержимое страницы. В `build` вы будете настраивать HTML-код в таблице.
4. Создайте таблицу с названием `html`. Внутри тегов перебирайте содержимое массива и выводите результаты в таблицу.
5. Добавьте класс `box` в одну из ячеек с индексом элемента из массива. Повторите данное действие для элементов каждой дополнительной строки.
6. После того как вы пропишете HTML-код элементов внутри `tr`, в главном элементе `row` создайте атрибут `data-row`, содержащий индекс элемента в массиве. Кроме того, добавьте в элемент еще один атрибут с именем `data-name`, который будет содержать выводимый текст.
7. Внутри атрибута того же элемента `tr` добавьте `onclick`, чтобы вызвать функцию `getData`, передающую текущий элемент объекта как `this` в параметр функции.
8. Добавьте таблицу HTML-кода на страницу.
9. Создайте функцию `getData`, которая будет вызываться в момент нажатия на `tr`. Как только на `tr` нажмут, используйте `getAttribute`, чтобы получить атрибуты значений строки и содержимое текстового вывода и сохранить их в разных переменных.
10. Используя значения сохраненных ранее атрибутов, выведите их `message` на странице.
11. Как только пользователь нажмет на элемент, на странице отобразятся сведения, полученные из атрибутов внутри элемента с `id`, равным `message`.

Прослушиватели событий элементов

События — это то, что происходит на веб-странице: щелчок на чем-либо, наведение указателя мыши на элемент, изменение элемента и многое другое. Мы уже рассмотрели, как добавлять обработчик `onclick`. Таким же способом можно добавлять обработчики `onchange` или `onmouseover`. Однако существует особое условие: один элемент может содержать в качестве атрибута только один обработчик. Если у элемента есть `onclick`, у него уже не может быть `onmouseover`.

На данный момент мы знаем только то, как добавлять прослушиватели событий, используя подобные этому HTML-атрибуты:

```
<button onclick="addRandomNumber()">Add a number</button>
```

Можно также зарегистрировать обработчики событий с помощью JavaScript. Такие объекты носят название *«прослушиватели событий»* (event listeners). Благодаря им мы можем добавить множество событий для одного элемента — JavaScript будет постоянно проверять (или прослушивать) эти события для конкретных элементов на странице. Добавление прослушивателя событий — двухэтапный процесс.

1. Выберите элемент, для которого хотите добавить событие.
2. Используйте следующий синтаксис: `addEventListener("event", function)`

Несмотря на то что процесс состоит из двух этапов, выполнить его можно в одной строке кода:

```
document.getElementById("square").addEventListener("click",
changeColor);
```

Данная строка получает элемент с идентификатором `square` и добавляет событие `changeColor`, которое имеет место при каждом нажатии на этот элемент. Обратите внимание: при использовании прослушивателя событий мы удаляем из события префикс `on`. Например, `click` ссылается на тот же тип события, что и `onclick`, но префикс `on` мы удалили.

Рассмотрим другой способ добавления прослушивателя: установку свойства `event` определенного объекта в функцию. (Не волнуйтесь, мы вернемся ко всем этим методам в главе 11 и изучим их более подробно).

Забавный факт: прослушиватели событий часто добавляются во время других событий!

Можно было бы здесь повторно использовать надежный прослушиватель `onclick`, но мы решили выбрать другой распространенный вариант — загрузить веб-страницу с помощью `onload`:

```
window.onload = function() {
  // все, что должно произойти после загрузки:
  // например, добавление прослушивателя
}
```

Далее выполняется прописанная функция. Подобная схема действий характерна для `window.onload`, но менее распространена, например `onclick` для `div` (хотя и возможна). Изучим первый рассмотренный нами прослушиватель событий. По-вашему, что он будет делать, когда вы нажмете на квадрат?

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      window.onload = function() {
        document.getElementById("square").addEventListener("click", changeColor);
      }
      function changeColor(){
        let red = Math.floor(Math.random() * 256);
        let green = Math.floor(Math.random() * 256);
        let blue = Math.floor(Math.random() * 256);
        this.style.backgroundColor = `rgb(${red}, ${green}, ${blue})`;
      }
    </script>
    <div id="square" style="width:100px;height:100px;background-color:
        grey;">Click for magic</div>
  </body>
</html>
```

На экране появляется серый квадрат с надписью `Click for magic`. После завершения загрузки веб-страницы для этого квадрата добавляется событие. Отныне всякий раз, когда на нем щелкают кнопкой мыши, выполняется функция `changeColor`. Для изменения цвета она использует произвольные переменные из модели RGB. Таким образом, когда вы нажимаете на квадрат, цвет обновляется и принимает случайное значение.

Вы можете добавлять события ко всем видам элементов. До сих пор мы использовали только событие `click`, но их гораздо больше: например, `focus`, `blur`, `focusin`, `focusout`, `mouseout`, `mouseover`, `keydown`, `keypress` и `keyup`. Мы поговорим о них в следующей главе, так что продолжайте!

Практическое занятие 10.9

Попробуйте альтернативный способ реализации логики из практического занятия 10.7. В качестве шаблона используйте следующий HTML-код, добавив туда содержимое `script`:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <div>
        <button>Button 1</button>
        <button>Button 2</button>
        <button>Button 3</button>
    </div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие действия.

1. Выберите все кнопки и поместите в объект JavaScript.
2. Переберите каждую кнопку и создайте функцию `output` в области действия кнопки.
3. Внутри функции `output()` создайте метод `console.log()`, который будет выводить текущий `textContent` объекта. Можете ссылаться на текущий родительский объект, используя ключевое слово `this`.
4. Как только вы переберете кнопки, добавьте прослушиватель событий, который при нажатии на кнопку вызывает функцию `output()`.

Создание новых элементов

В этой главе вы видели множество интересных способов управления DOM. Но мы все еще не изучили один важный момент: создание новых элементов и добавление их в DOM. Следующий JavaScript-код делает именно это — как видите, все не так сложно, как кажется:

```
let el = document.createElement("p");
el.innerText = Math.floor(Math.random() * 100);
document.body.appendChild(el);
```

Итак, сначала мы создаем элемент типа p (абзац) с помощью функции createElement() в объекте document. В процессе необходимо указать тип желаемого HTML-элемента (в данном случае p), что-то вроде:

```
<p>innertext here</p>
```

innerText добавляет случайное число. Следующий этап — элемент назначается новым конечным дочерним элементом body. Вы также можете добавить его в другой элемент. Просто выберите элемент, куда вам нужно поместить новый, и используйте метод appendChild().

Итак, наш элемент включен в HTML страницы. Там есть кнопка — всякий раз, когда ее нажимают, добавляется p.

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function addRandomNumber(){
        let el = document.createElement("p");
        el.innerText = Math.floor(Math.random() * 100);
        document.body.appendChild(el);
      }
    </script>
    <button onclick="addRandomNumber()">Add a number</button>
  </body>
</html>
```

Вот как выглядит страница после того, как на кнопку нажали пять раз (рис. 10.7).

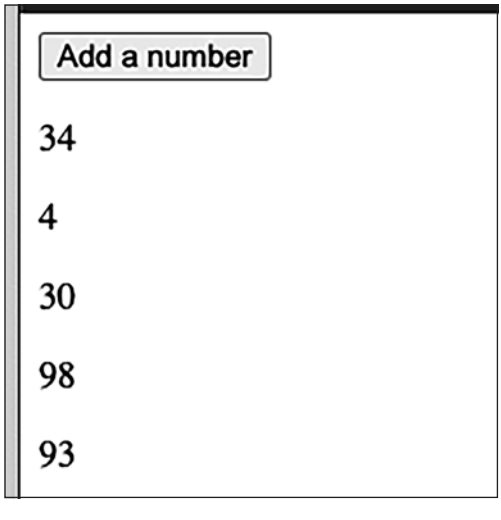


Рис. 10.7. Случайные числа после пятикратного нажатия кнопки

Как только мы обновим страницу, поле опустеет. Файл с исходным кодом не меняется, и мы его нигде не храним.

Практическое занятие 10.10

В данном упражнении мы создадим список покупок и, используя предложенный HTML-шаблон, обновим код, чтобы добавить туда новые элементы — они появятся в списке при нажатии кнопки:

```
<!DOCTYPE html>
<html>
<head>
    <title>Complete JavaScript Course</title>
    <style>
    </style>
</head>
<body>
    <div id="message">Complete JavaScript Course</div>
    <div>
        <input type="text" id="addItem">
        <input type="button" id="addNew" value="Add to List"> </div>
    <div id="output">
        <h1>Shopping List</h1>
        <ol id="sList"> </ol>
    </div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Выберите элементы страницы как объекты JavaScript.
2. Создайте прослушиватель событий `onclick` для кнопки добавления в список. После нажатия кнопки содержимое ввода должно отобразиться в конце списка. Для реализации этого можно вызвать функцию `addOne()`.
3. В `addOne()` создайте элементы `li` для включения в основной список на странице. Добавьте значение ввода к текстовому содержимому.
4. В функции `addOne()` получите текущее значение ввода `addItem`. Используйте его, чтобы создать `textNode` с этим значением, добавляя его в список. Включите `textNode` в список.

Проекты текущей главы

Сворачиваемый компонент-аккордеон

Создайте сворачивающийся и разворачивающийся компонент-аккордеон, который будет открывать элементы страницы, а также скрывать и показывать содержимое раздела при нажатии на вкладку заголовка. Используя следующий HTML-код в качестве шаблона, добавьте содержимое `script` и создайте желаемую функциональность с помощью JavaScript:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
    <style>
        .active {
            display: block !important;
        }
        .myText {
            display: none;
        }
        .title {
            font-size: 1.5em;
            background-color: #ddd;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="title">Title #1</div>
        <div class="myText">Just some text #1</div>
        <div class="title">Title #2</div>
        <div class="myText">Just some text #2</div>
        <div class="title">Title #3</div>
        <div class="myText">Just some text #3</div>
    </div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Выберите все элементы класса `title` с помощью `querySelectorAll()`.
2. Выберите все элементы класса `myText` с помощью `querySelectorAll()`. Их количество должно совпадать с количеством элементов `title`.

3. Выполните итерацию по всем элементам `title` и добавьте прослушиватели событий, которые по щелчку будут искать следующие такие же элементы.
4. Выберите элемент для действия `click` и переключите объект `classlist` этого элемента на `active`. Это даст возможность пользователю по щелчку на элементе скрывать или показывать содержимое раздела.
5. Добавьте функцию, которая будет вызываться при каждом нажатии на элементы, это удалит класс `active` для всех элементов и скроет все элементы с `myText`.

Интерактивная система голосования

Приведенный ниже код создаст динамический список людей. В нем можно будет нажимать на конкретные имена и обновлять соответствующие значения, указывающие количество нажатий на имя. На странице также будет поле ввода, которое позволит добавить в список еще больше пользователей: для каждого из них будет создан новый элемент в списке, с которым можно будет взаимодействовать так же, как и с элементами по умолчанию (рис. 10.8).

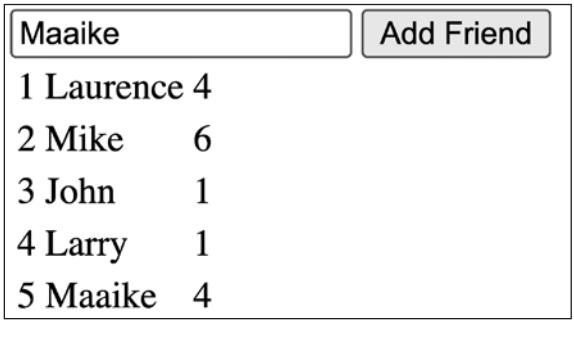


Рис. 10.8. Создание интерактивной системы голосования

В качестве шаблона используйте следующий HTML-код, добавив туда содержимое `script`.

```
<!DOCTYPE html>
<html>
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
    <div id="message">Complete JavaScript Course</div>
    <div>
        <input type="text" id="addFriend">
        <input type="button" id="addNew" value="Add Friend">
    </div>
    <table id="output"></table>
    <script>
```

```
    </script>
</body>
</html>
```

Выполните следующие действия.

1. Создайте массив имен людей и назовите его `myArray`. Это будет список имен по умолчанию.
2. Выберите элементы страницы в качестве объектов JavaScript, чтобы с ними можно было легко взаимодействовать в коде.
3. Добавьте прослушиватель событий для кнопки Add Friend. После ее нажатия он будет получать значение из поля ввода и передавать его функции, обновляющей список друзей на странице. Дополнительно: добавьте имя нового друга в созданный вами массив. Получите текущее значение в поле ввода и вставьте его в массив так, чтобы массив соответствовал значениям на странице.
4. Запустите функцию для формирования содержимого страницы, используя цикл `forEach()`, чтобы получить все элементы в массиве и добавить их на страницу. Присвойте начальное значение `0` счетчику голосов всех участников.
5. Создайте основную функцию, которая будет формировать элементы страницы, начиная со строки родительской таблицы, `tr`. Затем напишите три элемента `td` — ячейки таблицы. Добавьте туда содержимое. Подсчет голосов должен вестись в последнем столбце; имя человека следует указать в середине, а индекс плюс 1 — в первом столбце.
6. Добавьте ячейки в строку таблицы, а строку — в область вывода на странице.
7. Добавьте прослушиватель событий, который увеличит значение счетчика голосов конкретной строки, когда пользователь щелкнет на ней.
8. Получите текстовое содержимое из последнего столбца в строке: здесь должно храниться текущее значение счетчика (убедитесь, что тип данных счетчика — число, чтобы можно было его увеличивать на единицу). Увеличьте значение счетчика.
9. Обновите последний столбец.

Игра «Виселица»

Создайте игру «Виселица», используя массивы и элементы страницы. Можете взять следующий HTML-шаблон:

```
<!doctype html>
<html><head>
    <title>Hangman Game</title>
    <style>
        .gameArea {
            text-align: center;
```

```
            font-size: 2em;
        }
        .box,
        .boxD {
            display: inline-block;
            padding: 5px;
        }
        .boxE {
            display: inline-block;
            width: 40px;
            border: 1px solid #ccc;
            border-radius: 5px;
            font-size: 1.5em;
        }
    </style>
</head>
<body>
    <div class="gameArea">
        <div class="score"> </div>
        <div class="puzzle"></div>
        <div class="letters"></div>
        <button>Start Game</button>
    </div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Задайте массив, содержащий некоторые слова или фразы, которые вы хотите использовать в игре.
2. Создайте в JavaScript основной игровой объект со свойством, включающим текущее решение, и другим свойством, содержащим массив букв решения. Вам также необходим массив элементов страницы, соответствующих индексам каждой буквы решения, и, наконец, свойство, подсчитывающее количество букв, оставшихся для разгадки решения и, если необходимо, завершения игры.
3. Поместите все элементы страницы в переменные, чтобы к ним было легче получить доступ в коде.
4. Добавьте к кнопке Start Game прослушиватель событий, запускающий при ее нажатии функцию `startGame()`.
5. Внутри функции `startGame()` необходимо организовать проверку массива `words` на наличие оставшихся в нем слов. Если слова остались, скройте кнопку, установив для объекта `.display` значение `none`. Очистите содержимое игры и установите общий результат равным `0`. Внутри текущего слова в игровом объекте задайте значение, которое должно быть ответом `shift()` из массива, содержащего внутриигровые слова.

6. В решении с помощью `split()` преобразуйте строку в массив всех символов слова.
7. Создайте функцию `builder()`, которая будет использоваться для построения игрового поля. Вызовите функцию в функции `startGame()`, как только все игровые значения будут очищены и заданы.
8. Напишите отдельную функцию, которую вы сможете использовать для создания элементов страницы. В параметрах укажите тип нового элемента, родительский элемент, к которому он будет добавлен, выходное содержимое и класс, которые будут ему присвоены. Используя временную переменную, создайте элемент, присвойте ему класс, добавьте элемент к родительскому и установите значение `textContent`. Верните элемент.
9. В функции `builder()`, которая будет вызвана сразу после запуска `startGame()`, очистите `innerHTML` от букв и элементов игровой страницы.
10. Выполните итерацию по массиву решений, получая каждую букву решения. Примените `builder()` для создания элементов страницы. Добавьте выходное значение `-`, задайте класс и добавьте его к основному элементу страницы.
11. Проверьте, является ли значение пустым. Если да, очистите `textContent` и поменяйте цвет границы блока буквы на белый. Если нет, увеличьте значение так, чтобы оно отражало количество букв, которые необходимо угадать. Вставьте новый элемент в игровой массив.
12. Создайте функцию для обновления результата, чтобы вы могли выводить текущее количество оставшихся букв. Добавьте ее к функции `builder()`.
13. Создайте цикл для перебора 33 букв алфавита. Вы можете генерировать буквы, используя массив алфавита. Строковый метод `fromCharCode()` вернет символ из числового представления.
14. Создайте элемент для каждой буквы, добавляя `class` из `box`; добавьте его к элементу страницы `letters`. По мере создания каждого элемента напишите прослушиватель событий, запускающий функцию `checker()`.
15. Как только будет нажата необходимая буква, нужно вызвать функцию `checker()`, которая удалит основной класс, добавит другой класс, удалит прослушиватель событий и обновит цвет фона. После этого вызовите функцию `checkLetter()`, передающую значение выбранной буквы в аргумент.
16. Функция `checkLetter()` переберет все буквы решения. Добавьте условие, чтобы проверить, есть ли выбранная игроком буква в заданном слове. Обязательно преобразуйте введенную букву в верхний регистр, чтобы регистры букв совпадали и сравнение проходило аккуратно. Обновите подходящие буквы, используя игровой массив и индекс буквы в решении. Значения индексов для всех букв будут одинаковыми, что обеспечит простоту сопоставления визуального представления с представлением в массиве.
17. Вычтите единицу из глобального объекта игры, который отслеживает общее количество букв, оставшихся для разгадки задания, вызовите функцию

updatescore(), чтобы проверить, закончена ли игра, и обновите общий счет. Установите textContent для буквы, удалив исходную черточку.

18. В функции updatescore() установите итоговое значение равным числу оставшихся букв. Если общее оставшееся количество меньше или равно нулю — игра окончена. Отобразите кнопку, чтобы у игрока была возможность выбрать следующую фразу.

Вопросы для самопроверки

1. Каким будет результат выполнения следующего кода?

```
<div id="output">Complete JavaScript Course </div>
<script>
    var output = document.getElementById('output');
    output.innerText = "Hello <br> World";
</script>
```

2. Какие выходные данные отобразятся на странице браузера?

```
<div id="output">Complete JavaScript Course </div>
<script>
    document.getElementById('output').innerHTML = "Hello<br> World";
</script>
```

3. Что будет представлено в поле ввода?

```
<div id="output">Hello World</div>
<input type="text" id="val" value="JavaScript">
<script>
    document.getElementById('val').value = document.
getElementById('output').innerHTML;
</script>
```

4. Взгляните на следующий код. Что будет выведено на экран при щелчке на элементе со словом three? Что будет выведено на экран при щелчке на элементе со словом one?

```
<div class="holder">
    <div onclick="output('three')">Three
        <div onclick="output('two')">Two
            <div onclick="output('one')">One</div>
            </div>
        </div>
    </div>
<script>
    function output(val) {
        console.log(val);
    }
</script>
```

5. Какую строку необходимо дописать, чтобы удалить прослушиватель событий при нажатии кнопки в следующем коде?

```
<div class="btn">Click Me</div>
<script>
    const btn = document.querySelector(".btn");
    btn.addEventListener("click", myFun);
    function myFun() {
        console.log("clicked");

    }
</script>
```

Резюме

В этой главе ваши навыки веб-программирования действительно поднялись на новый уровень. Если вы умеете управлять DOM, вы можете делать на веб-странице что угодно, таким образом, она больше не является статической.

Сначала вы познакомились с динамическими страницами и узнали, как искать элементы в DOM. Пройдясь по элементам вручную, мы открыли, что есть более простой способ выбора элементов в DOM: с помощью методов `getElementBy...()` и `querySelector()`. Выбрав элементы, мы получили возможность изменять их, добавлять к ним новые элементы, а также выполнять с их помощью всевозможные действия. Мы взяли некоторые простейшие обработчики (например, `onclick`) и назначали им функции. Используя аргумент `this`, который был отправлен в качестве параметра, мы получили доступ к элементу, на котором щелкнули кнопкой мыши. Его можно модифицировать различными способами: например, изменив свойство `style`. Мы рассмотрели, как добавлять классы к конкретному элементу, создавать новые элементы и добавлять их в DOM. И наконец, мы работали с прослушивателями событий, которые действительно сделали наши динамические веб-страницы куда более продвинутыми, ведь с их помощью мы можем указать более одного обработчика событий для выбранного элемента. Все эти навыки позволяют делать в веб-браузере удивительные вещи. Вы теперь даже можете проектировать полноценные игры!

Следующая глава еще больше расширит ваши возможности в создании интерактивных веб-страниц (а также сделает этот процесс немного проще).

11 Интерактивный контент и прослушиватели событий

Вы уже знакомы с основами управления *объектной моделью документа (DOM)*. В предыдущей главе мы столкнулись с различными событиями и внедряли прослушиватель, чтобы отследить, имеют ли они место в данный момент. Если событие происходило, вызывалось указанное действие (функция).

Теперь мы собираемся сделать еще один шаг вперед в этой области и использовать прослушиватели событий для создания интерактивного веб-контента. Эта глава действительно усовершенствует ваши знания о DOM. В ее рамках мы рассмотрим следующие темы:

- интерактивный контент;
- указание событий;
- обработчик событий `onload`;
- обработчик действий мышью;
- свойство события `target`;
- поток событий DOM;
- `onchange` и `onblur`;
- обработчик событий клавиатуры;
- перетаскиваемые элементы;
- отправку форм;
- анимацию элементов.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Введение в интерактивный контент

Интерактивный контент — это содержимое, отвечающее на действия пользователя. Представьте, например, веб-приложение, в котором вы можете в режиме реального времени создавать открытки или играть в игру на сайте.

Интерактивность становится возможной благодаря изменениям в DOM, происходящим в ответ на действия пользователя. Эти действия могут быть любыми: ввод текста в поле, щелчок кнопкой мыши или наведение указателя мыши на определенный элемент, ввод с клавиатуры. Все это и есть события. Мы уже встречались с ними — но в этой теме есть еще столько всего для изучения!

Указание событий

Существует три способа указания событий. Мы рассмотрели их в предыдущей главе, поэтому давайте просто повторим их еще раз. Один из них основан на HTML, а два других — на JavaScript. В качестве примера возьмем событие `click`.

С помощью HTML

Сначала рассмотрим способ с HTML:

```
<p id="unique" onclick="magic()">Click here for magic!</p>
```

Самое замечательное в указании событий таким образом — довольно легкая читаемость и предсказуемость кода: как только вы нажмете на абзац, сработает функция `magic()`. Но есть и недостатки: таким способом можно указать только одно событие и его нельзя будет позже динамически изменить.

С помощью JavaScript

Вот как можно осуществить это с помощью JavaScript.

```
document.getElementById("unique").onclick = function() { magic(); };
```

Здесь мы получаем свойство, представляющее выбранное событие, и присваиваем ему нашу функцию. Выбираем `p`, как показано в предыдущем разделе, по значению атрибута `unique`, захватываем свойство `onclick` и назначаем ему функцию `magic()`, обернув ее в анонимную функцию. Мы также могли бы указать точную функцию в этой точке. Данную операцию можно в любой момент перезаписать другой функцией, сделав запускаемое событие более динамичным.

Теперь мы можем указать несколько разных событий, чего не получилось бы сделать с помощью HTML. Так, можно добавить события `keyup`, `keydown` или `mouseover` — мы рассмотрим данные типы событий в текущей главе.

Если вам нужно определить триггеры событий для всех элементов страницы, укажите их в цикле, чтобы соблюсти принципы чистого кода.

Практическое занятие 11.1

Персонализируйте свои веб-страницы. Дайте пользователям возможность изменять тему страницы со светлой на темную.

1. В базовом HTML-документе укажите переменную с логическим значением, которая будет переключать цветовые режимы.
2. Используйте `window.onclick`, чтобы настроить функцию, которая при нажатии выводит сообщение на экран. Можете использовать значение логической переменной.
3. Внутри функции добавьте условие, которое проверяет, является ли переменная `darkMode` истинной или ложной.
4. Если значение переменной `false`, измените стиль страницы, определив ей черный фон и белый цвет шрифта.
5. Добавьте отклик `else`, который сменит цвет фона на белый, а цвет текста — на черный. Соответствующим образом измените значение переменной `darkMode`.

С помощью прослушивателей событий

Второй JavaScript-способ указания событий — использование метода `addEventListener()`, добавляющего событие для элемента. С его помощью для одного и того же события (например, щелчка на элементе) можно указать несколько функций.

Стоит отметить, что при указании событий с помощью HTML и первого JavaScript-способа события получают префикс `on`: `onclick`, `onload`, `onfocus`, `onblur`, `onchange` и т. д. Когда мы применяем метод `addEventListener()`, мы указываем тип элемента внутри прослушивателя событий *без* префикса `on`. Например, как в случае с альтернативой `onclick`:

```
document.getElementById("unique").addEventListener("click", magic);
```

Обратите внимание, что здесь круглые скобки за функцией `magic` опущены. Мы не можем отправлять такие параметры. Если вам все же необходимо это сделать, придется обернуть функционал в анонимную функцию:

```
document.getElementById("unique").addEventListener("click", function()
{ magic(arg1, arg2) });
```

В примерах этой главы можно использовать любой из перечисленных способов указания события — но мы в большинстве своем будем обращаться к одному из JavaScript-вариантов.

Практическое занятие 11.2

Создайте в `textContent` несколько элементов `div`, содержащих имена цветов. Напишите код JavaScript, чтобы добавить прослушиватель событий `click` к каждому элементу, и по мере нажатия на каждый элемент обновляйте цвет фона `body`, чтобы он соответствовал названию цвета в `div`.

Обработчик событий onload

Событие `onload` запускается сразу же после загрузки определенного элемента. Это может быть полезным в нескольких случаях. Например, если вам требуется выбрать элемент с помощью `getElementById`, вы должны быть уверены, что этот элемент уже загружен в DOM. Данное событие применятся в основном к объекту `window`, но на самом деле актуально для любого элемента. В случае `window` событие запускается, когда объект `window` завершает загрузку. Вот как это выглядит:

```
window.onload = function() {
    // все, что должно произойти после загрузки страницы, помещается сюда
}
```

Событие `onload` в случае объектов `window` и `document` работает схожим образом, но все же есть определенные отличия. Разница заключается в используемом браузере. Событие `load` начинается в конце загрузки объекта `document` — таким образом, вы заметите, что все объекты в документе находятся в DOM, а ресурсы завершили загрузку.

Чтобы обработать любое событие, можно также использовать метод `addEventListener()` — он применим и для действий, начинающихся после загрузки всего содержимого в DOM. Для этого существует также особый встроенный `DOMContentLoaded()`. Его можно использовать для обработки события загрузки DOM.

Если `DOMContentLoaded()` задан, то будет выполнен сразу после создания DOM для страницы. Вот как его применить:

```
document.addEventListener("DOMContentLoaded", (e) => {
    console.log(e);
});
```

Информация отобразится на экране, когда все содержимое DOM будет загружено. Как вариант (с которым вы будете сталкиваться довольно часто), можно прописать событие в теге `body`, например, так:

```
<body onload="unique()"></body>
```

Данный код назначает для `body` функцию `unique()`, которая отключится в момент прекращения загрузки. Совместить `addEventListener()` и HTML не получится: один вариант будет переписывать другой в зависимости от порядка их расположения на веб-странице. Если все же нужно, чтобы при загрузке DOM произошло два события, понадобятся два вызова `addEventListener()` в коде JavaScript.

Практическое занятие 11.3

Следующее упражнение на примере базового HTML-файла продемонстрирует порядок загрузки объекта `window` и объекта `document` с помощью события `DOMContentLoaded`. Это событие активируется, как только содержимое объекта `document` будет загружено в браузер. Объект `window` загрузится позже, даже если сначала выполнится оператор `window.onload`.

1. В базовом HTML-файле создайте функцию `message`, для которой требуется два параметра: первый — строковое значение для сообщения, а второй — объект события. Внутри функции выведите на экран данное сообщение и события с помощью `console.log`.
2. Используя объект `window`, добавьте объекту события функцию `onload`. Вызовите функцию, передав строковое значение `Window Ready` и объект события в функцию `message` для вывода результата.
3. Создайте вторую функцию для фиксации загрузки содержимого DOM. К объекту `document` добавьте прослушиватель событий, анализирующий `DOMContentLoaded`. Как только это событие будет запущено, передайте объект события и строковое значение `Document Ready` в функцию вывода `message`.
4. Измените порядок прослушивателей событий, разместив оператор события `document` перед окном `onload`. Повлияет ли это на результат?

5. С помощью объекта `document` добавьте прослушиватель события `DOMContentLoaded` — он должен отправлять в функцию аргументы `Document Ready` и объект вызванного события.
6. Запустите скрипт и посмотрите, какое событие выполнится первым. Измените порядок событий, чтобы увидеть, изменится ли последовательность вывода.

Обработчик событий мыши

Существуют различные обработчики событий мыши (события мыши — это действия, которые она выполняет):

- `ondblclick` — двойной щелчок кнопкой мыши;
- `onmousedown` — кнопка нажата и удерживается;
- `onmouseup` — кнопка больше не удерживается;
- `onmouseenter` — указатель перемещается на элемент;
- `onmouseleave` — указатель покидает область элемента и его дочерних элементов;
- `onmousemove` — указатель перемещается через элемент;
- `onmouseout` — указатель покидает отдельный элемент;
- `onmouseover` — указатель зависает над элементом.

Рассмотрим один из них в действии. Как думаете, что делает следующий код?

```
<!doctype html>
<html>
  <body>
    <div id="divvy" onmouseover="changeColor()" style="width: 100px; height:
100px; background-color: pink;">
    <script>
      function changeColor() {
        document.getElementById("divvy").style.backgroundColor = "blue";
      }
    </script>
  </body>
</html>
```

Как только указатель зависнет над розовым квадратом (`div` с `id="divvy"`), тот сразу же изменит цвет на голубой. Так происходит потому, что в HTML-код вставлен `onmouseover`, указывающий на JavaScript-функцию, которая изменяет цвет квадрата.

Рассмотрим аналогичный, немного более сложный пример.

```
<!doctype html>
<html>
  <body>
    <div id="divvy" style="width: 100px; height: 100px; backgroundcolor: pink;">
    <script>
      window.onload = function donenow() {
        console.log("hi");
        document.getElementById("divvy").addEventListener("mousedown",
          function() { changeColor(this, "green"); });
        document.getElementById("divvy").addEventListener("mouseup",
          function() { changeColor(this, "yellow"); });
        document.getElementById("divvy").addEventListener("dblclick",
          function() { changeColor(this, "black"); });
        document.getElementById("divvy").addEventListener("mouseout",
          function() { changeColor(this, "blue"); });
      }
      console.log("hi2");

      function changeColor(el, color) {
        el.style.backgroundColor = color;
      }
    </script>
  </body>
</html>
```

Снова начинаем с розового квадрата. К элементу `div` прикреплены четыре прослушивателя событий:

- `mousedown` — когда кнопка мыши нажата, но еще не отпущена, квадрат примет зеленый цвет;
- `mouseup` — как только кнопка будет отпущена, квадрат станет желтым;
- `dblclick` — наш любимый вариант. Как думаете, что произойдет при двойном щелчке? Двойной щелчок включает в себя два события `mousedown` и два `mouseup` (перед вторым `mouseup` двойного щелчка нет). Итак, квадрат будет зеленым, желтым, зеленым, черным (и останется черным до наступления другого события);
- `mouseout` — когда указатель сдвинется из области квадрата, фигура примет голубой цвет и останется такой, пока не наступит одно из предшествующих событий.

Обработчики событий мыши обеспечивают широкие возможности взаимодействия. С их помощью вы можете сделать много интересного. Скажем, вам нужен очень динамичный инструмент для принятия решений, основанный на событии `mouseover`. Это будет таблица из четырех столбцов, первый — со статическим содержимым, остальные — с динамическим. Первый столбец определяем под категории продуктов. Во втором будут описаны подкатегории этих категорий. В третьем будут названы сами продукты, а в четвертом будет дана информация о них. Подобный инструмент потребует множество прослушивателей событий и большое количество операций по удалению и добавлению прослушивателей.

Практическое занятие 11.4

Наша цель — изменить цвет фона элемента на странице в зависимости от происходящих событий мыши. При возникновении события `mousedown` элемент должен сменить цвет на зеленый. Когда курсор зависнет над элементом — элемент должен стать красным, а когда переместится за границы элемента — желтым. При щелчке цвет элемента должен меняться на зеленый, а при отпускании клавиши — на голубой. Действия также должны фиксироваться в консоли.

1. Создайте пустой элемент и назначьте ему класс.
2. Выберите элемент, используя название класса.
3. Назначьте для элемента переменную.
4. Обновите содержимое элемента, чтобы в нем говорилось: `hello world`.
5. Измените высоту и ширину элемента и задайте цвет фона по умолчанию, используя свойства стиля элемента.
6. Создайте функцию для обработки двух элементов: первым будет значение цвета в виде строковой переменной, а вторым — объект события вызова.
7. Выведите на экран значение цвета для функции и тип события — для события.
8. Добавьте прослушиватель событий для элементов `mousedown`, `mouseover`, `mouseout` и `mouseup`. Отправьте созданной вами функции два аргумента каждого из этих событий: значение цвета и объект события.
9. Запустите код и проверьте его в своем браузере.

Свойство события target

Всякий раз, когда происходит событие, переменная события становится доступной. У нее много свойств, и вы можете их просмотреть. Примените команду, которая выполняется для события, в функции:

```
console.dir(event);
```

В консоли отобразится множество свойств. Одно из интересующих нас на данный момент — свойство `target` (цель). Это HTML-элемент, запускающий события. Мы можем использовать его для получения различной информации от веб-страницы. Рассмотрим простой пример:

```
<!doctype html>
<html>
  <body>
```

```
    <button type="button" onclick="triggerSomething()">Click</button>
    <script>
      function triggerSomething() {
        console.dir(event.target);
      }
    </script>
  </body>
</html>
```

В данном случае целью `event.target` является элемент `button`. Поэтому в консоли будет выведен элемент `button` и все его свойства, включая потенциальные сестринские и родительские элементы.

Создание HTML-форм, где есть несколько полей ввода и кнопка, — вариант, для которого использование родительских свойств может быть очень удобным. Форма обычно является прямым родителем находящейся в ней кнопки. Через этот родительский элемент можно извлекать данные из полей ввода, как показано в следующем примере:

```
<!doctype html>
<html>
  <body>
    <div id="welcome">Hi there!</div>
    <form>
      <input type="text" name="firstname" placeholder="First name" />
      <input type="text" name="lastname" placeholder="Last name" />
      <input type="button" onclick="sendInfo()" value="Submit" />
    </form>
    <script>
      function sendInfo() {
        let p = event.target.parentElement;
        message("Welcome " + p.firstname.value + " " + p.lastname.value);
      }

      function message(m) {
        document.getElementById("welcome").innerHTML = m;
      }
    </script>
  </body>
</html>
```

В результате мы получим небольшую форму, подобную приведенной на рис. 11.1.

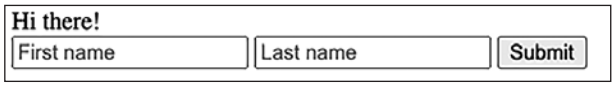


Рис. 11.1. Базовая HTML-форма

Как только вы введете данные в поля и нажмете кнопку Submit, форма примет вид как на рис. 11.2.

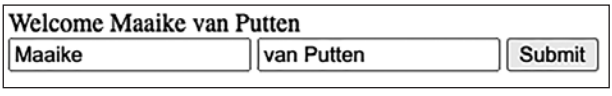


Рис. 11.2. Данные, введенные в базовую HTML-форму и динамическое приветственное сообщение

Следующая команда записывает `event.target[CIT]` в функционал HTML-кнопки:

```
let p = event.target.parentElement;
```

Родительский элемент кнопки (в данном случае форма) сохраняется в переменной `p`. `p` является родителем и представляет элемент формы. Таким образом, следующая команда получит значение данных из окна ввода:

```
p.firstname.value;
```

`p.lastname.value` аналогично получит значение фамилии. Мы раньше не встречались с этим свойством — в двух словах, `value` помогает вам получить данные из элементов ввода.

Далее оба вводимых значения объединяются и передаются в функцию `message()`. Эта функция меняет внутренний HTML-код в элементе `div` на персонализированное приветственное сообщение. Вот почему текст Hi there! меняется на Welcome Maaike van Putten.

Практическое занятие 11.5

Это упражнение поможет понять, как получить значение поля ввода и поместить его в элемент страницы, а также отследить нажатия кнопок и вывести подробную информацию о целевом свойстве события.

Измените текст в элементе `div`. В качестве шаблона можете использовать следующий HTML-документ, добавив туда JavaScript:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
```

```
    <div class="output"></div>
    <input type="text" name="message" placeholder="Your Message">
    <button class="btn1">Button 1</button>
    <button class="btn2">Button 2</button>
    <div>
        <button class="btn3">Log</button>
    </div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Используйте представленный HTML-код в качестве шаблона. Добавьте код JavaScript, чтобы выбрать каждый элемент страницы, включая `div`, поле ввода и элемент `button`. Поместите эти элементы в переменные.
2. Создайте пустой массив `log`, который будет использоваться для отслеживания и фиксации всех событий.
3. Напишите функцию, которая будет фиксировать сведения о событии в объекте, добавляя его в массив `log`. Используя свойство `target`, создайте объект и добавьте его в массив, где временно хранятся входное значение, тип события, имя класса и имя тега элемента `target`.
4. В функции логирования событий получите содержимое поля ввода и присвойте это значение для `textContent` элемента `div`.
5. Очистите содержимое элемента `div` после того, как запишете информацию в массив `log`.
6. Добавьте к двум первым кнопкам прослушиватель событий, отправляющий объект события в созданную ранее функцию отслеживания.
7. Добавьте прослушиватель событий третьей кнопке, выводящей содержимое журнала на экран.

Поток событий DOM

Посмотрим, что произойдет, если вы щелкнете на элементе, с которым связано несколько других элементов.

Мы собираемся создать вложенные `div`-элементы. Для большей наглядности примера мы добавили в `body` стиль. (На самом деле стили лучше прописывать в `head`, а еще лучше иметь отдельный CSS-файл, но актуальный вариант немного короче.) Вот как вложенные `div`-элементы будут выглядеть (рис. 11.3).

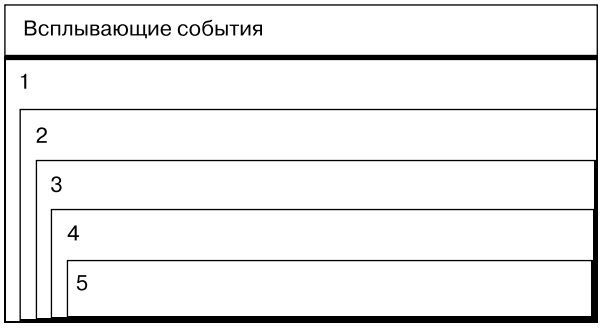


Рис. 11.3. Всплытие событий на веб-странице

Ниже приведен связанный с этим изображением код. Скрипт находится в нижней части кода и будет выполнен после вышестоящих фрагментов. Он добавит прослушиватель событий каждому элементу `div`, и далее все, что он будет делать, — регистрировать `innerText`. Значение внешнего элемента из серии вложенных `div` будет равно `12345`, где каждая цифра будет указана с новой строки.

Итак, вопрос: как данный код будет вызывать события? Скажем, мы щелкнем на цифре `5` — что произойдет? Запустятся события всех вложенных элементов `div` или только одного из пяти? И если будут выполнены все события, каким будет порядок их запуска: от внутреннего события к внешнему или наоборот?

```
<!DOCTYPE html>
<html>
  <body>
    <style>
      div {
        border: 1px solid black;
        margin-left: 5px;
      }
    </style>
    <div id="message">Bubbling events</div>
    <div id="output">
      1
      <div>
        2
        <div>
          3
          <div>
            4
            <div>5</div>
          </div>
        </div>
      </div>
    </div>
    <script>
      function bubble() {
```

```
        console.log(this.innerText);
      }
      let divs = document.getElementsByTagName("div");
      for (let i = 0; i < divs.length; i++) {
        divs[i].addEventListener("click", bubble);
      }
    </script>
  </body>
</html>
```

Код ведет себя по умолчанию. Он выполнит все пять событий, каждое из которых вложено в свой `div`. Порядок запуска событий — от внутреннего к внешнему. Таким образом, первый `innerText` будет равен 5, потом 45 — и так далее вплоть до последнего 12345 (рис. 11.4).

Данное явление называется *всплытием событий*. Так происходит, когда вы запускаете для элемента конкретные обработчики: сначала выполняются собственные события элемента, затем — его родителей, и так далее. Термин «всплытие» выбран потому, что запуск осуществляется от внутреннего события к внешнему — выглядит все это как поднимающиеся воздушные пузырьки.

Подобное поведение можно изменить, указав `true` в качестве третьего аргумента при добавлении прослушивателя событий:

```
divs[i].addEventListener("click", bubble, true);
```

Результат будет следующим (рис. 11.5).

Рис. 11.4. Вывод на экран событий

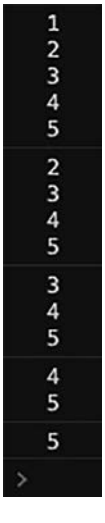


Рис. 11.5. Отображение на экране погружения событий

Перемещение от внешнего элемента к внутреннему называется *погружением событий*. В настоящее время данный прием больше не используется, но если вам действительно нужно его реализовать, используйте аргумент `useCapture` для `addEventListener()` (третий аргумент), установив ему значение `true` (по умолчанию его значение равно `false`).

Погружение и всплытие событий позволяют нам применять *делегирование событий*. Делегирование событий — это концепция, при которой вместо добавления обработчиков событий к каждому элементу конкретного HTML-блока мы определяем оболочку и добавляем событие к ней, после чего оно также применяется ко всем дочерним элементам. Реализуем этот принцип в следующем упражнении.

Практическое занятие 11.6

Данный пример демонстрирует реализацию погружения и делегирования событий. Благодаря тому что прослушиватель событий добавлен к родительскому и дочерним элементам в главном элементе, сообщения на экране будут упорядочены в соответствии с заданными погружением событий свойствами.

У всех элементов `div` со свойством класса `box` будет один и тот же объект события. Мы можем также выводить в консоль целевое событие `textcontent`, определяя таким образом, какой элемент нажат.

Используйте следующий шаблон:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
    <style>
        .box {
            width: 200px;
            height: 100px;
            border: 1px solid black
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="box" id="box0">Box #1</div>
        <div class="box" id="box1">Box #2</div>
        <div class="box" id="box2">Box #3</div>
        <div class="box" id="box3">Box #4</div>
    </div>
    <script>
```

```
    </script>
  </body>
  </html>
```

Выполните следующие шаги.

1. В JavaScript-коде выделите все элементы с классами; отдельно выберите элементы основного контейнера.
2. Добавьте к контейнеру прослушиватель событий, выводящий на экран значение `4` при аргументе `useCapture`, равном `false`, и значение `1` при аргументе `useCapture`, равном `true`.
3. Для каждого из вложенных элементов добавьте прослушиватели события `click`. при значении `console.log()`, равном `3` для аргумента `useCapture` в значении `false` и равном `2` для аргумента `useCapture` в значении `true`.
4. Пощелкайте на элементах страницы, чтобы увидеть, как работает делегирование событий и в каком порядке выводятся значения на странице.
5. В обоих событиях `click` для элементов `box` добавьте вывод на экран значения `target` для `textContent`.

События onchange и onblur

Два других события, которые часто используются при работе с полями ввода, — это `onchange` и `onblur`. `onchange` срабатывает при изменении элемента, например значения поля ввода. `onblur` срабатывает, когда объект выходит из фокуса. К примеру, когда курсор находится в одном поле ввода, а затем перемещается в другое — запускается событие `onblur` первого поля ввода.

Перед вами пример реализации обоих событий. Это фрагмент HTML-элемента `body`. Поля ввода содержат `onblur` и `onchange`, а также дополнительную функцию.

```
<!DOCTYPE html>
<html>
  <body>
    <div id="welcome">Hi there!</div>
    <form>
      <input type="text" name="firstname" placeholder="First name"
             onchange="logEvent()" />
      <input type="text" name="lastname" placeholder="Last name"
             onblur="logEvent()" />
      <input type="button" onclick="sendInfo()" value="Submit" />
    </form>
    <script>
      function logEvent() {
        let p = event.target;
```

```
        if (p.name == "firstname") {
          message("First Name Changed to " + p.value);
        } else {
          message("Last Name Changed to " + p.value);
        }
      }

      function sendInfo() {
        let p = event.target.parentElement;
        message("Welcome " + p.firstname.value + " " + p.lastname.value);
      }

      function message(m) {
        document.getElementById("welcome").innerHTML = m;
      }
    </script>
  </body>
</html>
```

Поле ввода `firstname` содержит событие `onchange`. Если значение данных в поле ввода изменяется, событие срабатывает — при условии, что поле ввода выходит из фокуса. Если поле ввода выходит из фокуса, а значение остается неизменным, событие `onchange` не активируется. Данное условие несправедливо для `onblur`, присвоенного полю ввода `lastname`: даже если значение не изменилось, событие будет запущено.

Другим событием, которое часто используется в работе с полями ввода, является `onfocus`, или просто `focus` (когда применяется в сочетании с прослушивателем событий). Данное событие связано с вводом курсора в поле: оно запускается, когда курсор появляется в поле ввода и позволяет ввести данные.

Практическое занятие 11.7

JavaScript будет прослушивать изменения содержимого в поле ввода. Разместим на странице два поля ввода. Соответствующее событие будет вызвано, как только поле ввода выйдет из фокуса, а значение изменится. Также добавим в поля ввода `blur` и `focus`, которые будут выводиться по мере возникновения конкретных событий. Прослушиватели событий для обоих элементов ввода будут одинаковыми. По мере изменения содержимого полей ввода и смещения фокуса выводимый текст будет обновляться, используя значение поля ввода, которое вызвало событие.

Используйте следующий HTML-шаблон:

```
<!doctype html>
<html>
<head>
```

```
    <title>JS Tester</title>
</head>
<body>
    <div class="output1">
    </div>

    <input type="text" placeholder="First Name" name="first"><br>
    <input type="text" placeholder="Last Name" name="last"><br>
    <script>
    </script>
</body>
</html>
```

Выполните следующие действия.

1. В JavaScript-коде поместите HTML-элемент выходных данных в переменной, которую вы будете использовать для отображения содержимого на странице.
2. Выберите оба поля ввода. Можно применить `querySelector()` и `"input [name='first']"`, которые позволят вам сделать выбор с помощью имени поля.
3. Добавьте прослушиватель событий для отслеживания измененных значений к первому и второму полям ввода. Соответствующие изменения будут вызваны только в том случае, если значение в поле станет другим, а вы щелкнете на поле ввода.
4. Создайте отдельную функцию для обработки вывода содержимого на страницу, обновляя `textContent` вывода.
5. Отправляйте значения входных полей по мере их изменения в `textContent` вывода.
6. Добавьте четыре дополнительных прослушивателя событий и проанализируйте `blur` и `focus` каждого ввода. Как только событие будет запущено, выведите на экран тип события.

Обработчик событий клавиатуры

Существует несколько событий клавиатуры. Одно из них — `onkeypress`. Событие `onkeypress` вызывается — наверное, вы уже догадались — всякий раз, когда нажимается клавиша. Нажатие означает, что кнопка нажата и отпущена. Если хотите, чтобы событие произошло при зажатой кнопке (то есть перед ее отпусканием), можете использовать событие `onkeydown`. Если нужно, чтобы событие произошло при отпускании клавиши, пропишите событие `onkeyup`.

Есть много вещей, которые мы можем сделать с ключевыми событиями, — например, ограничить количество символов, отображаемых в поле ввода. Каждый раз, когда нажимается клавиша, мы можем проверить символ и решить, будет ли он помещен в поле.

Получить значение клавиши, которое вызвало событие, можно следующим образом:

```
event.key;
```

Следующий HTML-код содержит два поля ввода. Можете понять, что здесь происходит?

```
<!doctype html>
<html>
  <body>
    <div id="wrapper">JavaScript is fun!</div>
    <input type="text" name="myNum1" onkeypress="numCheck()">
    <input type="text" name="myNum2" onkeypress="numCheck2()">
    <script>
      function numCheck() {
          message("Number: " + !isNaN(event.key));
          return !isNaN(event.key);
      }

      function numCheck2() {
          message("Not a number: " + isNaN(event.key));
          return isNaN(event.key);
      }

      function message(m) {
          document.getElementById('wrapper').innerHTML = m;
      }
    </script>
  </body>
</html>
```

Первое поле проверяет, является ли значение числом. Если это число, то вверху будет написано `Number: true`; в противном случае появится запись `Number: false`. Второе поле проверяет, является ли значение нечисловым. Если значение нечисловое, в поле появится `Not a number: true`; в противном случае там будет `Not a number: false`.

Итак, это один из способов применения события `onkeypress`, но мы можем сделать намного больше. Событию `onkeypress` можно добавить оператор `return`:

```
onkeypress="return numCheck2()";
```

Если он вернет значение `true`, значение клавиши будет добавлено в поле ввода. Если вернется `false` — не будет.

Следующий фрагмент кода позволяет вводить в поле только числа. Всякий раз, когда пользователь попытается ввести что-то другое, функция удалит это значение.

```
<!doctype html>
<html>
  <body>
    <div id="wrapper">JavaScript is fun!</div>
    <input type="text" name="myNum1" onkeypress="return numCheck()"
           onpaste="return false">
    <input type="text" name="myNum2" onkeypress="return numCheck2()"
           onpaste="return false">
    <script>
      function numCheck() {
          message(!isNaN(event.key));
          return !isNaN(event.key);
      }

      function numCheck2() {
          message(isNaN(event.key));
          return isNaN(event.key);
      }

      function message(m) {
          document.getElementById("wrapper").innerHTML = m;
      }
    </script>
  </body>
</html>
```

Как видите, команда `return` включена в `onkeypress`, чтобы гарантировать ввод только цифр. Возможно, ваше внимание привлекла еще одна вещь: `onpaste="return false"`. Она нужна для того, чтобы справляться с умными людьми, которые копируют числа в нечисловое поле (или другие символы в числовое поле) и все равно умудряются вводить недопустимые значения.

Практическое занятие 11.8

Распознавая нажатия клавиш и определяя значения символов в фокусе по мере нажатия клавиш, мы можем установить содержимое, скопированное в поле ввода.

1. Создайте два поля ввода в вашем HTML-коде. Добавьте элемент для вывода контента.
2. С помощью JavaScript выберите элементы страницы. Вы можете назначить элементу переменную `output` класса `output`. Создайте другую

переменную с названием `eles` и определите все поля ввода (используя `querySelectorAll()`) как ее значения. Таким образом, мы можем перебирать список узлов и назначать одни и те же события всем соответствующим элементам.

3. Используя `forEach()`, выполните итерацию по всем выводимым на страницу элементам. Добавьте к ним ко всем одни и те же прослушиватели событий.
4. Добавьте прослушиватель `keydown` и проверьте, является ли значение числом. Если значение числовое, добавьте его в область вывода.
5. С наступлением события `keyup` выведите на экран значение соответствующей клавиши.
6. Проверьте, было ли что-либо скопировано в поле ввода. Если было, выведите слово `paste` на экран.

Перетаскиваемые элементы

Существуют специальные обработчики событий для перетаскивания объектов. Чтобы что-то перетащить, требуется начальная точка. Давайте напишем CSS- и HTML-код для области перетаскивания.

```
<!doctype>
<html>
  <head>
    <style>
      .box {
        height: 200px;
        width: 200px;
        padding: 20px;
        margin: 0 50px;
        display: inline-block;
        border: 1px solid black;
      }

      #dragme {
        background-color: red;
      }
    </style>
  </head>
  <body>
    <div class="box">1</div>
    <div class="box">2</div>
  </body>
</html>
```

Теперь мы собираемся добавить элемент, который будет перетаскиваться. Чтобы пометить элемент как перетаскиваемый, необходимо включить атрибут `draggable`. Вот код, который мы впишем во второй `div`, заключенный в первый:

```
    <div class="box"> 1
      <div id="dragme" draggable="true">
        Drag Me Please!
      </div>
    </div>
```

Далее необходимо решить, что произойдет, когда мы отпустим перетаскиваемый элемент. Можно описать это в коде поля, куда будет перемещен элемент. Мы добавим соответствующие функции для обоих полей, таким образом элемент можно будет перемещать из одного поля в другое и обратно.

```
<div class="box" ondrop="dDrop()" ondragover="nDrop()">
      1
  <div id="dragme" ondragstart="dStart()" draggable="true">
    Drag Me Please!
  </div>
    </div>
<div class="box" ondrop="dDrop()" ondragover="nDrop()">2</div>
```

В конец `body` необходимо также добавить следующий скрипт:

```
  <script>
    let holderItem;

    function dStart() {
      holderItem = event.target;
    }

    function nDrop() {
      event.preventDefault();
    }

    function dDrop() {
      event.preventDefault();
      if (event.target.className == "box") {
        event.target.appendChild(holderItem);
      }
    }
  </script>
```

Мы начинаем с описания элемента, который хотим удерживать во время перетаскивания. В начале события `ondragstart` мы сохраняем перетаскиваемый элемент в переменной `holderItem`. Обычно, когда вы перетаскиваете элемент, конструкция

HTML не позволяет его отпустить. Чтобы это все же произошло, необходимо предотвратить событие по умолчанию, запрещающее отпускать элемент, который вы собираетесь перетащить. Можете поступить так:

```
event.preventDefault();
```

Прежде чем нивелировать поведение по умолчанию, необходимо осуществить проверку целевой области: может ли она принять перемещаемый элемент? В предыдущем примере мы хотели убедиться, что имя класса целевого элемента действительно равно `box`. Если условие верно, мы добавляем `holderItem` в качестве дочернего элемента поля.

Мы создали страницу, на которой можно перемещать HTML-элемент из одного поля в другое. Если вы попробуете отпустить элемент в любой другой точке, он вернется в исходное поле.

Практическое занятие 11.9

В рамках упражнения мы проведем проверку типа «Я не робот». С помощью перетаскивания можно убедиться, что на странице находится реальный пользователь, а не бот. Мы создадим визуальный эффект перетаскивания активного элемента, на котором пользователь щелкает кнопкой мыши и удерживает нажатие, чтобы начать действие перетаскивания. Как только кнопка мыши будет отпущена, событие перетаскивания завершится. Успешные действия выводятся в консоли.

Используйте следующий шаблон:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
    <style>
        .box {
            width: 100px;
            height: 100px;
            border: 1px solid black;
            background-color: white;
        }
        .red {
            background-color: red;
        }
    </style>
```

```
</head>
<body>
    <div class="box">1
        <div id="dragme" draggable="true">
            Drag Me Please!
        </div>
    </div>
    <div class="box">2</div>
    <script>
    </script>
</body>
</html>
```

Данный HTML-код создает стили элемента, который будет перемещаться, а также устанавливает ширину, высоту и границу элемента `box`. Он также создает класс `red` и добавляет красный фон к активному элементу — так мы увидим эту активность. Далее мы добавляем два элемента `div` с классами, аналогичными получаемому элементу `box`. Наконец, создаем вложенный в одно из полей элемент `div`, присваиваем ему `id`, равное `dragme`, после чего устанавливаем атрибуту `draggable` значение `true` и добавляем небольшой текст, чтобы помочь пользователю. Завершите скрипт, выполнив следующие действия.

1. Выберите элемент `draggable` как объект в коде JavaScript.
2. Добавьте прослушиватель событий `dragstart`, который установит для элемента `draggable` прозрачность `0.5`.
3. Добавьте еще один прослушиватель событий `dragend`, который удалит значение прозрачности.
4. Выберите все поля для перемещения с помощью `querySelectorAll()`.
5. Добавьте прослушиватели событий во все принимающие поля, установив параметры таким образом, чтобы класс `red` добавлялся к элементу всякий раз, когда пользователь запускает событие `dragenter`, — это оповестит пользователя, что действие осуществлено.
6. Добавив метод `preventDefault()` к элементу, установите `dragover`, чтобы отключить любые действия, существующие по умолчанию.
7. При наступлении события `dragleave` удалите класс `red`.
8. Включив прослушиватель событий `drop` в поле, добавьте элемент `draggable` к цели события.
9. Чтобы код работал со всеми элементами одинаково, удалите в них действия по умолчанию. Для этого можно применить метод `preventDefault()`.
10. Для многих описанных выше событий можно добавить логирование, чтобы поэтапно отследить их выполнение.

Отправка формы

Событие может быть запущено во время отправки формы. Этого можно достичь разными способами. Один из них — добавить к элементу `form` атрибут `onsubmit`:

```
<form onsubmit="doSomething()">
```

Указанная функция будет запускаться всякий раз, когда будет принят ввод типа `submit`:

```
<input type="submit" value="send">
```

С помощью HTML-элемента `form` можно сделать больше — например, перенаправить пользователя на другую страницу. Мы должны указать способ, которым хотим отправить значения формы, используя атрибут `method`, и страницу местоположения, используя атрибут `action`.

```
<form action="anotherpage.html" method="get" onsubmit="doStuff()">
```

Пока не беспокойтесь о `get` — он просто говорит о том, что отправляет значения по URL-адресу. При использовании `get` URL выглядит так:

```
www.example.com/anotherpage.html?name=edward
```

После вопросительного знака передающиеся далее переменные расположены в парах «ключ — значение». Вот какая форма сформировала URL, когда `name` было присвоено значение `edward`:

```
<!doctype html>
<html>
  <body>
    <form action="anotherpage.html" method="get">
      <input type="text" placeholder="name" name="name" />
      <input type="submit" value="send" />
    </form>
  </body>
</html>
```

`anotherpage.html` может использовать переменные из URL. Можно реализовать это в скрипте `anotherpage.html`, например таким образом:

```
<!doctype html>
<html>
  <body>
    <script>
      let q = window.location.search;
      let params = new URLSearchParams(q);
      let name = params.get("name");
      console.log(name);
    </script>
  </body>
</html>
```

До сих пор мы отправляли формы, используя атрибуты `action` и `onsubmit`. `action` перенаправляет в другое место (это может быть конечная точка API другой страницы). `onsubmit` определяет событие, которое выполняется при отправке формы.

Есть и другие интересные вещи, которые можно осуществить с помощью события форм `onsubmit`. Помните применение `return` для `onkeypress`? Что-то похожее можно сделать и здесь! Если функция будет возвращать логическое значение, форма отправится только при логическом значении, равном `true`. Это очень удобно, когда мы хотим выполнить некоторую проверку формы.

Взгляните на этот код — сможете ли вы определить условие, при котором произойдет отправка формы?

```
<!doctype html>
<html>
  <body>
    <div id="wrapper"></div>
    <form action="anotherpage.html" method="get" onsubmit="return valForm()">
      <input type="text" id="firstName" name="firstName"
       placeholder="First name" />
      <input type="text" id="lastName" name="lastName" placeholder="Last name" />
      <input type="text" id="age" name="age" placeholder="Age" />
      <input type="submit" value="submit" />
    </form>
    <script>
      function valForm() {
        var p = event.target.children;
        if (p.firstName.value == "") {
          message("Need a first name!!");
          return false;
        }
        if (p.lastName.value == "") {
          message("Need a last name!!");
          return false;
        }
        if (p.age.value == "") {
          message("Need an age!!");
          return false;
        }
        return true;
      }

      function message(m) {
        document.getElementById("wrapper").innerHTML = m;
      }
    </script>
  </body>
</html>
```

Форма состоит из трех полей ввода и одной кнопки подтверждения. Поля предназначены для ввода имени, фамилии и возраста. Если одно из полей не будет заполнено, форма не отправится — функция вернет значение `false`. В раздел `div` над формой также будет добавлено сообщение с объяснением того, что пошло не так.

Практическое занятие 11.10

В данном упражнении мы создадим средство проверки формы. Необходимо убедиться, что в конкретные поля введены нужные значения. Код будет проверять входные значения на соответствие заранее определенным условиям.

1. Настройте форму, включив в нее три поля ввода: `First`, `Last` и `Age`. Добавьте кнопку отправки.
2. В JavaScript-коде выберите форму в качестве объекта.
3. Добавьте прослушиватель событий для подтверждения отправки формы.
4. Задайте для `error` значение по умолчанию `false`.
5. Создайте функцию `checker()`, которая будет проверять длину строки и выводить значение длины на экран.
6. Добавьте условия к каждому из значений поля. Прежде чем изменить значение переменной `error` на `true`, проверьте, есть ли в поле какое-либо значение. Если ответ равен `false`, должна выводиться ошибка.
7. Примените `console.log()` для вывода деталей ошибки.
8. Проверьте вводимое значение возраста, чтобы узнать, соответствует ли оно 19 годам и старше. В случае несоответствия вызовите ошибку.
9. В конце проверки убедитесь, что значение `true` для переменной `error` является актуальным. Если это так, используйте `preventDefault()`, чтобы остановить отправку формы. Выведите соответствующую ошибку на экран.

Анимация элементов

В конце главы мы хотим показать, как можно оживлять элементы с помощью HTML, CSS и JavaScript. Это позволит совершать с веб-страницей еще более интересные вещи. Например, мы можем вызвать анимацию в качестве события, используя это для самых разных целей: иллюстрации объяснения, привлечения внимания пользователя к определенному месту на странице или проведения игры.

Рассмотрим очень простой пример. Возьмем клавишу `position` и установим для нее в CSS значение `absolute`. Это поможет расположить элемент относительно его ближайшего родительского элемента (в нашем случае `body`). Вот код для фиолетового квадрата, который при нажатии клавиши перемещается слева направо:

```
<!doctype html>
<html>
  <style>
    div {
      background-color: purple;
      width: 100px;
      height: 100px;
      position: absolute;
    }
  </style>
  <body>
    <button onclick="toTheRight()">Go right</button>
    <div id="block"></div>

    <script>
      function toTheRight() {
        let b = document.getElementById("block");
        let x = 0;
        setInterval(function () {
          if (x === 600) {
            clearInterval();
          } else {
            x++;
            b.style.left = x + "px";
          }
        }, 2);

      }
    </script>
  </body>
</html>
```

Чтобы заставить квадрат двигаться, нам нужно присвоить блоку `div` абсолютную позицию. Мы полагаемся на CSS-свойство `left`: чтобы быть слева от чего-то, это что-то должно быть абсолютным, иначе `left` не может позиционироваться относительно него. Именно поэтому нам нужно разместиться на определенное количество пикселей слева от `div`, и именно поэтому необходимо, чтобы положение `div` было абсолютным: тогда позиция движущегося блока может быть указана относительно его родительского элемента.

При нажатии кнопки активируется функция `toTheRight()` — она захватывает `block` и сохраняет его в переменной `b`. `x` будет присвоено значение `0`. Затем мы используем очень мощную встроенную функцию JavaScript: `setInterval()`. Эта функция продолжит вычислять выражение до тех пор, пока не будет вызвана функция `clearInterval()`, — это произойдет, когда `x` (мера нашего удаления налево) до-

стигнет значения `600`. Процесс повторяется каждые 2 миллисекунды, что дает нам эффект скольжения.

В это же время вы можете устанавливать и другие позиции: например, `style.top`, `style.bottom` и `style.right`. Или добавлять новые элементы, чтобы создать эффект падающего снега или равномерно движущегося автомобиля. С этим знанием в копилке даже небо для вас не предел.

Практическое занятие 11.11

В рамках упражнения создадим событие для простого интерактивного элемента на странице: будем щелкать на фиолетовом квадрате и смотреть на его перемещения. Фиолетовый квадрат будет передвигаться при каждом нажатии на него. Как только он достигнет границ страницы, он изменит направление и поплывет слева направо или справа налево, в зависимости от того, какой стороны достигнет.

1. Задайте стиль элемента. Установите `height` и `width`, прежде чем указать для `position` значение `absolute`.
2. Создайте элемент, который хотите перемещать по странице.
3. Выберите и сохраните элемент с помощью JavaScript.
4. Определите объект со значениями `speed`, `direction` и `position`.
5. Добавьте прослушиватель для события щелчка на элементе.
6. Установите для счетчика периодов значение по умолчанию `30`.
7. Если значение счетчика меньше 1, завершите период и очистите его значение.
8. Как только период, используя значение `x`, повторится 30 раз, элемент остановится и будет ожидать нажатия,
9. Во время движения проверяйте значение положения: больше ли оно 800 или меньше 0 — это наши условия изменения направления. Направление движения задает значение `direction`. Если, перемещаясь, элемент выходит за пределы контейнера, нам нужно отправить его в другом направлении: это можно осуществить, умножив значение `direction` на отрицательную единицу. Положительное значение станет отрицательным, отправляя элемент влево. Отрицательное значение станет положительным, отправляя элемент вправо.
10. Обновите значение `style.left` элемента. Добавьте `px`, так как присвоенное значение стиля относится к типу `string`.
11. Выведите результат на экран.

Проекты текущей главы

Ваша собственная аналитика

Определите, на какие элементы страницы нажал пользователь, и запишите их идентификаторы, теги и название класса.

1. Создайте в вашем HTML-коде основной контейнер.
2. Добавьте четыре элемента внутрь контейнера, каждый с классом `box` и уникальным идентификатором с собственным текстовым содержимым.
3. Настройте JavaScript-код так, чтобы он содержал массив, куда вы можете добавлять сведения о каждом щелчке кнопкой мыши.
4. Выберите основной контейнер в качестве переменной.
5. Добавьте прослушиватель событий для отслеживания щелчков кнопкой мыши на элементе.
6. Создайте функцию для обработки щелчков кнопкой мыши. Получите целевой элемент из объекта события.
7. Проверьте, есть ли у элемента идентификатор, чтобы вы не отслеживали щелчки кнопкой мыши на основном контейнере.
8. Задайте объект для отслеживания значений. Включите в него `textContent`, `id`, `tagName` и `className` элемента.
9. Добавьте временный объект в массив отслеживания.
10. Выведите значения, записанные в массиве отслеживания, на экран.

Звездная рейтинговая система

Используя JavaScript, создайте компонент звездного рейтинга, который будет полностью интерактивным и динамичным (рис. 11.6).

Рис. 11.6. Создание системы звездного рейтинга

В качестве исходного шаблона можете использовать следующий HTML- и CSS-код, дополнив необходимыми строками элемент `script`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Star Rater</title>
    <style>
        .stars ul {
            list-style-type: none;
            padding: 0;
        }
        .star {
            font-size: 2em;
            color: #ddd;
            display: inline-block;
        }
        .orange {
            color: orange;
        }
        .output {
            background-color: #ddd;
        }
    </style>
</head>
<body>
    <ul class="stars">
        <li class="star">&#10029;</li>
        <li class="star">&#10029;</li>
        <li class="star">&#10029;</li>
        <li class="star">&#10029;</li>
        <li class="star">&#10029;</li>
    </ul>
    <div class="output"></div>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Выберите внутри `ul` все звезды класса `stars` в отдельный объект. Создайте второй объект для элемента `output`.
2. Напишите другой объект, содержащий результаты вызова `querySelectorAll()` для элементов с классом `star`.
3. Переберите итоговый список узлов с помощью цикла, добавив к объекту значение индекса плюс 1 и прослушиватель событий, отслеживающий щелчки кнопкой мыши. Подключите функцию `starRate()` к событию `click` каждого элемента `star`.

4. Внутри функции `starRate()` добавьте к выводу значение `star`, используя событие `target` и значение `star` объекта, заданное на предыдущем шаге.
5. Переберите все звезды с помощью `forEach()`, чтобы проверить, является ли значение индекса `star` меньшим, чем значение `star`. Если это так, используйте `class orange`. В противном случае удалите `class orange` из элемента `classList`.

Отслеживание позиции мыши

Отследите позиции `x` и `y` указателя внутри элемента. При перемещении мыши внутри элемента значения позиций `x` и `y` будут обновляться.

1. Создайте элемент страницы и задайте ему размеры, включая `height` и `width`. Пропишите класс стиля с названием `active` и свойством фона `background-color` в значении `red`. Наконец, создайте элемент вывода, содержащий ваш текст.
2. Выберите главный контейнер и добавьте в него прослушиватель событий. Отслеживайте `mouseover`, `mouseout` и `mousemove`.
3. При наступлении события `mouseover` добавьте класс `active`. При событии `mouseout` удалите класс `active`.
4. Событие `mousemove` должно вызывать функцию, которая отслеживает позиции `clientX` и `clientY` элемента события, включает их в удобочитаемое предложение и выводит в элемент вывода.

Игра на скорость со щелчками кнопкой мыши

Задача игры заключается в том, чтобы как можно быстрее щелкнуть на красном поле в момент его появления. Поле будет случайным образом помещено в контейнер и расположено согласно случайным значениям. В контейнер будет внедрен прослушиватель событий, отслеживающий время появления поля и щелчка на нем, чтобы рассчитать продолжительность события `click`.

Можете использовать следующий шаблон. HTML-код здесь становится немного сложнее — добавьте `<script>`, чтобы сделать его интерактивным!

```
<!DOCTYPE html>
<html>

<head>
    <title>Click Me Game</title>
    <style>
        .output {
            width: 500px;
            height: 500px;
            border: 1px solid black;
```

```
            margin: auto;
            text-align: center;
        }
        .box {
            width: 50px;
            height: 50px;
            position: relative;
            top: 50px;
            left: 20%;
            background-color: red;
        }
        .message {
            text-align: center;
            padding: 10px;
            font-size: 1.3em;
        }
    </style>
</head>

<body>
    <div class="output"></div>
    <div class="message"></div>
    <script>

    </script>
</body>
</html>
```

Работайте с представленным HTML-кодом, используя JavaScript.

1. Здесь два элемента `div`: один с классом `output` для игровой области, а другой — с классом `message` для поля с инструкциями игроку. Выберите эти главные элементы как объекты, используя JavaScript.
2. Создайте с помощью JavaScript другой элемент внутри `output` и добавьте элемент `div` — главный объект для щелчка кнопкой мыши. Добавьте к новому элементу стиль `box` и присоедините его к `output`.
3. Добавьте с помощью JavaScript в область `message` инструкцию для пользователя: `Press to Start`. Игрок должен будет нажать вновь созданный `div` с классом `box` для начала игры.
4. Создайте глобальный объект `game`, чтобы отслеживать продолжительность игры и значение стартового времени. Он будет использоваться для вычисления интервала в секундах между моментом отображения элемента и моментом, когда игрок нажимает на него. Установите для `start` значение `null`.
5. Создайте функцию, которая будет генерировать случайное число и возвращать случайное значение, причем аргументом будет максимальное значение, которое вы хотите использовать.

6. Добавьте прослушиватель событий для элемента `box`. После нажатия на него должен начаться игровой процесс. Измените параметр отображения элемента `box` на `none`. С помощью метода `setTimeout()` вызовите функцию `addBox()` и установите для задержки случайное значение в миллисекундах. По мере необходимости его можно менять. Данный параметр задает интервал между появлением объекта, на котором нужно щелкнуть кнопкой мыши, и его исчезновением. Если начальное значение равно `null`, добавьте текстовое содержимое в область `message`.
7. Если у `start` есть значение, получите значение Unix-времени с помощью `getTime()` текущего объекта `date`. Вычтите время начала игры из текущего значения времени в миллисекундах, а затем разделите на 1000, чтобы получить значение в секундах. Выведите результат `message`, чтобы игрок мог увидеть свои достижения.
8. Создайте функцию для обработки щелчков, добавив поле, как только закончится время таймера. Обновите текстовое содержание `message`, указав там следующую запись: `Click it...`. Установите значение `start` на текущее время в миллисекундах. Примените стиль `block` к элементу, чтобы он появился на странице.
9. В доступном пространстве (общая ширина контейнера 500 минус ширина поля 50) установите произвольную позицию выше и слева от элемента, используя `Math.random()`.
10. Играйте в игру и обновляйте стиль по мере необходимости.

Вопросы для самопроверки

1. Где можно использовать `window.innerHeight` и `window.innerWidth`?
2. Что делает `preventDefault()`?

Резюме

В этой главе мы разобрали многие способы повышения интерактивности веб-страниц. Мы рассмотрели различные варианты указания событий, а затем углубились в изучение отдельных обработчиков событий. Обработчик события `onload` запускается, когда элемент, для которого он указан (обычно это объект `document`), полностью загружен. Такой способ отлично подходит для упаковки других функций, позволяя избежать выбора содержимого DOM, которого еще нет.

Мы также рассмотрели обработчики событий мыши, реагирующие на различные действия, которые можно выполнить с помощью мыши на веб-странице. Работают все эти обработчики схожим образом, но каждый из них обеспечивает отличный

от остальных тип взаимодействия с пользователем. Затем вы узнали, как получить доступ к элементу, который запускает событие, вызвав `event.target` (свойство, которое содержит элемент, вызывающий событие).

Далее мы более детально разобрали `onchange`, `onblur` и обработчики событий клавиатуры. После этого рассмотрели, как инициировать взаимодействие при отправке форм. Вы познакомились с атрибутом `action` в HTML, который перенаправляет форму, и с событием `onsubmit`, которое определяет действие, выполняющееся при отправке формы. Затем мы рассмотрели некоторые возможности взаимодействия со всеми этими событиями, такие как перетаскивание элементов на странице и анимация элементов.

В следующей главе мы перейдем к некоторым более продвинутым вопросам, изучение которых поднимет ваши JavaScript-навыки на новый уровень!

12 Средний уровень JavaScript

Концепции и подходы, представленные в книге ранее, — это не единственный способ решения описанных проблем. В данной главе мы предложим вам заглянуть немного глубже, проявить любопытство и выработать хорошую привычку оптимизировать решения.

Ранее мы обещали, что в этой главе вас ждет много интересного. Оптимальное использование некоторых встроенных методов требует знания регулярных выражений, которые мы сейчас и изучим. Однако интересных задач будет намного больше. Вот список тем, которые мы рассмотрим далее:

- регулярные выражения;
- функции и объект `arguments`;
- поднятие (hoisting) в JavaScript;
- использование строгого режима;
- отладка;
- использование файлов cookie;
- локальное хранилище;
- JSON.

Как видите, темы разнообразны и не похожи друг на друга, но все они занятные и рассчитаны на продвинутого пользователя. Разделы этой главы не так связаны друг с другом, как, возможно, вы ожидали. В основном здесь рассмотрены отдельные вопросы, которые помогут серьезно улучшить понимание происходящего и углубить ваши знания JavaScript.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Регулярные выражения

Регулярные выражения (также известные как *regex*) — это простой способ описания текстовых шаблонов.

Можете рассматривать их как строковые переменные следующего уровня. Существуют различные реализации регулярных выражений. В зависимости от интерпретатора регулярные выражения могут немного различаться написанием. И все же в некоторой степени они стандартизированы, поэтому везде их принято писать (почти) одинаково. Далее мы будем применять регулярные выражения в формате, используемом в JavaScript.

Регулярные выражения могут быть полезными во многих ситуациях: например, когда вам нужно найти ошибки в большом файле или извлечь идентификационную строку клиентского браузера, в котором работает пользователь. Также их можно использовать для проверки формы: с помощью регулярных выражений вы можете указать допустимые шаблоны таких полей, как адреса электронной почты или номера телефонов.

Регулярные выражения полезны не только для поиска строк, но и для их замены. Наверняка вы словили себя на мысли: *регулярные выражения потрясающие — но есть ли здесь подвох?* И да, к сожалению, он здесь присутствует. Поначалу регулярное выражение может выглядеть так, будто кошка вашего соседа прошлась по клавиатуре и просто набрала несколько случайных символов. Например, следующее регулярное выражение проверяет наличие действительного адреса электронной почты:

```
/([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)/g
```

Не бойтесь: познакомившись с текущим разделом, вы сможете расшифровать эти тайные паттерны. Мы не собираемся подробно рассматривать все, что касается регулярных выражений, но покажем то, благодаря чему вы сможете с ними работать и по ходу дела расширять свои знания.

Начнем с простого. Регулярное выражение задается между двумя слешами. Вот его рабочая запись:

```
/JavaScript/
```

Выражение будет считаться правильным, поскольку обрабатываемая строка содержит слово `JavaScript`. Когда обнаруживается соответствие, результат считается положительным. И это можно использовать для многих целей.

Например, можно применить встроенную JavaScript-функцию `match()`. Эта функция возвращает результат совпадения регулярных выражений (если таковое

имеется) в виде подстроки, которая соответствует начальной позиции этой строки и входной строке.

На самом деле есть и другие встроенные функции, которые используют регулярные выражения, но мы рассмотрим их позже. Пока задержимся на функции `match()` — она удобна для демонстрации работы регулярных выражений. Рассмотрим ее в действии:

```
let text = "I love JavaScript!";
console.log(text.match(/javascript/));
```

В результате регистрируется значение `null` — по умолчанию функция чувствительна к регистру, следовательно, выражение не соответствует заданной записи. Если бы мы искали совпадение по `/ava/` или просто `/a/`, оно бы нашлось, поскольку в выражении есть и то и другое. Если вы не хотите, чтобы функция учитывала регистр, можете задать условие, поставив `i` после закрывающего слеша. В этом примере выражение будет соответствовать предыдущей строке:

```
console.log(text.match(/javascript/i));
```

Результат будет логирован: теперь функция не чувствительна к регистру, поэтому, по ее мнению, наша строка действительно содержит `javascript`. Вот как выглядит запись:

```
[
  'JavaScript',
  index: 7,
  input: 'I love JavaScript!',
  groups: undefined
]
```

Результатом является объект, содержащий найденное совпадение, индекс, в котором оно было зафиксировано, и просмотренные входные данные. Группам присвоено значение `undefined`. (К слову, можно прописывать группы в круглых скобках — вы изучите это в соответствующем разделе.)

Регулярные выражения в JavaScript часто встречаются в сочетании со встроенным методом поиска и замены строк, который мы рассмотрим далее.

Указание нескольких вариантов слов

Чтобы указать определенный диапазон параметров, можно применить такой синтаксис:

```
let text = "I love JavaScript!";
console.log(text.match(/javascript|nodejs|react/i));
```

В этом случае в выражении ожидаются совпадения по `javascript`, `nodejs` или `react`. Код в данном виде выведет первое совпадение и прекратит работу. Операция не приведет к поиску двух или более совпадений — в итоге формат записи в консоли будет таким же, как и раньше:

```
let text = "I love React and JavaScript!";
console.log(text.match(/javascript|nodejs|react/i));
```

Результат:

```
[
  'React',
  index: 7,
  input: 'I love React and JavaScript!',
  groups: undefined
]
```

Если требуется найти все совпадения, можно указать глобальный модификатор `g` — точно так же, как мы задавали условие поиска без учета регистра. В примере ниже проверяется следующее: все совпадения без учета регистра. Модификаторы находятся за последним слешем. Можете применять несколько модификаторов одновременно, как показано ниже, или использовать только `g`:

```
let text = "I love React and JavaScript!";
console.log(text.match(/javascript|nodejs|react/gi));
```

В результате будет возвращено `React` и `JavaScript`:

```
[ 'React', 'JavaScript' ]
```

Как видите, результат выглядит совсем по-другому. Как только вы укажете `g`, функция вернет массив совпадающих слов. Сейчас нас это не слишком удивляет, ведь именно эти слова мы запрашивали. Но в более сложных конструкциях вывод может получиться неожиданным. Об этом и поговорим.

Варианты символов

До сих пор наши выражения были вполне читаемы, правда? Варианты символов — вот из-за чего запись начнет выглядеть несколько запутанно. Допустим, мы хотим выполнить поиск строки, состоящей только из одного символа: `a`, `b` или `c`. Запишем это следующим образом:

```
let text = "d";
console.log(text.match(/[abc]/));
```

В консоли логируется `null`, потому что `d` не является `a`, `b` или `c`. `d` найдется, если основная строка будет выглядеть так:

```
console.log(text.match(/[abcd]/));
```

Результат будет следующим:

```
[ 'd', index: 0, input: 'd', groups: undefined ]
```

Поскольку это диапазон символов, можно записать его короче:

```
let text = "d";
console.log(text.match(/[a-d]/));
```

Если бы нам нужна была и строчная буква, и прописная, мы бы указали такое выражение:

```
let text = "t";
console.log(text.match(/[a-zA-Z]/));
```

На самом деле для достижения аналогичной цели мы могли бы использовать модификатор без учета регистра, но в этом случае он применится ко всему регулярному выражению — а вам нужен только определенный символ:

```
console.log(text.match(/[a-z]/i));
```

Мы получим совпадения в обоих указанных выше вариантах. Если нужно провести поиск также среди чисел, запись оформляется следующим образом:

```
console.log(text.match(/[a-zA-Z0-9]/));
```

Как видим, можно просто объединить диапазоны и искать в них один конкретный символ, подобно тому как мы прописывали его возможные варианты (например, `[abc]`). В приведенном выше примере указаны три возможных диапазона. Результат будет соответствовать любой строчной или прописной букве от a до z, а также всем цифровым символам.

Кстати, это не значит, что таким образом можно искать только односимвольную строку. В случаях выше результатом будет только первый соответствующий символ, потому что мы не добавили глобальный модификатор. Однако специальным символам соответствие не будет найдено:

```
let text = "äé!";
console.log(text.match(/[a-zA-Z0-9]/));
```

Чтобы решить проблему нахождения сложных символов, в регулярное выражение включается точка: она выполняет функцию специального подстановочного

знака, который может соответствовать любому символу. Как думаете, что делает следующий код?

```
let text = "Just some text.";
console.log(text.match(/./g));
```

Поскольку у функции есть глобальный модификатор, в результате мы получим любой символ — а на самом деле все символы выражения:

```
[
  'J', 'u', 's', 't',
  ' ', 's', 'o', 'm',
  'e', ' ', 't', 'e',
  'x', 't', '.'
]
```

А что, если в выражении требуется найти саму точку? Если хотите, чтобы специальный символ (тот, который используется в регулярных выражениях для указания паттерна) имел обычное значение или обычный символ имел значение специального, можете обособить его с помощью обратного слеша:

```
let text = "Just some text.";
console.log(text.match(/\./g));
```

В этом примере мы экранируем точку, добавляя перед ней обратный слеш. Следовательно, она больше не функционирует как подстановочный знак и будет найдена как буквальное совпадение. В результате мы получим:

```
[ '.' ]
```

Есть несколько обычных символов, которые при добавлении перед ними обратного слеша приобретают значение специальных. Мы не собираемся их описывать подробно, но предлагаем рассмотреть некоторые примеры:

```
let text = "I'm 29 years old.";
console.log(text.match(/\d/g));
```

Если мы обособим `d` (`\d`), будет проведен поиск всех цифр в выражении. Поиск будет глобальным, это значит, нужна любая цифра. Вот что мы получим в итоге:

```
[ '2', '9' ]
```

Можно также обособить `s` (`\s`), что поможет найти все символы пробела:

```
let text = "Coding is a lot of fun!";
console.log(text.match(/\s/g));
```

В общем случае код выведет в консоль табуляцию и другие типы пропусков в тексте. Показанный выше пример вернет в качестве результата несколько пробелов:

```
[ ' ', ' ', ' ', ' ', ' ' ]
```

Очень полезным является специальный символ `\b`, который ищет буквенное сочетание, находящееся в начале слова. Итак, в следующем примере `in` в слове `beginning` найдено не будет:

```
let text = "In the end or at the beginning?";
console.log(text.match(/\bin/gi));
```

Зато в конечном итоге будет найдено `In` в начале предложения:

```
[ 'In' ]
```

Хотя вы можете проверить, являются ли символы числами, метод `match()` определен для объекта `string`, поэтому вы не можете вызвать его на числовых объектах. Попробуйте выполнить следующее:

```
let nr = 357;
console.log(nr.match(/3/g));
```

В результате вы должны получить ошибку `TypeError`, сообщающую, что `nr.match()` не является функцией.

Группы

Существует множество причин, по которым стоит группировать ваши регулярные выражения. Всякий раз, когда вам требуется сопоставить группу символов, вы можете заключить их в круглые скобки. Взгляните на следующий пример:

```
let text = "I love JavaScript!";
console.log(text.match(/(love|dislike)\s(javascript|spiders)/gi));
```

В данном случае сначала будет проведен поиск `love` или `dislike`, потом вставлен символ пробела, а позже проведен поиск `javascript` или `spiders`. При поиске будут зарегистрированы все совпадения вне зависимости от регистра букв. Результат получится следующим:

```
[ 'love JavaScript' ]
```

Здесь мы можем составить примерно четыре комбинации. Две из них, по нашему мнению, имеют больше смысла:

- `love spiders`;
- `dislike spiders`;
- `love JavaScript`;
- `dislike JavaScript`.

Группы являются очень мощным инструментом, если уметь их многократно применять. Посмотрим, как это реализуется. Очень часто вам придется повторять определенный фрагмент регулярных выражений. Решить такую задачу можно несколькими способами. Например, если мы хотим найти последовательности четырех буквенно-цифровых символов, можем сформулировать это так:

```
let text = "I love JavaScript!";
console.log(text.match(/[a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9]/g));
```

В результате получим:

```
[ 'love', 'Java', 'Scri' ]
```

Повторять блоки ужасно. Поищем варианты получше. Если мы хотим, чтобы конкретный блок присутствовал только один раз или не присутствовал вовсе, мы можем использовать знак вопроса. Вот как это выглядит для необязательных символов:

```
let text = "You are doing great!";
console.log(text.match(/n?g/gi));
```

При этом выполняется поиск символа `g`, которому может предшествовать (или нет) символ `n`. В итоге получим:

```
[ 'ng', 'g' ]
```

Возможно, однократное появление не лучший пример повтора. Посмотрим, как получить больше повторений. Если вы хотите, чтобы в консоли отобразилось хотя бы одно значение, но не против, если их будет больше, можете использовать знак `+`. Перед вами пример:

```
let text = "123123123";
console.log(text.match(/(123)+/));
```

Код будет искать однократное или многократное появление группы `123`. И, поскольку эта строка у нас есть, группа будет найдена. Вот что появится в консоли:

```
[ '123123123', '123', index: 0, input: '123123123', groups: undefined ]
```

В данном случае будет выведена вся строка, так как в ней, кроме повторяющихся `123`, больше ничего нет.

В некоторых ситуациях вы захотите, чтобы совпадения определенного фрагмента регулярного выражения были найдены любое количество раз. Подобное в коде обозначается звездочкой (`*`). Вот пример регулярного выражения:

```
/(123)*a/
```

В таком случае будет проведен поиск всех `a`, которым предшествует `123`, повторенное любое количество раз. Таким образом будут найдены следующие совпадения:

- `123123123a`;
- `123a`;
- `a`;
- `ba`.

Последнее, что нужно отметить по поводу повторения, — возможность формировать еще более конкретные запросы с помощью синтаксиса `{min, max}`:

```
let text = "abcabcabc";
console.log(text.match(/(abc){1,2}/));
```

Результат будет следующим:

```
[ 'abcabc', 'abc', index: 0, input: 'abcabcabc', groups: undefined ]
```

Код выполняется так, потому что `abc` встречается в строке как единожды, так и дважды. Как вы можете видеть, мы использовали группы, но в выходных данных они по-прежнему остаются неопределенными. Чтобы это исправить, нужно дать им имена. Вот как это можно сделать:

```
let text = "I love JavaScript!";
console.log(text.match(/(?<language>javascript)/i));
```

Результат в консоли:

```
[
  'JavaScript',
  'JavaScript',
  index: 7,
  input: 'I love JavaScript!',
  groups: [Object: null prototype] { language: 'JavaScript' }
]
```

О регулярных выражениях можно говорить долго, но для выполнения многих интересных задач вам хватит и этих знаний. Давайте рассмотрим несколько практических примеров.

Практическое применение регулярных выражений

Регулярные выражения могут быть полезны во многих ситуациях. Они пригодятся везде, где необходимо найти определенные строки. Давайте обсудим, как вы можете использовать регулярные выражения в сочетании с другими строковыми методами, в том числе для проверки адресов электронной почты и IPv4.

Поиск и замена строк

В главе 8 мы рассмотрели методы поиска и замены строк. Однако теперь мы не хотим, чтобы при поиске учитывался регистр. Угадайте, что мы добавим в регулярное выражение:

```
let text = "That's not the case.";
console.log(text.search(/Case/i));
```

Модификатор `i` позволяет игнорировать различие между прописными и строчными буквами. Код вернет значение `15` — индекс позиции, в которой было зафиксировано совпадение. С обычным вводом такое осуществить невозможно.

Как, по-вашему, с помощью регулярного выражения можно изменить поведение метода `replace`, чтобы заменить все экземпляры, а не только первый экземпляр строки? Опять же добавив модификатор! На сей раз мы используем глобальный модификатор `g`. Чтобы почувствовать разницу, взгляните на выражение без `g`:

```
let text = "Coding is fun. Coding opens up a lot of opportunities.";
console.log(text.replace("Coding", "JavaScript"));
```

Результатом будет:

```
JavaScript is fun. Coding opens up a lot of opportunities.
```

Без применения регулярного выражения в строке будет заменено только первое совпадение. А теперь посмотрите на это же выражение, но уже с глобальным модификатором `g`:

```
let text = "Coding is fun. Coding opens up a lot of opportunities.";
console.log(text.replace(/Coding/g, "JavaScript"));
```

Вывод получится следующим:

```
JavaScript is fun. JavaScript opens up a lot of opportunities.
```

Как видите, все найденные слова были изменены.

Практическое занятие 12.1

Следующее упражнение активизирует замену символов в указанном строковом значении. В первом поле ввода будет указано, какая символьная строка будет заменена, а во втором поле — какие символы заменят их после нажатия кнопки.

Используйте в качестве шаблона приведенный ниже HTML-код и добавьте скрипт, необходимый для выполнения задачи:

```
<!doctype html>
<html>

<head>
    <title>Complete JavaScript Course</title>
</head>

<body>
    <div id="output">Complete JavaScript Course</div>
    Search for:
    <input id="sText" type="text">
    <br> Replace with:
    <input id="rText" type="text">
    <br>
    <button>Replace</button>
    <script>

    </script>
</body>

</html>
```

Выполните следующие шаги.

1. Выберите все три элемента страницы с помощью JavaScript и присвойте их переменным, чтобы на них можно было легко ссылаться в вашем коде.
2. Добавьте к кнопке прослушиватель событий, чтобы при ее нажатии вызывать функцию.
3. Создайте функцию `lookup()`, которая будет находить и заменять текст в элементе вывода. Присвойте текстовое содержимое переменной `s`, а заменяемый ввод — переменной `rt`.

4. Создайте новое регулярное выражение со значением первого поля ввода, которое позволит вам заменить текст. С его помощью, используя метод `match()`, найдите совпадения. Внедрите все это в условие, которое при нахождении совпадений выполнит соответствующий блок кода.
5. Если совпадение найдено, примените `replace()` для присваивания нового значения.
6. Обновите область вывода с помощью недавно созданного и замененного итогового текста.

Подтверждение адреса электронной почты

Чтобы создать шаблон регулярного выражения, важно сначала описать словами каждый его элемент. Шаблон адреса электронной почты состоит из пяти частей и выглядит следующим образом: `[name]@[domain].[extension]`.

Поясним каждую часть.

1. `name` — один или несколько символов алфавита, нижние подчеркивания, тире или точки.
2. `@` — литерал.
3. `domain` — один или несколько символов алфавита, нижние подчеркивания, тире или точки.
4. `.` — литерал.
5. `extension` — один или несколько символов алфавита, нижние подчеркивания, тире или точки.

Итак, оформим наше регулярное выражение в соответствии с этими пунктами.

1. `[a-zA-Z0-9._-]+`.
2. `@`.
3. `[a-zA-Z0-9._-]+`.
4. `\.` (помните: точка является особым символом регулярных выражений, так что необходимо ее обособлять).
5. `[a-zA-Z0-9._-]+`.

Далее мы поместим все в одно выражение:

```
/([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)/g
```

И посмотрим на него в действии:

```
let emailPattern = /([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)/g;
let validEmail = "maaike_1234@email.com";
```

```
let invalidEmail = "maaike@mail@.com";
console.log(validEmail.match(emailPattern));
console.log(invalidEmail.match(emailPattern));
```

Мы протестировали шаблон как на правильно, так и на неправильно записанном адресе электронной почты, и вот что получилось:

```
[ 'maaike_1234@email.com' ]
null
```

Как видите, он возвращает правильный адрес электронной почты и значение `null` (нет совпадения) для недопустимой записи.

Практическое занятие 12.2

Создайте приложение, которое использует JavaScript и регулярные выражения для проверки соответствия строкового входного значения заданному формату адреса электронной почты. Используйте шаблон:

```
<!doctype html>
<html>
<head>
    <title>JavaScript Course</title>
</head>
<body>
    <div class="output"></div>
    <input type="text" placeholder="Enter Email">
    <button>Check</button>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Пропишите в скрипте элементы страницы `input`, `output` и `button`. Выберите их в качестве объектов JavaScript.
2. Добавьте к кнопке прослушиватель событий, чтобы при ее нажатии вызвать блок кода, получающий текущее значение в поле ввода. Создайте пустое значение ответа, которое будет заполнять содержимое `div` вывода.
3. Добавьте тест со строковым значением из поля ввода и выражением формата электронной почты. Если результатом сравнения будет `false`, измените значение вывода (на `Invalid Email`) и его цвет (на `red`).

4. Если условие теста возвращает значение `true`, добавьте ответ, подтверждающий правильность введенного адреса электронной почты, и измените цвет вывода на `green`.
5. Отобразите значение ответа в поле вывода.

Функции и объект arguments

JavaScript обрабатывает аргументы в функциях, добавляя их в пользовательский объект, называемый `arguments`. Этот объект работает во многом как массив, и мы можем использовать его вместо имени параметра. Рассмотрим следующий код:

```
function test(a, b, c) {
  console.log("first:", a, arguments[0]);
  console.log("second:", b, arguments[1]);
  console.log("third:", c, arguments[2]);
}

test("fun", "js", "secrets");
```

Так будет выглядеть результат:

```
first: fun fun
second: js js third:
secrets secrets
```

Когда вы обновляете один из параметров, аргумент соответствующим образом изменяется. То же самое работает и в обратную сторону:

```
function test(a, b, c) {
  a = "nice";
  arguments[1] = "JavaScript";
  console.log("first:", a, arguments[0]);
  console.log("second:", b, arguments[1]);
  console.log("third:", c, arguments[2]);
}

test("fun", "js", "secrets");
```

Код изменит `arguments[0]` и `b`, так как они относятся соответственно к `a` и `arguments[1]`. Это видно по результату в консоли:

```
first: nice nice
second: JavaScript JavaScript
third: secrets secrets
```

Данный способ позволяет получить доступ к аргументам, если функция вызывается с бóльшим количеством аргументов, чем было объявлено в ее сигнатуре. Однако более актуальным является использование параметра `rest` вместо объекта `arguments`.

Если вы забыли, что такое rest, освежите память, заглянув в главу 6.

Практическое занятие 12.3

В этом упражнении мы используем подобный массиву объект `arguments` и извлечем из него значения. Мы переберем элементы в аргументах и вернем последний элемент в списке, используя свойство длины `arguments`.

1. Создайте функцию без каких-либо параметров. Создайте цикл для итерации по длине объекта `arguments` — это позволит выполнять итерацию каждого элемента аргументов в функции.
2. Задайте переменную `lastOne` с пустым значением.
3. При переборе аргументов установите для `lastOne` текущее значение аргумента, возвращая его с помощью индекса `i`. Аргумент будет иметь индекс, который можно использовать для ссылки на значение при выполнении итерации по объекту `arguments`.
4. Верните в качестве ответа значение `LastOne`, которое должно содержать только значение последнего аргумента.
5. Выведите ответ из функции, передайте в функцию ряд аргументов и логируйте результат в консоль. Перед вами должен появиться только последний элемент списка. Если вы хотите просмотреть каждый из элементов, можете выводить их отдельно на экран при просмотре значений или создать массив, который затем можно возвращать, добавляя элементы по мере перебора аргументов.

Поднятие в JavaScript

В главе 6 мы обсуждали три вида переменных: `const`, `let` и `var`. Мы строго рекомендовали вам применять `let` вместо `var` из-за различий в области их видения. *Поднятие (hoisting)* в JavaScript — причина такой рекомендации. Поднятие — это принцип перемещения объявлений переменных в верхнюю часть области, в ко-

торой они определены. Это позволяет вам делать то, чего вы не можете делать во многих других языках, и, кстати, на то есть веские причины. Фрагмент кода ниже вроде как выглядит нормально:

```
var x;
x = 5;
console.log(x);
```

В консоли просто выводится `5`. Но благодаря поднятию происходит следующее:

```
x = 5;
console.log(x);
var x;
```

Если попытаться сделать то же самое с `let`, вы получите ошибку `ReferenceError`. Вот почему лучше использовать `let`: очевидно, что такое поведение очень трудно понимаемо, непредсказуемо и на самом деле не особо вам нужно.

Причина подобного поведения заключается в том, что интерпретатор JavaScript перемещает все объявления `var` перед обработкой файла в его верхнюю часть — объявления, а не инициализации. Поэтому при использовании переменной вы получите `undefined`: вы же ее не инициализировали.

Поднятие было придумано, чтобы освобождать память, но его побочные эффекты неприемлемы. Тем не менее есть способ его отключить, давайте посмотрим, как именно.

Использование строгого режима

Мы можем модифицировать понимающее и всепрощающее поведение JavaScript с помощью строгого режима. Включается он следующей строкой (данная команда в вашем коде должна быть первой):

```
"use strict";
```

Вот один из примеров того, что происходит, когда мы не используем строгий режим:

```
function sayHi() {
  greeting = "Hello!";
  console.log(greeting);
}

sayHi();
```

Итак, мы забыли объявить `greeting`. В обычной ситуации JavaScript исправит нашу ошибку, добавив переменную `greeting` на самый верхний уровень программы — в ответ в консоли появится `Hello!`. Однако в строгом режиме программа выдаст ошибку:

```
"use strict";

function sayHi() {
  greeting = "Hello!";
  console.log(greeting);
}

sayHi();
```

Ошибка:

```
ReferenceError: greeting is not defined
```

Вы можете использовать строгий режим только в определенной функции: просто добавьте его в начало этой функции. Строгий режим влияет и на некоторые другие вещи. Например, при заданном строгом режиме вы сможете использовать в качестве переменных и функций меньше слов, чем обычно, ведь в будущем они, скорее всего, станут зарезервированными ключевыми словами, которые понадобятся JavaScript для его собственного языка.

Строгий режим — отличный способ привыкнуть к особенностям использования JavaScript при настройке фреймворков или даже будущем написании TypeScript. В настоящее время строгий режим считается хорошей практикой, поэтому рекомендуем по возможности к нему прибегать, хоть часто это не лучший (простейший) вариант работы с существующим старым кодом.

Теперь, когда мы рассмотрели строгий режим, пришло время погрузиться в совершенно другой режим: отладки! Режим отладки предназначен для тех случаев, когда вы не заняты написанием или запуском своего приложения. Вы особым образом его активизируете, чтобы определить местоположение любых ошибок.

Отладка

Отладка — тонкое искусство. Поначалу бывает очень трудно определить, что не так с вашим кодом. Если вы запускаете JavaScript в браузере и он ведет себя неожиданным образом, первый шаг — открыть консоль браузера. Обычно там отображаются ошибки, способные помочь вам в дальнейшем.

Если это не решит проблему, можно выводить в консоли каждый шаг вашего кода и фиксировать переменные. Это даст вам некоторое представление о происходящих

процессах. Возможно, вы просто обращаетесь к переменной, которая не определена. Или ожидаете определенного значения от математического вычисления, в котором допущена ошибка, поэтому результат совершенно отличается от задуманного. Довольно распространенный способ контроля — использование `console.log()` на всех этапах разработки.

Контрольные точки

Более профессиональный метод отладки — это установка контрольных точек (в некоторых редакторах — *точек останова*). Подобное можно осуществить в большинстве браузеров и IDE. Щелкаете кнопкой мыши на строке перед нужным кодом (на панели Sources (Источники) браузера Chrome, в других браузерах названия могут отличаться), где появляется точка или стрелочка. Теперь ваше приложение после запуска будет приостанавливаться на данном этапе, чтобы дать вам возможность проверить значения переменных и пройти оттуда по коду построчно.

Таким образом, вы поймете, что происходит и как это исправить. Вот как используются контрольные точки в Chrome (и аналогично в большинстве других браузеров). Перейдите на вкладку Sources (Источники) панели Inspect (Панель разработчика). Выберите файл, в котором хотите создать контрольные точки. Щелкните на номере строки — и там установится контрольная точка (рис. 12.1).

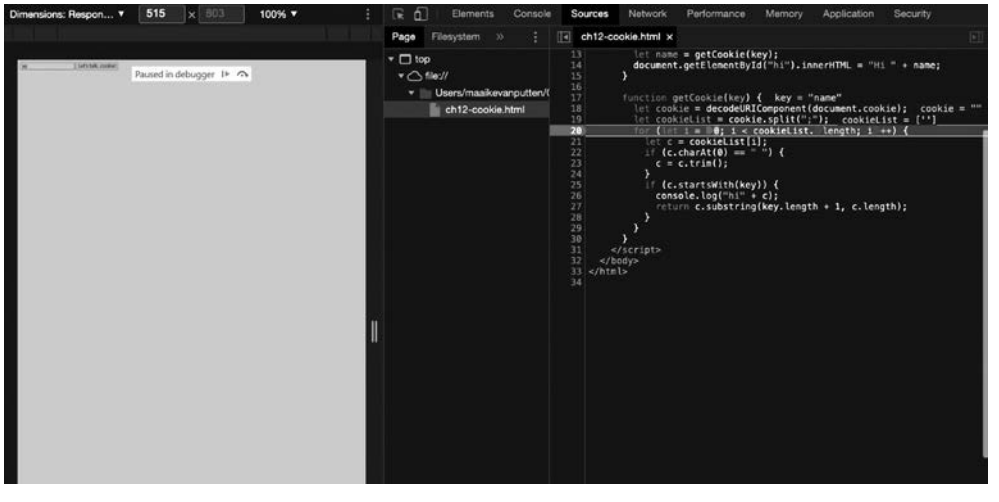


Рис. 12.1. Контрольные точки в браузере

Попробуйте запустить строку кода. Она начнет выполняться и остановится. В правом углу экрана отобразятся все переменные и значения (рис. 12.2).

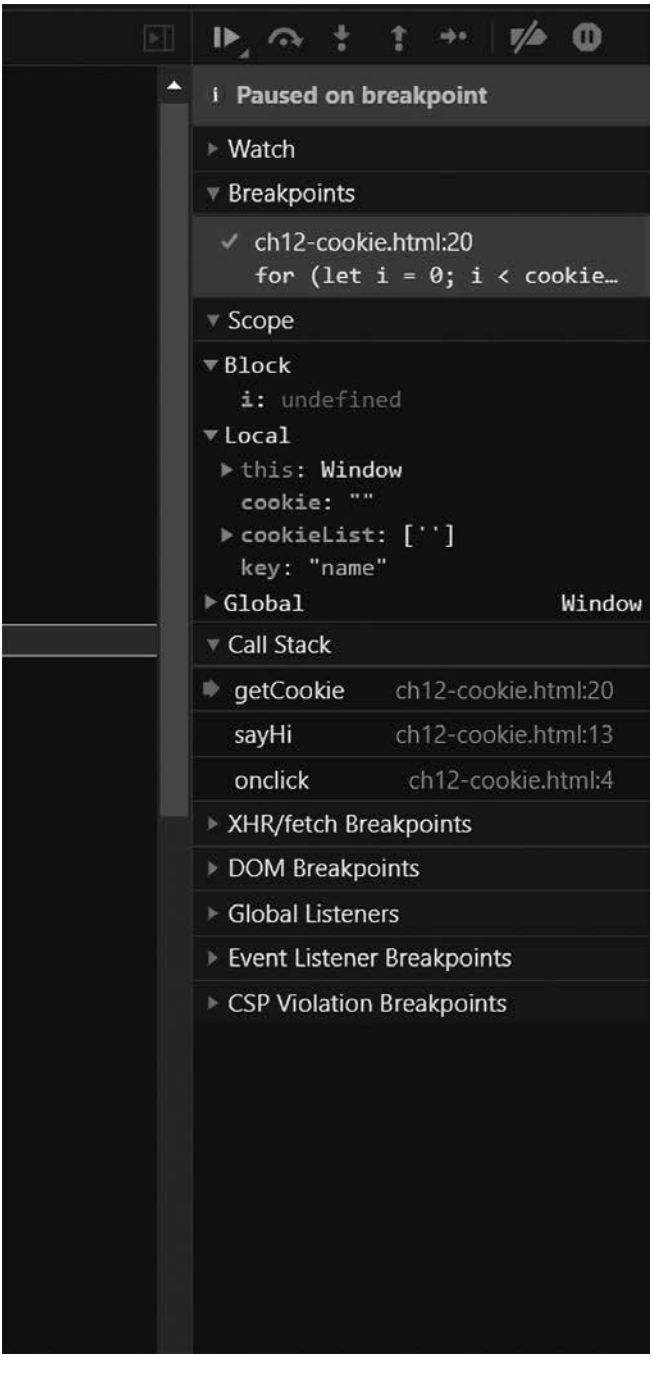


Рис. 12.2. Изучение переменных из контрольных точек

Теперь вы можете детально рассмотреть свой код. Нажатие на значок воспроизведения сверху возобновляет выполнение скрипта (до тех пор пока он не достигнет следующей контрольной точки или не вернется к той же точке). Нажатие на значок круглой стрелки позволит перейти к следующей строке и снова проверить значения.

Есть много вариантов работы с контрольными точками, которые мы не будем здесь описывать. Для получения более подробной информации об отладке кода с помощью контрольных точек загляните в документацию выбранного вами редактора или ознакомьтесь с соответствующей документацией Google Chrome: https://developer.chrome.com/docs/devtools/javascript/ breakpoints/.

Практическое занятие 12.4

В следующем упражнении мы покажем, как использовать контрольные точки редактора для проверки значения переменной в определенный момент выполнения скрипта. Это простой пример, но аналогичный процесс может быть использован, чтобы найти информацию о работе более крупных скриптов в определенных точках во время выполнения кода или чтобы определить суть возникшей проблемы.

В работе контрольных точек есть небольшие нюансы, зависящие от конкретного редактора. Детально ознакомьтесь с документацией вашей среды разработки: это должно дать вам представление о том, что позволят сделать контрольные точки, когда дело дойдет до отладки.

В качестве примера можете использовать следующий короткий скрипт:

```
let val = 5;
val += adder();
val += adder();
val += adder();
console.log(val);
function adder(){
    let counter = val;
    for(let i=0;i<val;i++){
        counter++;
    }
return counter ;
}
```

Не забывайте добавлять тег <script>. Открывайте скрипт в виде HTML-документа, если тестируете его в консоли браузера.

Это упражнение было протестировано в настольном редакторе, но оно в равной степени реализуемо в браузерных консолях и других средах разработки. Выполните следующие шаги.

1. Откройте скрипт в выбранном редакторе (или на вкладке Sources панели Inspect вашего браузера). Щелкните слева от строки кода, где вы хотели бы добавить контрольную точку. Появится точка или какой-то другой индикатор, сообщающий, что контрольная точка действительно установлена (рис. 12.3).

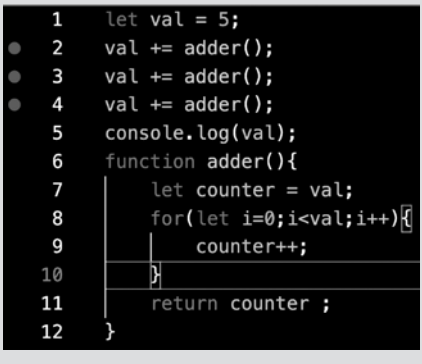


Рис. 12.3. Установка контрольных точек

2. Запустите код с заданными контрольными точками. Мы выбрали Run ▸ Start Debugging (Запуск ▸ Начало отладки), но в вашем браузере название этой команды может быть другим. Перезагрузите веб-страницу, если используете браузерную консоль, чтобы повторно запустить код с учетом ваших новых контрольных точек (рис. 12.4).

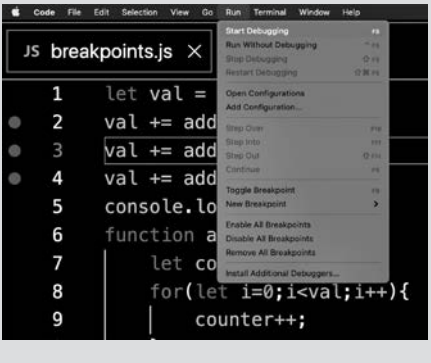


Рис. 12.4. Запуск кода с добавленными контрольными точками

3. Теперь вы увидите консоль отладки. Там должна быть вкладка со списком переменных в коде и текущими значениями на первой контрольной точке. В нашем редакторе она носит название VARIABLES (Переменные), а в браузере Chrome — Scope (Область действия).
4. Вы можете использовать опции меню для перехода к следующей контрольной точке, остановки отладки или перезапуска последовательности контрольных точек. Нажмите значок воспроизведения, чтобы перейти к следующей контрольной точке. Он обновится до значения 5, как указано в строке 1, и остановится в первой контрольной точке. Обратите внимание, что выделенная строка еще не была запущена (рис. 12.5).

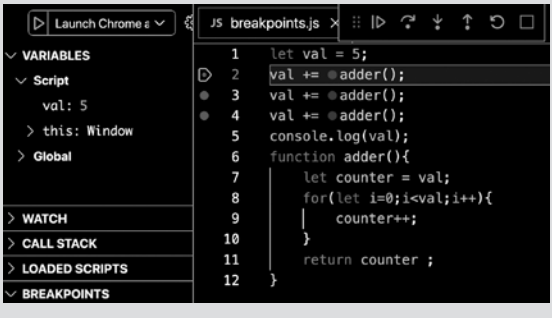


Рис. 12.5. Просмотр переменных в консоли

5. Нажмите значок воспроизведения еще раз — скрипт будет выполняться до тех пор, пока не достигнет следующей контрольной точки. Значение переменной обновится в результате выполнения кода в строке 2 (рис. 12.6).

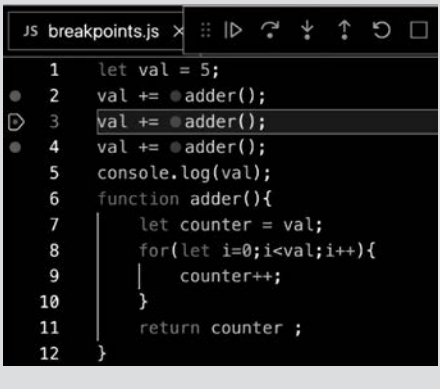


Рис. 12.6. Прохождение контрольных точек в скрипте

6. Нажмите кнопку воспроизведения снова, чтобы перейти к следующей контрольной точке, которая еще раз увеличивает значение `val` (рис. 12.7).

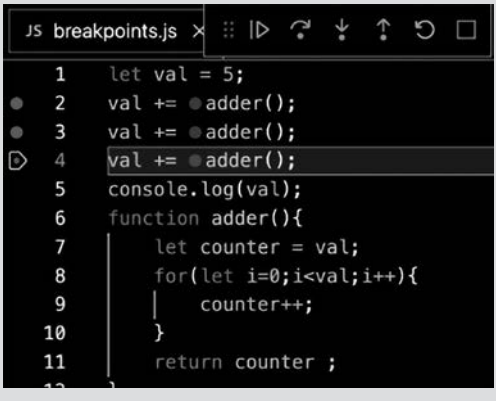


Рис. 12.7. Конечная контрольная точка

7. При достижении последней контрольной точки вам будут доступны только варианты перезапуска или остановки отладчика. Если нажмете «стоп», процесс отладки будет завершен (рис. 12.8).

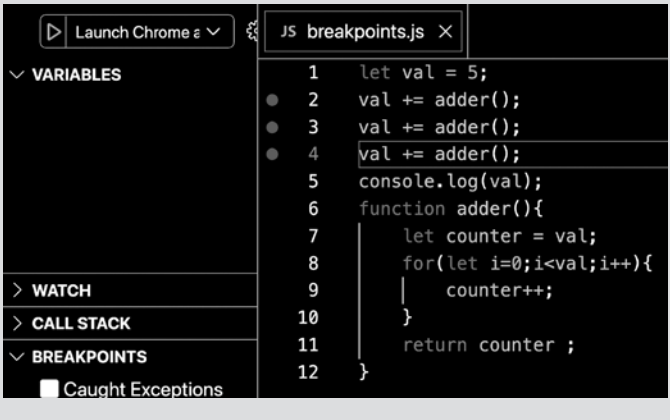


Рис. 12.8. Контрольные точки в браузере

Итоговое значение переменной `val` после третьей контрольной точки было определено как `135`. Запишите значения `val` после первого и второго вызовов функции `adder()`, которые вы увидели с помощью контрольных точек.

Мы рассмотрели базовое упражнение — приглашаем вас протестировать контрольные точки в более объемных скриптах, чтобы вы могли лучше разобраться в теме и понять, как работает ваш код во время выполнения.

Обработка ошибок

Мы уже видели множество ошибок. До сих пор мы позволяли программе, обнаружив ошибки, завершать работу. Но есть и другие пути борьбы с ними. Когда мы имеем дело с кодом, зависящим от какого-либо внешнего ввода (такого как API, пользовательский ввод или файл), нам нужно будет справиться с ошибками, которые этот ввод может вызвать.

Если есть вероятность, что определенный фрагмент кода выдаст ошибку, можно поместить этот код в блок `catch` — возможная ошибка кода будет там перехвачена.

Будьте аккуратны и не используйте такой способ слишком часто. Но вам и не придется этого делать, если сразу во время написания кода вы будете думать в первую очередь о том, как предотвратить возможные ошибки.

Перед вами пример кода, выдающего ошибку. Он окружен блоками `try` и `catch`. Предположим, что функция `somethingVeryDangerous()` может выдать ошибки:

```
try {
  somethingVeryDangerous();
} catch (e) {
  if (e instanceof TypeError) {
    // борьба с исключениями TypeError
  } else if (e instanceof RangeError) {
    // борьба с исключениями RangeError
  } else if (e instanceof EvalError) {
    // борьба с исключениями EvalError
  } else {
    // борьба со всеми остальными исключениями
    throw e; // rethrow
  }
}
```

Если выбрасывается ошибка, она попадет в блок `catch`. Поскольку `Error` может означать множество различных ошибок, мы собираемся проверить, с какой конкретно ошибкой имеем дело, и сделать ее специальную обработку. Точный класс ошибки устанавливается с помощью оператора `instanceof`. После обработки ошибок остальная часть кода продолжит выполняться в обычном режиме.

С помощью блоков `try` и `catch` можно сделать еще кое-что — добавить блок `finally`. Он выполняется независимо от того, выдаются ли ошибки, что отлично подходит для целей очистки. Вот простой пример:

```
try {
  trySomething();
} catch (e) {
  console.log("Oh oh");
} finally {
  console.log("Error or no error, I will be logged!");
}
```

Мы не знаем результат вывода, пока `trySomething()` не определена. Если бы код выдал ошибку, он бы сначала зафиксировал на экране `Oh oh`, а затем сообщил: `Error or no error, I will be logged!`. Если `trySomething()` не выбросит ошибку, на экране появится только последняя фраза.

Наконец, если по какой-то причине вам нужно выбросить ошибку, можете сделать это с помощью ключевого слова `throw`:

```
function somethingVeryDangerous() {
  throw RangeError();
}
```

Такой способ может быть очень полезен в случаях, когда вы имеете дело с неподконтрольными вам процессами: ответ API, пользовательский ввод или ввод из считываемого файла. Если случаются неожиданные вещи, иногда есть смысл выдать ошибку, чтобы соответствующим образом с ней справиться.

Практическое занятие 12.5

1. Используя `throw`, `try` и `catch`, проверьте, является ли значение числом. Если не является — создайте свою ошибку.
2. Напишите функцию с одним аргументом `val`.
3. Примените `try` и внутри него добавьте условие проверки `val` как числа. Проверьте `val` с помощью `isNaN`. Если условие истинно, выдайте ошибку, сообщающую, что это не число.
4. В обратном случае выведите на экран `Got a number`.
5. Используйте `catch` для захвата любых ошибок. Выведите их значения на экран.
6. Добавьте `finally` для запуска и вывода значения, а когда функция завершится, также включите туда значение `val`.
7. Создайте один запрос к функции со строковым аргументом, а другой — с числом. Просмотрите результаты в консоли.

Использование файлов cookie

Cookie (куки) — это небольшие файлы данных, которые хранятся на вашем компьютере и используются веб-сайтами. Файлы cookie были изобретены для хранения информации о пользователе веб-сайта. Куки представляют собой строки с особым паттерном. Они содержат пары «ключ — значение», разделенные точками с запятой.

Вы можете создавать такие файлы и использовать их в дальнейшем. Перед вами пример написания cookie:

```
document.cookie = "name=Maaike;favoriteColor=black";
```

Этот код при запуске на клиентской стороне (например, в вашем теге `<script>`) работает не во всех браузерах. Например, Chrome такого не позволяет, и код необходимо запускать с сервера. (Для реализации представленных здесь примеров мы использовали Safari, но нет никаких гарантий, что он будет и дальше поддерживать данный функционал.) Альтернативой являются API веб-хранилища.

В Chrome файлы cookie можно включить из командной строки, активизировав определенные настройки, или в настройках конфиденциальности. Но не забудьте отключить их, если они вам больше не понадобятся. Вот так можно считать информацию из cookie:

```
let cookie = decodeURIComponent(document.cookie);
let cookieList = cookie.split(";");
for (let i = 0; i < cookieList.length; i++) {
  let c = cookieList[i];
  if (c.charAt(0) == " ") {
    c = c.trim();
  }
  if (c.startsWith("name")) {
    alert(c.substring(5, c.length));
  }
}
```

Этот пример получает все куки-файлы с помощью `decodeURIComponent()`, а затем разделяет их знаком `;`. В итоге мы получаем массив `cookieList` с парами «ключ — значение» в виде строк. Далее мы перебираем все пары, обрезаем их (удаляем пробелы спереди и сзади) и проверяем, начинаются ли они с `name` — это было название нашего ключа cookie.

Если мы хотим получить значение, то должны начать его считывать после ключа. Длина ключа в данном случае — четыре символа (`name`). Это переносит нас к индексу 3. Стоит также пропустить знак равенства с индексом 4 — следовательно, начинаем с индекса 5. В этом случае мы добавляем к имени предупреждение. Вот пример простого веб-сайта, который использует файл cookie для приветствия пользователя:

```
<!DOCTYPE html>
<html>
  <body>
```

```
    <input onchange="setCookie(this)" />
    <button onclick="sayHi('name')">Let's talk, cookie!</button>
    <p id="hi"></p>
    <script>
      function setCookie(e) {
        document.cookie = "name=" + e.value + ";";
      }

      function sayHi(key) {
        let name = getCookie(key);
        document.getElementById("hi").innerHTML = "Hi " + name;
      }

      function getCookie(key) {
        let cookie = decodeURIComponent(document.cookie);
        let cookieList = cookie.split(";");
        for (let i = 0; i < cookieList.length; i++) {
          let c = cookieList[i];
          if (c.charAt(0) == " ") {
            c = c.trim();
          }
          if (c.startsWith(key)) {
            console.log("hi" + c);
            return c.substring(key.length + 1, c.length);
          }
        }
      }
    </script>
  </body>
</html>
```

Если вы пишете новый сайт, вам, вероятно, не следует применять этот метод. Однако всякий раз, когда вы будете обращаться к старому коду, скорее всего, вы столкнетесь с подобным. Теперь вы знаете, что это значит и как это настроить — повезло вам!

Практическое занятие 12.6

Давайте сделаем обработчик куки-файлов. Напишите несколько функций, которые позволят вам взаимодействовать с файлами cookie страницы, в том числе считывать их значения по имени, создавать новые файлы cookie с использованием имени, устанавливать их действие на определенное количество дней, а также удалять их. Можете использовать следующий HTML-шаблон для начала работы:

```
<!doctype html>
<html>
<head>
    <title>Complete JavaScript Course</title>
```

```
</head>
<body>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Настройте свою веб-страницу и выведите в JavaScript-коде значение `document.cookie`. Оно должно быть пустым.
2. Создайте функцию, которая будет получать параметры для `cookieName` и `cookieValue`, а также число дней, на которые вы хотите установить файл cookie.
3. Проверьте, действительно ли значение `days`, и внутри блока проверки задайте текущую дату. Установите значение `setTime` для срока истечения действия файла cookie в миллисекундах, умножив дни на миллисекунды.
4. Измените объект даты до истечения срока действия файла cookie в миллисекундах на строковое значение UTC.
5. Задайте `document.cookie` значение `cookieName = cookieValue`, добавьте детали истечения срока и в конце укажите `path=/`.
6. Напишите функцию для создания тестового файла cookie со значением и сроком действия, установленным на дату через несколько дней. Создайте таким же образом второй файл cookie — когда вы обновите свою страницу, в консоли должно отобразиться уже два файла.
7. Укажите вторую функцию для считывания значения файла cookie, установите ему значение `false`, а затем создайте массив файлов cookie, разделенных точками с запятой.
8. Пройдитесь по всем файлам cookie и снова разделите их там, где стоят знаки равенства. Это даст вам первый элемент с нулевым индексом — имя файла cookie. Добавьте условие, чтобы проверить, совпадает ли это имя с запрошенным в параметрах функции. Если все правильно, присвойте значение второму элементу в индексе, которое будет значением файла cookie с выбранным именем. Верните значение `cookievalue` в функцию.
9. Добавьте два сообщения лога, используя функцию для чтения обоих заданных вами файлов cookie. Выведите значения файлов cookie в консоль.
10. Чтобы удалить файл cookie, необходимо установить ему срок действия, предшествующий текущей дате. Можете создать файл cookie с датой `-1` и отправить его для удаления, вызвав функцию создания cookie.
11. Попробуйте удалить куки-файл по имени.

Локальное хранилище

Мы рассматривали файлы cookie как способ хранения пользовательских данных, но на самом деле для решения этой задачи есть более актуальный способ: *локальное хранилище*. Локальное хранилище — это удивительная и забавная тема, благодаря изучению которой ваши возможности по созданию интеллектуальных сайтов расширятся. С помощью локального хранилища можно сохранять пары «ключ — значение» в браузере и использовать их в новом сеансе (когда браузер будет открыт снова). Информация обычно хранится в папке на компьютере пользователя, а в какой именно — зависит от браузера.

Это позволяет веб-сайту сохранять нужную информацию и извлекать ее даже после обновления страницы или закрытия браузера. Преимущество локального хранилища перед файлами cookie заключается в том, что файлы оттуда не нужно передавать при каждом HTTP-запросе, как в случае с куки: локальное хранилище просто находится у пользователя и ждет доступа.

Объект `localStorage` принадлежит объекту `window`, который мы рассматривали ранее. Для эффективного использования `localStorage` необходимо знать несколько его методов. Прежде всего, нужно иметь возможность получать и задавать пары «ключ — значение» в локальном хранилище. Мы применим `setItem()` везде, где требуется сохранить данные, и `getItem()`, когда позже будем извлекать значение. Вот как это сделать:

```
<!DOCTYPE html>
<html>
  <body>
    <div id="stored"></div>
    <script>
      let message = "Hello storage!";
      localStorage.setItem("example", message);

      if (localStorage.getItem("example")) {
        document.getElementById("stored").innerHTML =
          localStorage.getItem("example");
      }
    </script>
  </body>
</html>
```

Этот фрагмент кода выводит на страницу надпись `Hello storage!`. Вносить элементы в хранилище можно с помощью метода `setItem`, указав их ключ и значение. Доступ к `localStorage` осуществляется напрямую или через объект `window`. Здесь мы указали `example` в качестве ключа, а `Hello storage!` — в качестве значения. Затем занесли данные в локальное хранилище. После этого проверили, задан ли ключ `example` в локальном хранилище, и вывели данные, записывая их в `innerHTML` объекта `div` с идентификатором `stored`.

Если перед повторной загрузкой страницы удалить в коде строку `setItem()`, программа все равно выведет это значение, поскольку информация была сохранена при первом запуске скрипта и никогда не удалялась. Срока давности у данных в локальном хранилище не существует, но можно их удалить вручную.

Мы также можем извлечь ключ, используя индекс. Это очень полезно, когда необходимо перебрать пары «ключ — значение», но названия ключей нам неизвестны. Вот как извлечь ключ по индексу:

```
window.localStorage.key(0);
```

В этом случае ключ — `name`. Чтобы получить соответствующее ключу значение, мы можем сделать так:

```
window.localStorage.getItem(window.localStorage.key(0));
```

Кроме того, мы можем удалять пары «ключ — значение»:

```
window.localStorage.removeItem("name");
```

Все пары «ключ — значение» можно удалить из локального хранилища одной командой:

```
window.localStorage.clear();
```

Таким образом, значения в локальном хранилище сохраняются даже после закрытия браузера. Это обеспечивает работу множества «умных» функций: теперь ваше приложение способно запоминать, что вы ввели в форму, какие настройки переключили на веб-сайте и что просматривали ранее.

Тем не менее не рассматривайте локальное хранилище как альтернативу, способную обойти проблемы с cookie и конфиденциальностью. Оно, хотя и не так известно, как куки-файлы, вызывает те же проблемы. Вам все равно придется указать на своем сайте, что вы отслеживаете пользователей и храните информацию о них.

Практическое занятие 12.7

Создадим список покупок, значения которого будут помещаться в локальное хранилище браузера. Это пример использования JavaScript для преобразования строк в объекты, пригодные для JavaScript, и обратно в строки, которые могут располагаться в локальном хранилище. Можете взять следующий шаблон:

```
<!doctype html>
<html>
<head>
    <title>JavaScript</title>
```

```
    <style>
        .ready {
            background-color: #ddd;
            color: red;
            text-decoration: line-through;
        }
    </style>
</head>
<body>
    <div class="main">
        <input placeholder="New Item" value="test item" maxlength="30">
        <button>Add</button>
    </div>
    <ul class="output">
    </ul>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. В JavaScript-коде выберите все элементы страницы как объекты JavaScript.
2. Создайте массив `tasks`, содержащий значение локального хранилища `tasklist`, если оно существует. В противном случае задайте пустой массив `tasks`. Можете конвертировать строковые значения в пригодный для JavaScript объект, используя `JSON.parse`.
3. Переберите элементы в массиве `tasklist`. Они будут сохранены как объекты с именем и логическим значением их проверенного статуса. Напишите отдельную функцию для создания элемента `task`, добавив его на страницу списка.
4. В функции создания `task` добавьте новый элемент списка и `textNode`. Отправьте `textNode` в список элементов. Добавьте элемент списка в область вывода на странице. Если логическое значение `task` равно `true`, тогда добавьте класс `style` в значении `ready`.
5. Добавьте прослушиватель событий в элемент списка, который при нажатии переключит значение класса на `ready`. Каждый раз при изменении какого-либо элемента списка необходимо будет сохранять его в локальном хранилище. Создайте для задачи функцию, которая будет сохранять данные и обеспечивать совпадение отображаемого списка со списком локального хранилища. Нужно будет очистить текущий массив списка задач и заново сформировать его из визуальных данных — для управления процессом построения списка создайте функцию.

6. Созданная функция очистит текущий массив `tasks` и выделит все элементы `li` на странице. Перебираем все элементы списка, получаем их текстовые значения и проверяем, равно ли значение их класса `ready`. Если это так, помечаем условие проверки как `true`. Добавьте результаты в массив `tasks` — это приведет к перестройке массива и проверке его соответствия тому, что пользователь видит на экране. Чтобы сохранить массив `tasks` в локальном хранилище, отправьте данные в функцию сохранения задач. Как результат, если страница будет обновлена, вы увидите тот же список.
7. В функции сохранения задач отправьте элемент `localstorage` в массив задач. Вам нужно будет преобразовать объект в строку, чтобы он мог войти в строковый параметр локального хранилища.
8. Теперь, обновив страницу, вы увидите список задач. Пункт списка можно вычеркнуть, щелкнув на нем кнопкой мыши. А новые элементы можно добавить, заполнив поле ввода и нажав кнопку отправки.

JSON

JSON расшифровывается как *JavaScript Object Notation* и является не чем иным, как форматом данных. Мы встречали это обозначение, когда создавали наши объекты в JavaScript. Однако JSON не равно «объекты JavaScript» — это просто способ представления данных в том же формате, что и объекты JavaScript, который, впрочем, может быть в такой объект легко преобразован.

JSON — это стандарт, используемый для взаимодействия с API, включая даже те API, которые не написаны на JavaScript! API могут принимать данные (например, из формы на веб-сайте) в формате JSON. В настоящее время API почти всегда отправляют данные в JSON: это происходит, например, при входе в интернет-магазин — товары обычно отображаются благодаря вызову API, подключенного к базе данных. В подобном случае полученные данные преобразуются в JSON и отправляются обратно на веб-сайт. Вот пример использования JSON:

```
{
  "name" : "Malika",
  "age" : 50,
  "profession" : "programmer",
  "languages" : ["JavaScript", "C#", "Python"],
  "address" : {
    "street" : "Some street",
    "number" : 123,
    "zipcode" : "3850AA",
    "city" : "Utrecht",
    "country" : "The Netherlands"
  }
}
```

Это объект, который, по-видимому, описывает человека. В нем есть пары «ключ — значение». Ключи всегда должны указываться в кавычках; значения же включаются в кавычки только в том случае, если являются строками. Итак, первый ключ — `name`, а первое значение — `Malika`.

Списки значений (или JavaScript-массивы) обозначены скобками `[]`. Объект JSON содержит в себе список `languages`, указанный в квадратных скобках, а также другой объект: `address` — это можно определить по фигурным скобкам.

На самом деле в JSON есть только несколько вариантов содержимого:

- пары «ключ — значение» со значениями следующих типов: `string`, `number`, `Boolean` и `null`;
- пары «ключ — значение» со списками, обозначенными скобками `[ ]` и содержащими внутри свои элементы;
- пары «ключ — значение» с другими объектами, обозначенными скобками `{ }` и содержащими внутри другие элементы JSON.

Эти варианты можно объединять. Объект может содержать другие объекты, а список — другие списки: мы уже видели подобное в примере выше, где в нашем объекте был вложенный объект адреса.

Вложений может быть намного больше: список может содержать объекты, которые могут содержать списки с объектами, списки со списками и т. д. Выглядит это немного путано — и в этом-то и суть. Несмотря на всю простоту JSON, вложения все равно могут его усложнить. Поэтому мы и поместили его здесь, в главе с продвинутыми темами.

Рассмотрим немного более сложный пример:

```
{
    "companies": [
        {
            "name": "JavaScript Code Dojo",
            "addresses": [
                {
                    "street": "123 Main street",
                    "zipcode": 12345,
                    "city" : "Scott"
                },
                {
                    "street": "123 Side street",
                    "zipcode": 35401,
                    "city" : "Tuscaloosa"
                }
            ]
        },
        {
```

```
            "name": "Python Code Dojo",
            "addresses": [
                {
                    "street": "123 Party street",
                    "zipcode": 68863,
                    "city" : "Nebraska"
                },
                {
                    "street": "123 Monty street",
                    "zipcode": 33306,
                    "city" : "Florida"
                }
            ]
        }
    ]
}
```

Перед вами список компаний, содержащий два объекта `company`. У компаний есть две пары «ключ — значение», содержащие имя и список адреса. В каждом из адресных списков находится по два адреса, состоящих из трех пар «ключ — значение»: `street`, `zipcode` и `city`.

Практическое занятие 12.8

Следующее упражнение продемонстрирует, как создать объект JSON, который впоследствии вы используете в качестве объекта JavaScript. Сначала вы напишете простой список имен и статусов, который можно будет перебирать и выводить результаты на экран. Далее вы загрузите данные JSON в JavaScript и отобразите содержимое объекта.

1. Создайте JavaScript-объект с данными в формате JSON. Он должен содержать по крайней мере два элемента, каждый из которых является объектом с как минимум двумя парными значениями.
2. Напишите функцию, которая при вызове будет перебирать каждый элемент JSON-объекта в JavaScript и выводить результат в консоль. Отображайте каждый элемент на экране с помощью `console.log`.
3. Вызовите функцию и запустите JavaScript-код.

Парсинг JSON

Существует множество библиотек и инструментов преобразования строки JSON в объект. Строка JavaScript может быть преобразована в объект JSON с помощью функции `JSON.parse()`. Данные, полученные из другого источника, всегда имеют

значение типа `string`, поэтому, чтобы работать с ними как с объектом, их необходимо преобразовать. Вот как это делается:

```
let str = "{\"name\": \"Maaike\", \"age\": 30}";
let obj = JSON.parse(str);
console.log(obj.name, "is", obj.age);
```

После парсинга строку можно трактовать как объект. Следовательно, на экран будет выведено `Maaike is 30`.

Иногда нужно сделать и наоборот: преобразовать объект JavaScript в строку JSON. Сделать это можно с помощью функции `JSON.stringify()`. Рассмотрим ее в действии:

```
let dog = {
    "name": "wiesje",
    "breed": "dachshund"
};
let strdog = JSON.stringify(dog);
console.log(typeof strdog);
console.log(strdog);
```

К `strdog` применяется метод `stringify()` — и объект становится строкой. У нее больше нет свойств `name` и `breed` — они будут неопределенными. В результате на экране появится следующая запись:

```
string
{"name":"wiesje","breed":"dachshund"}
```

Парсинг JSON может быть полезен, например, для хранения данных JSON непосредственно в базе данных.

Практическое занятие 12.9

Закрепим использование методов JSON для парсинга JSON и преобразования строковых значений в JSON.

1. Создайте JSON-объект с несколькими элементами и объектами (можете использовать JSON-объект из предыдущего урока).
2. Используя метод `stringify()`, преобразуйте в JavaScript JSON-объект в строковую версию и присвойте его переменной `newStr` `[{"name":"Learn JavaScript","status":true},{"name":"Try JSON","status":false}]`.
3. Используя `JSON.parse()`, преобразуйте значение `newStr` обратно в объект и присвойте его переменной `newObj`.
4. Выполните итерацию по элементам в `newObj` и выведите результаты на экран.

Проекты текущей главы

Сборщик адресов электронной почты

Используйте следующий HTML-код в качестве шаблона и добавьте JavaScript-код для создания функции извлечения адресов электронной почты:

```
<!doctype html>
<html>
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
    <textarea name="txtarea" rows=2 cols=50></textarea> <button>Get Emails</
button>
    <textarea name="txtarea2" rows=2 cols=50></textarea>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. В JavaScript-коде выберите обе текстовые области и кнопку как объекты JavaScript.
2. Добавьте прослушиватель событий к вызывающей функцию кнопке, которая получает содержимое первой области `textarea` и фильтрует его, чтобы принимать только адреса электронной почты.
3. В функции извлечения получите содержимое первого поля ввода. Используя `match()`, верните массив адресов электронной почты, которые были сопоставлены с содержимым первой области `textarea`.
4. Чтобы удалить все дубликаты, создайте отдельный массив для уникальных значений.
5. Просмотрите все найденные адреса электронной почты и проверьте, есть ли каждый из них в массиве `holder`; если нет — добавьте адрес в массив.
6. Используя метод массива `join()`, объедините найденные адреса электронной почты и выведите их во вторую область `textarea`.

Валидатор форм

Данный проект — пример типичной структуры формы, где вы проверяете введенные значения и утверждаете их перед отправкой. Если значения не соответствуют критериям проверки, пользователю возвращается соответствующий

отклик. В качестве исходного шаблона можете использовать следующий HTML- и CSS-код:

```
<!doctype html>
<html>
<head>
    <title>JavaScript Course</title>
    <style>
        .hide {
            display: none;
        }
        .error {
            color: red;
            font-size: 0.8em;
            font-family: sans-serif;
            font-style: italic;
        }
        input {
            border-color: #ddd;
            width: 400px;
            display: block;
            font-size: 1.5em;
        }
    </style>
</head>
<body>
    <form name="myform"> Email :
        <input type="text" name="email"> <span class="error hide"></span>
        <br> Password :
        <input type="password" name="password"> <span class="error hide"></span>
        <br> User Name :
        <input type="text" name="userName"> <span class="error hide"></span>
        <br>
        <input type="submit" value="Sign Up"> </form>
    <script>

    </script>
</body>
</html>
```

Выполните следующие шаги.

1. Используя JavaScript, выделите все элементы страницы и обозначьте их как объекты JavaScript, чтобы их было легче выбирать в коде. Выберите также все элементы страницы с классом `error` в качестве объекта.
2. Добавьте прослушиватель событий для отправки и перехвата щелчка кнопкой мыши, предотвращая действие формы по умолчанию.
3. Переберите все элементы страницы с классом `error` и добавьте класс `hide`, который удалит их из виду (раз это новая попытка отправки формы).

4. Используя регулярное выражение, проверьте правильность значения, введенного в поле электронной почты.
5. Для реагирования на ошибки создайте функцию, которая удаляет класс `hide` из элемента рядом с тем элементом, который вызвал событие. Примените фокусировку к этому элементу внутри функции.
6. Если введенные данные не соответствуют заданному регулярному выражению, передайте параметры только что созданной функции обработки ошибок.
7. Проверьте значение поля ввода пароля, чтобы убедиться, что там используются только буквы и цифры. Также проверьте длину: она должна составлять 3–8 символов. Если какое-либо из условий имеет значение `false`, с помощью функции `error` добавьте ошибку и напишите сообщение для пользователя. После этого установите для `error` логическое значение `true`.
8. Добавьте объект, чтобы отслеживать заполнение формы, и значения к нему, перебирая все входные данные, устанавливая свойствам такие же имена, как и вводу, а значения — аналогичные вводимым.
9. Перед завершением работы функции проверьте, есть ли ошибки. Если их нет, подтвердите отправку формы.

Простой математический опросник

В следующем проекте мы создадим простую математическую викторину. Приложение даст возможность пользователю ответить на вопросы, проверит эти ответы и оценит их точность. Можете использовать представленный HTML-шаблон:

```
<!doctype html>
<html>
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
    <span class="val1"></span> <span>+</span> <span class="val2"></span> = <span>
        <input type="text" name="answer"></span><button>Check</button>
    <div class="output"></div>
</body>
</html>
```

Выполните следующие шаги.

1. Поместите код JavaScript в функцию `app`. В функции создайте объекты переменных со всеми элементами страницы (чтобы их можно было использовать в скрипте) и пустой объект `game`.
2. Добавьте прослушиватель событий `DOMContentLoaded`, вызывающий инициализацию функции `app` после загрузки страницы.

3. В функции `init()` добавьте прослушиватель событий к кнопке. Прослушивайте щелчок и отслеживайте событие в функции `checker`. Вместе с функцией `init` загрузите и другую функцию — `loadQuestion()`.
4. Создайте одну функцию для загрузки вопросов и другую для генерирования случайного числа из минимальных и максимальных значений в аргументах.
5. Сгенерируйте два случайных значения в функции `loadQuestion()` и включите их в объект `game`. Вычислите результат сложения обоих значений и также присвойте это значение игровому объекту.
6. Назначьте и обновите `textContent` элементов страницы, которым требуются динамические значения чисел для вопроса с вычислением.
7. При нажатии кнопки используйте тернарный оператор, чтобы определить, был ли ответ на вопрос правильным или нет. Установите цвет `green` для правильного ответа и `red` для неправильного.
8. Создайте на странице элемент для вывода всех вопросов и отслеживания результатов. В функции `checker()` добавьте новый HTML-элемент с цветом стиля, которым будут выделены правильные или неправильные ответы. Отобразите первое и второе значения, а также ответ и покажите результат пользователя в скобках.
9. Очистите поле ввода и загрузите следующий вопрос.

Вопросы для самопроверки

1. Что вернет следующее регулярное выражение?

```
Expression / ([a-e])\w+/g
"Hope you enjoy JavaScript"
```

2. Являются ли куки-файлы частью объекта `document`?
3. Что следующий код будет делать с файлом cookie в JavaScript?

```
const mydate = new Date();
mydate.setTime(mydate.getTime() - 1);
document.cookie = "username=; expires=" + mydate.toGMTString();
```

4. Что будет выведено на экран при исполнении следующего кода?

```
const a = "hello world";
(function () {
    const a = "JavaScript";
})();
console.log(a);
```

5. Что отобразится на экране в результате выполнения данного кода?

```
<script>
"use strict";
```

```
myFun();
console.log(a);
function myFun() {
    a = "Hello World";
}
</script>
```

6. Что вы увидите в консоли после выполнения этого кода?

```
console.log("a");
setTimeout(() => {
    console.log("b");
}, 0);
console.log("c");
```

Резюме

В этой главе появились более важные и сложные темы, к которым вы, вероятно, не были готовы в начале книги. Как результат, вы углубили свое понимание JavaScript в нескольких областях — в первую очередь в регулярных выражениях. С помощью регулярных выражений мы можем назначать шаблоны строк и использовать их для поиска совпадений в других строках.

Далее мы рассмотрели функции и объект `arguments`, с помощью которого можно получить доступ к аргументам по их индексу. Затем вы изучили поднятие JavaScript и строгий режим, который позволяет задавать JavaScript немного больше условий. Использование JavaScript в строгом режиме, как правило, хорошая практика и отличный способ подготовиться к работе с фреймворками JavaScript.

Мы также обсудили отладку: контрольные точки или отображение выходных данных в браузерной консоли дадут представление о происходящих процессах, а правильная обработка ошибок поможет предотвратить ненужные сбои программы. Наконец, мы рассмотрели создание файлов cookie в JavaScript и использование локального хранилища, а также JSON — синтаксиса для передачи данных. Мы познакомились с различными типами пар «ключ — значение» и способами парсинга JSON. Мы также изучили, как хранить пары «ключ — значение» в `localStorage` объекта `window`.

Благодаря материалу этой главы вы еще лучше разобрались в JavaScript и узнали новые приемы, подходящие как для работы с современным JavaScript-кодом, так и со старым (устаревшим). В следующей главе мы углубимся в еще более сложную тему: параллелизм. Там вы узнаете все о многозадачности вашего JavaScript-кода.

13 Параллелизм

Настало время более продвинутой темы — вы готовы к этому! Мы будем работать с асинхронным кодом и рассмотрим некоторые возможности многозадачности. Данная концепция называется *параллелизмом*. Не волнуйтесь, если эта глава покажется вам немного трудной, ведь мы переходим к продвинутому программированию в JavaScript. Далее мы рассмотрим следующие темы:

- параллелизм;
- функции обратного вызова;
- промисы;
- операторы `async`/`await`;
- цикл событий.

Да, это сложно, но параллелизм действительно может ускорить процессы в вашей программе и повысить ее производительность. Это более чем достаточная причина погрузиться в такую продвинутую тему!

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Введение в параллелизм

Параллелизм (многопоточность) — это концепция, согласно которой процессы в JavaScript происходят «в одно и то же время», или *параллельно*. Рассмотрим пример, не связанный с кодом, и поговорим об управлении домашним хозяйством. Когда я прихожу домой в пятницу вечером, я должен сделать ряд задач: накормить, выкупать и уложить спать детей, загрузить в стиральную машину белье — и, честно говоря, еще много всего. Если бы я не мог делать несколько дел одновременно, это был бы очень тяжелый и долгий вечер. Сначала приготовить ужин — поставить пиццу в духовку и подождать ее готовности. Потом накормить детей, искупать и уложить их спать. А после сложить чистое белье, загрузить его, включить стиральную машину и подождать, пока она отработает. К счастью, я обладаю навыком

многозадачности, так что вечер я проведу примерно так: ставлю пиццу в духовку, тем временем загружаю стиральную машину, включаю ее и складываю чистое белье; после этого кормлю детей, разбираюсь со стиркой, пока дети в душе, — и в итоге освобождаюсь даже раньше запланированного времени.

То же самое касается вашего компьютера и используемых приложений. Если бы он не мог выполнять несколько задач одновременно, вы, вероятно, были бы очень раздражены. Вы не могли бы проверять почту или слушать музыку во время написания кода — и многое другое. И все же ваш компьютер умеет переключаться между различными задачами. То же самое можно реализовать на уровне приложения. Например, мы можем выполнить вызов API и, пока ждем ответ, сделать что-то полезное. Это можно осуществить как раз благодаря концепции параллелизма.

В JavaScript есть три стратегии, о которых вам необходимо знать при работе с параллелизмом: *функции обратного вызова*, *промисы* и ключевые слова `async` и `await`.

Функции обратного вызова

Функции обратного вызова (колбэки) — это первое, что мы должны понимать, когда говорим о параллелизме. Хорошая новость заключается в том, что в них достаточно легко разобраться. Функция `callback` принимает другую функцию в качестве аргумента, который затем вызывается, когда остальная часть первой функции завершена. Другими словами, это просто функция, вызывающая функцию:

```
function doSomething(callback) {
    callback();
}

function sayHi() {
    console.log("Hi!");
}

doSomething(sayHi);
```

Функция `doSomething()`, созданная с параметром `callback`, — это всего лишь вызов любой функции, которая передается в качестве аргумента. В нашем случае этим аргументом является функция `sayHi()`. Таким образом, данный фрагмент кода — просто очень сложный способ добиться вывода `Hi!` на экран.

Вот пример `callback`, который действительно что-то делает:

```
function judge(grade) {
    switch (true) {
        case grade == "A":
            console.log("You got an", grade, ": amazing!");
            break;
        case grade == "B":
            console.log("You got a", grade, ": well done!");
```

```
            break;
        case grade == "C":
            console.log("You got a", grade, ": alright.");
            break;
        case grade == "D":
            console.log("You got a", grade, ": hmmm...");
            break;
        default:
            console.log("An", grade, "! What?!");
    }
}

function getGrade(score, callback) {
    let grade;
    switch (true) {
        case score >= 90:
            grade = "A";
            break;
        case score >= 80:
            console.log(score);
            grade = "B";
            break;
        case score >= 70:
            grade = "C";
            break;
        case score >= 60:
            grade = "D";
            break;
        default:
            grade = "F";
    }
    callback(grade);
}

getGrade(85, judge);
```

Здесь находятся две функции: `judge()` и `getGrade()`. Мы вызываем функцию `getGrade()` с двумя аргументами: `85` и функцией `judge()`. Обратите внимание, что при вызове функции в качестве аргумента мы не дописываем ей круглые скобки в конце. Функция `judge()` сохраняется в `callback`. После определения оценки функция, хранящаяся в обратном вызове (в данном случае `judge()`), вызывается со значением оценки.

Данную функцию также можно применить для чего-то более полезного, чем просто оценка: например, для отправки электронного письма с определенным текстом в зависимости от результатов тестирования. В этом случае нам бы не пришлось изменять функцию `getGrade()`: мы бы просто написали для данной задачи новую функцию и вызвали `getGrade()` с новой функцией в качестве второго аргумента.

Возможно, вы разочарованы, ведь пока все это выглядит не очень увлекательно. Функции обратного вызова становятся действительно ценными в асинхронном

контексте: например, когда одна функция все еще ожидает результатов вызова базы данных перед вызовом функции `callback`, которая будет эти данные обрабатывать.

С обратными вызовами работают некоторые встроенные JavaScript-функции — например, `setTimeout()` и `setInterval()`. Они будут принимать функцию, запускающуюся сразу после того, как указанное время истекло, или через каждый определенный промежуток времени в течение указанного интервала. Мы уже встречались с таким кодом:

```
setInterval(encourage, 500);

function encourage() {
    console.log("You're doing great, keep going!");
}
```

Функции, которые здесь вставляются в качестве аргументов, называются колбэками или функциями обратного вызова. Параллелизм действительно начинается с обратных вызовов, но несколько вложенных обратных вызовов затрудняют чтение кода.

Когда все это записывается как одна функция с анонимными функциями внутри, в коде появляется слишком много последовательных отступов. Мы называем это *callback hell* (ад обратных вызовов) или *Christmas tree problem* (проблема рождественской ели): код оказывается вложенным столько раз, что по своему виду напоминает ель.

Функции обратного вызова — отличная концепция, но они очень быстро могут сделать код уродливым. Обещаем, что поможем найти лучшее решение.

Практическое занятие 13.1

Это упражнение продемонстрирует, как использовать функцию обратного вызова при передаче значения из одной функции в другую. Мы создадим функцию `callback` для приветствия, отображающего полное имя пользователя в строке.

1. Создайте функцию `greet()`, которая принимает один аргумент, `fullName`. Этот аргумент должен быть массивом. Выведите элементы массива на экран, интерполированные в строку приветственного сообщения.
2. Создайте вторую функцию, которая имеет два аргумента: первый — строка для полного имени пользователя, а второй — функция `callback`.
3. Разбейте строку на массив, используя метод `split()`.
4. Отправьте массив полных имен в функцию `greet()`, созданную на первом шаге.
5. Вызовите функцию `callback`.

Промисы

С помощью промисов мы можем организовать наш код немного более простым для обслуживания способом. Промис — это специальный объект, соединяющий код, который должен выдать результат, и код, который должен использовать этот результат на следующем шаге.

При создании промиса мы даем ему функцию. В следующем примере мы используем уже знакомое вам правило: писать функцию на месте. Поэтому мы определяем функцию внутри списка аргументов, часто с помощью стрелочных функций. Этой функции нужны два параметра — и эти параметры являются обратными вызовами. В данном примере мы назвали их `resolve` и `reject`.

Вы можете назвать эти параметры как угодно, но общепринятыми являются resolve (или res) и reject (или rej).

Когда вызвана функция `resolve()`, промис считается успешным — поэтому все, что находится между стрелками, возвращается и используется в качестве входных данных для метода `then` объекта `Promise`. Если вызвана функция `reject()` — промис не удался, поэтому метод `catch()` объекта `Promise` (если и был добавлен) будет выполнен с аргументом функции `reject()`.

Наверняка очень многая информация здесь вам непонятна, поэтому вот пример промиса, который должен помочь:

```
let promise = new Promise(function (resolve, reject) {
    // сделайте что-нибудь, что может занять некоторое время
    // давайте просто зададим x для этого примера
    let x = 20;
    if (x > 10) {
        resolve(x); // в случае успеха
    } else {
        reject("Too low"); // в случае ошибки
    }
});

promise.then(
    function (value) {
        console.log("Success:", value);
    },
    function (error) {
        console.log("Error:", error);
    }
);
```

Сначала мы создаем промис. При написании `Promise` мы еще не знаем, каким будет его значение, но точно знаем, что оно равно значению, переданному в качестве аргумента функции `resolve`. Это своего рода заглушка (placeholder).

Поэтому при вызове промиса мы обычно говорим: «Определите значение промиса и, когда оно станет известно, инициируйте одну функцию, если промис был выполнен, или другую функцию, если был отклонен. Когда промис и не выполнен, и не отклонен, мы считаем, что промис находится на рассмотрении».

`then()` само по себе является промисом, поэтому, когда оно возвращается, мы можем использовать полученный результат для следующего экземпляра `then()`. Следовательно, можно связать экземпляры `then()` в цепочку такого вида:

```
const promise = new Promise((resolve, reject) => {
    resolve("success!");
})
    .then(value => {
        console.log(value);
        return "we";
    })
    .then(value => {
        console.log(value);
        return "can";
    })
    .then(value => {
        console.log(value);
        return "chain";
    })
    .then(value => {
        console.log(value);
        return "promises";
    })
    .then(value => {
        console.log(value);
    })
    .catch(value => {
        console.log(value);
    })
```

Результат будет следующим:

```
success!
we
can
chain
promises
```

Функции `resolve` реализованы с помощью стрелочной функции. Оператор `return` — это входящее значение `value` для следующей функции. Как видите, следующий блок представляет собой функцию `catch()`. Если какая-либо из функций приведет к отказу и, следовательно, промис будет отклонен, блок `catch()` будет выполнен и выведет все, что функция `reject()` отправила в метод `catch()`. Например:

```
const promise = new Promise((resolve, reject) => {
    reject("oops... ");
})
```

```
.then(value => {
    console.log(value);
    return "we";
})
.then(value => {
    console.log(value);
    return "can";
})
.then(value => {
    console.log(value);
    return "chain";
})
.then(value => {
    console.log(value);
    return "promises";
})
.then(value => {
    console.log(value);
})
.catch(value => {
    console.log(value);
})
```

Данный код просто выдаст `oops...`, потому что первый промис был отклонен, а не выполнен. Такой способ отлично подходит для создания асинхронных процессов, которым необходимо дождаться завершения другого процесса. Мы можем попытаться выполнить определенный набор действий и, когда что-то пойдет не так, использовать метод `catch()`, чтобы справиться с этим.

Практическое занятие 13.2

В этом упражнении вы создадите счетчик, который с помощью промисов будет последовательно выводить значения.

1. Настройте промис, который разрешается со значением `Start Counting`.
2. Создайте функцию `counter()` с одним аргументом, который получает значение и выводит его на экран.
3. Настройте в промисе следующую функцию с четырьмя экземплярами `then()`, которые должны выводить значение в функцию счетчика и возвращать значение, обеспечивающее ввод для последующего экземпляра `then()`. Возвращенные значения должны быть `one`, `two`, `three`. Вывод в консоли получится следующим:

```
Start Counting
One
Two
Three
```

Операторы async и await

Только что мы рассмотрели синтаксис `Promise`. Функция может вернуть промис с помощью ключевого слова `async`. Это делает промисы более удобными для чтения и по виду очень похожими на синхронный (непараллельный) код. Использование промисов в этом случае будет точно таким же, как в предыдущем разделе, или же мы можем прибегнуть к более мощному ключевому слову `await`, чтобы дождаться выполнения промиса. `await` работает только в асинхронной функции.

В асинхронном контексте можно ожидать и другие промисы, как видно из следующего примера:

```
function saySomething(x) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve("something" + x);
        }, 2000);
    });
}

async function talk(x) {
    const words = await saySomething(x);
    console.log(words);
}

talk(2);
talk(4);
talk(8);
```

Можете рассказать, что этот код выполняет? Мы вызываем асинхронную функцию `talk()` в строке трижды. Каждый из этих вызовов ожидает функцию `saySomething()`. В функции `saySomething()` содержится новый промис, который разрешается с помощью функции `setTimeout()`, ожидающей в течение двух секунд, прежде чем реализоваться со значением `something + x`. Таким образом, через две секунды все три функции выполнятся одновременно (по крайней мере так кажется человеческому глазу).

Если функция `talk()` не будет асинхронной, она выдаст ошибку `SyntaxError` из-за ключевого слова `await`. Ключевое слово `await` действительно только в асинхронных функциях, поэтому функция `talk()` должна быть асинхронной. Без `async` и `await` код будет хранить результат функции `saySomething()`, ожидающий `Promise`, и выводить его каждый раз при вызове функции:

```
Promise { <pending> }
Promise { <pending> }
Promise { <pending> }
```

На данный момент мы рассмотрели все основные строительные блоки параллелизма. Полученных знаний должно хватить для начала работы с данной концепцией в реальных проектах. Это действительно продвинутая тема. Отлаживать параллельный код

хлопотно, но, применив его в нужный момент и получив прекрасные результаты с точки зрения продуктивности, вы поймете, что все усилия того стоили.

Практическое занятие 13.3

Следующее упражнение продемонстрирует, как применять `await` для ожидания `Promise` внутри функции `async`. Используя `await` и `async`, создайте таймер обратного отсчета `timeout()` и увеличьте значение глобального счетчика.

1. Создайте глобальную переменную для таймера.
2. Создайте функцию, принимающую один аргумент. Верните результат нового промиса, устанавливая функцию `setTimeout()`, которая будет содержать экземпляр разрешения.
3. Увеличивайте значение таймера внутри `setTimeout()`, каждую секунду добавляя единицу. Разрешите промис с помощью значений таймера и переменной, которая была аргументом функции.
4. Создайте асинхронную функцию, которая выводит на экран значение глобального счетчика и значение аргумента функции.
5. Создайте переменную для захвата возвращаемого значения разрешения из функции `await`. Выведите результаты на экран.
6. Создайте цикл десяти итераций, увеличивая значение и вызывая функцию `async`. Передавайте значение увеличиваемой переменной в функции в качестве параметра.

Результат должен выглядеть следующим образом:

```
ready 1 counter:0
ready 2 counter:0
ready 3 counter:0
x value 1 counter:1
x value 2 counter:2
x value 3 counter:3
```

Цикл событий

Мы хотели бы закончить главу объяснением того, как в JavaScript обрабатывается асинхронность и параллелизм. JavaScript — это однопоточный язык. Поток в данном контексте означает *путь выполнения*. Если существует только один путь, задачам придется ждать выполнения друг друга, следовательно, одновременно может произойти только одно событие.

Эта однопоточность обеспечивается *циклом событий* — процессом, который выполняет актуальные задачи. Возможно, сейчас вы растеряны: мы же только что говорили о параллелизме и выполнении асинхронных действий в одно и то же время! Что ж, JavaScript действительно является однопоточным языком — но это не значит, что он не может передавать некоторые задачи на аутсорсинг. Именно так JavaScript удается выполнять многопоточные действия.

Стек вызовов и очередь обратных вызовов

JavaScript работает со *стеком вызовов*, и все действия, которые он должен выполнить, помещаются в эту очередь. Цикл событий — процесс, который постоянно отслеживает стек вызовов, и всякий раз, когда нужно выполнить задачи, реализует их одну за другой. Задачи, расположенные сверху, выполняются в первую очередь.

Вот крошечный скрипт:

```
console.log("Hi there");
add(4,5);

function add(x, y) {
    return x + y;
}
```

На рис. 13.1 показана визуализация стека вызовов и цикла событий данного скрипта.

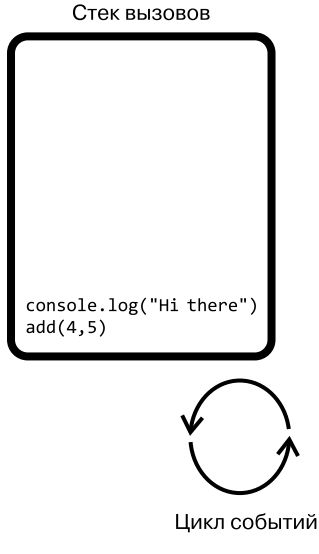


Рис. 13.1. Визуализация стека вызовов и цикла событий

Никакой многопоточности здесь вроде бы и нет. Но она есть:

```
console.log("Hi there");
setTimeout(() => console.log("Sorry I'm late"),
1000);
console.log(add(4, 5));

function add(x, y) {
    return x + y;
}
```

Задача `setTimeout()` передается на аутсорсинг веб-API браузера (подробнее об API-интерфейсах — в главе 15). Когда она будет выполнена, то появится в специальном месте — очереди обратных вызовов, или *колбэков (callback queue)*. Когда стек вызовов опустеет (и только тогда!), цикл событий проверит очередь на предмет наличия задач. Если какие-либо ожидающие обратные вызовы присутствуют, они будут выполняться один за другим, причем после выполнения каждого из них цикл событий сначала проверит, остались ли еще задачи в стеке вызовов.

На рис. 13.2 показана визуализация ситуации с аутсорсингом `setTimeout()`.

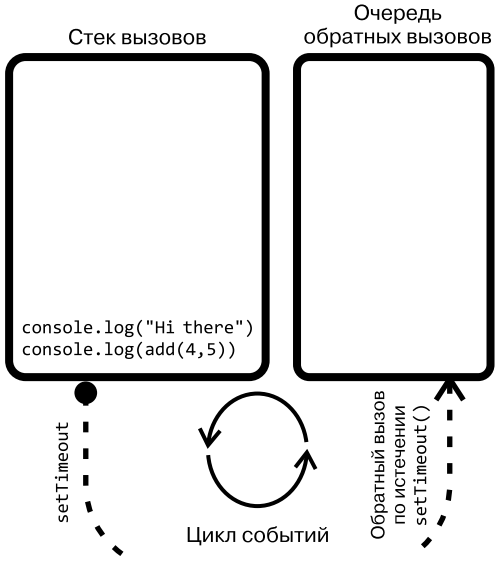


Рис. 13.2. Визуализация setTimeout, переданного на аутсорсинг

Когда `setTimeout()` истечет, цикл событий выполнит все, что уже было в *стеке вызовов*, проверит очередь обратного вызова и реализует в ней любые задачи (рис. 13.3).

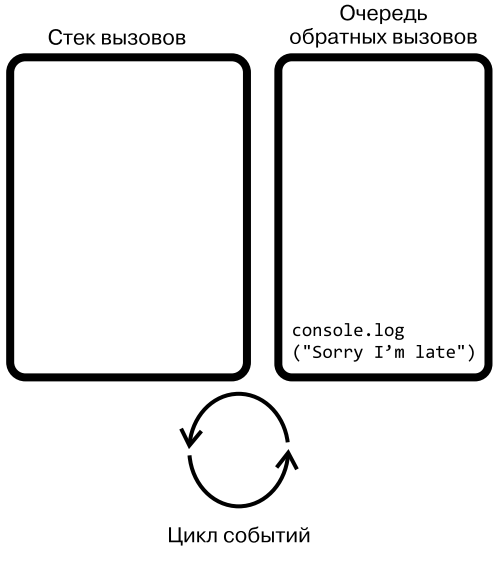


Рис. 13.3. Визуализация задачи в очереди обратных вызовов

И вот что выведет скрипт:

```
Hi there
9
Sorry I'm late
```

Посмотрим, хорошо ли вы изучили приведенный выше материал. Как думаете, что произойдет, когда мы установим таймер, как здесь, на `0`?

```
console.log("Hi there");
setTimeout(() => console.log("Sorry I'm late"), 0);
console.log(add(4,5));

function add(x, y) {
    return x + y;
}
```

Это приведет к аналогичному результату. Когда таймер будет установлен в значение `0`, `setTimeout()` будет передан на аутсорсинг. Обратный вызов сразу же будет помещен в очередь, но цикл событий даже не проверит его до тех пор, пока стек вызовов не опустеет. Таким образом, он по-прежнему будет выводить `Sorry I'm late` после `9`, даже несмотря на то, что таймер находится на `0`.

Проект текущей главы

Проверка паролей

Используя массив допустимых значений пароля, созданная в следующем упражнении программа проверяет наличие одного из этих значений в массиве всех принятых паролей. Установите промис для проверки действительности пароля и, в зависимости от результата, разрешите промис со статусом `true` либо отклоните со статусом `false`. Верните результаты проверки.

1. Создайте массив допустимых паролей.
2. Напишите функцию входа в систему, которая будет проверять, является ли аргумент значением, включенным в массив паролей. Для проверки массива можно применить методы `indexof()` или `includes()`. Верните логическое значение, полученное в результате проверки.

includes() — метод массива, который может проверить, включено ли определенное значение в число элементов массива. В зависимости от результата проверки он вернет соответствующее логическое значение.

3. Добавьте функцию, которая возвращает промис. Верните объект JavaScript с логическим значением `true` или `false`, чтобы указать, допустим ли пароль. Примените `resolve` и `reject`.

4. Создайте функцию, которая проверяет пароль, отправляет его в функцию входа в систему и, используя `then()` и `catch()`, выводит результат либо отклонения пароля, либо его разрешения.
5. Отправьте несколько паролей в функцию проверки: как из массива, так и не из него.

Вопросы для самопроверки

1. Исправьте ошибку в следующем коде:

```
function addOne(val){
  return val + 1;
}
function total(a, b, callback){
  const sum = a + b;
  return callback(sum);
}
console.log(total(4, 5, addOne()));
```

2. Запишите результат работы следующего кода:

```
function checker(val) {
  return new Promise((resolve, reject) => {
    if (val > 5) {
      resolve("Ready");
    } else {
      reject(new Error("Oh no"));
    }
  });
}
checker(5)
  .then((data) => {console.log(data); })
  .catch((err) => {console.error(err); });
```

3. Какие строки кода необходимо добавить к предыдущей функции, чтобы результат ее выполнения всегда гарантировал вывод в консоль слова `done`?
4. Обновите приведенный ниже код, чтобы функция возвращала промис:

```
function myFun() {
  return "Hello";
}
myFun().then(
  function(val) { console.log(val); },
  function(err) { conole.log(err); }
);
```

Резюме

В этой главе мы рассмотрели параллелизм. Это концепция, следуя которой код способен выполнять несколько действий одновременно. Мы можем определять порядок этих действий, используя обратные вызовы, промисы и ключевые слова `async` и `await`. Внедрение их в ваши приложения и страницы значительно улучшит взаимодействие с ними! В наши дни пользователи очень требовательны. Если ваш веб-сайт загружается недостаточно быстро, они, скорее всего, откажутся с ним работать и вернутся в Google в поисках лучшей альтернативы. Параллелизм поможет такому веб-сайту быстрее давать нужный результат.

Следующие две главы посвящены использованию JavaScript для современной веб-разработки и затронут такие темы, как HTML5 в JavaScript, а также фреймворки JavaScript, которые действительно способны изменить правила игры.

14 HTML5, Canvas и JavaScript

HTML5 появился в 2012 году и был стандартизирован в 2014 году. Благодаря этому браузеры начали поддерживать огромное количество новых функций. Внедрение HTML5 повлияло на широту доступных JavaScript-возможностей. Взаимодействие с графикой, видео, а также многие другие опции с момента появления HTML5 стали настолько продвинутыми, что веб-браузеры фактически из-за этого даже перестали поддерживать Flash.

HTML5 позволяет улучшить структуру веб-страниц за счет добавления новых элементов, таких как `<header>`. Модель DOM также была значительно улучшена, что привело к повышению производительности. Есть много других обновлений — вы увидите некоторые из них в данной главе. Одно из них, довольно интересное и полезное, — элемент `<canvas>` — мы также рассмотрим далее.

JavaScript сам по себе дает множество удивительных функций, но при взаимодействии с HTML5 возможностей становится гораздо больше, особенно тогда, когда дело доходит до динамических интерактивных веб-приложений. Эта комбинация позволяет сделать представление контента куда лучшим. Теперь мы можем работать с файлами в браузере, а также рисовать в `canvas` и добавлять туда изображения и текст.

В этой главе мы рассмотрим некоторые удивительные вещи, которые подарил нам HTML5. Не все представленные возможности будут напрямую связаны друг с другом, но все они появились благодаря мощной комбинации HTML5 и JavaScript и, конечно же, очень увлекательны и полезны. Они позволят сделать интерфейс вашего приложения еще более динамичным, интерактивным и привлекательным для пользователя.

Вот темы, которые мы рассмотрим в текущей главе:

- HTML5 и JavaScript;
- чтение локальных файлов;
- геолокация;
- элемент `canvas`;
- динамический элемент `canvas`;

- рисование на `canvas` с помощью мыши;
- сохранение динамических изображений;
- медиаконтент на странице;
- цифровая доступность.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

HTML5 и JavaScript

HTML5 формально является версией HTML. Это огромный шаг вперед по сравнению с его предыдущей версией, благодаря которому мы можем создавать полноценные приложения в веб-браузере, доступные даже в автономном режиме. Как видно из рабочего описания HTML5, это часто нечто большее, чем просто HTML: здесь также подразумевается комбинация HTML5 с JavaScript, CSS, JSON и др.

Благодаря HTML5 структура нашей страницы будет улучшена. Добавятся такие новые элементы, как `<header>`, `<nav>` и `<article>`. С помощью элемента `<video>` мы можем воспроизводить видео — следовательно, нам больше не нужен Flash. И, как мы уже говорили, мы можем работать с элементом `<canvas>` для создания визуальных элементов (таких как анимация, графики и т. д.) или для их представления. Некоторые задачи, для выполнения которых раньше нужен был JavaScript, теперь можно реализовать с помощью HTML (например, добавление видео и аудио на веб-страницу). Изменения в модели DOM также ускорили загрузку веб-страницы. Далее мы собираемся углубиться в изучение некоторых возможностей, специфичных для HTML5. Давайте начнем с доступа к файлам из браузера.

Чтение локальных файлов

С появлением HTML5 мы наконец можем, используя JavaScript, который запускается в нашем браузере, взаимодействовать с локальными файлами. Благодаря этой функции можно загружать файлы с нашего устройства в веб-приложение и там считывать их данные. Следовательно, теперь мы можем прикреплять файлы — например, к формам. Это полезная опция для многих случаев: например, там, где нам необходимо добавить резюме к онлайн-заявлению о приеме на работу.

Сначала убедимся, что используемый вами браузер поддерживает такую возможность. Для этого можно запустить простой скрипт:

```
<!DOCTYPE html>
<html>
  <body>
```

```
    <div id="message"></div>
    <script>
      let message = document.getElementById("message");
      if (window.FileReader) {
        message.innerText = "Good to go!";
      } else {
        message.innerText = "No FileReader :(";
      }
    </script>
  </body>
</html>
```

Когда вы откроете этот файл в своем браузере, на странице должно появиться сообщение Good to go!, если ваш браузер действительно поддерживает чтение файлов. Если же вы увидели сообщение No FileReader :(— попробуйте обновить браузер или использовать другой (рабочие варианты — это, например, Chrome и Firefox).

Загрузка файлов

На самом деле загрузка файлов прописывается в коде проще, чем вы можете себе представить. Мы выражаем свое желание загрузить файл, добавив ввод типа `file`. Перед вами базовый пример, который выполняет только эту операцию:

```
<!DOCTYPE html>
<html>
  <body>
    <input type="file" onchange="uploadFile(this.files)" />
    <div id="message"></div>
    <script>
      let message = document.getElementById("message");

      function uploadFile(files) {
        console.log(files[0]);
        message.innerText = files[0].name;
      }
    </script>
  </body>
</html>
```

Он выдает нам пустую HTML-страницу с кнопкой Choose file и комментарием No file chosen. При нажатии кнопки открывается проводник, где вы можете выбрать файл. После выбора запускается JavaScript. Мы отправляем файлы свойств, которые активны в теле — это список файлов. Берем 0-й индекс — первый элемент в списке. Файлы представляются в виде объектов.

Объект файла отображается на экране, что позволяет нам просматривать все свойства и связанные с ними значения. Некоторые из них (самые важные) — это `name`, `size`, `type` и `lastModified`, но на самом деле их намного больше.

Вводим название файла в `innerText` сообщения `div`. На экране вы увидите, как название файла появится в `div`. Что-то подобное можно сделать для нескольких файлов. Вот как загрузить несколько файлов одновременно:

```
<html>
  <body>
    <input type="file" multiple onchange="uploadFile(this.files)" />
    <div id="message"></div>
    <script>
      let message = document.getElementById("message");

      function uploadFile(files) {
        for (let i = 0; i < files.length; i++) {
          message.innerHTML += files[i].name + "<br>";
        }
      }
    </script>
  </body>
</html>
```

Мы добавили атрибут `multiple` к нашему входному элементу. Текст на кнопке изменился. Теперь вместо Choose file там написано Choose files — и мы можем выбрать больше файлов.

Мы также немного изменили нашу функцию загрузки, добавив цикл. Вместо `innerText` мы теперь используем `innerHTML` — в таком случае можно вставить HTML-разрыв. Скрипт выведет имена всех выбранных файлов под полем ввода на экране.

Чтение файлов

Для чтения файлов существует специальный объект JavaScript. У него очень подходящее название: `FileReader`. Вот как его можно применять:

```
<!DOCTYPE html>
<html>
  <body>
    <input type="file" onchange="uploadAndReadFile(this.files)" />
    <div id="message"></div>
    <script>
      let message = document.getElementById("message");

      function uploadAndReadFile(files) {
        let fr = new FileReader();
        fr.onload = function (e) {
          message.innerHTML = e.target.result;
        };
        fr.readAsText(files[0]);
      }
```

```
    </script>
  </body>
</html>
```

Как видите, для подключения HTML и JavaScript к файлу сначала необходимо указать, что должно произойти. Мы делаем это, добавляя `onload` в качестве анонимной функции, которая отправляет данные о событии.

Затем чтение данных может быть выполнено с помощью одного из методов `readAs()` объекта `FileReader`. В данном случае мы применили метод `readAsText()`, потому что имеем дело с текстовым файлом. Он инициирует фактическое считывание. Когда работа метода закончится, запускается связанная с ним функция `onload`, добавляя результат считывания к нашему сообщению. Такой вариант принимает все типы файлов — но не во всех случаях это даст какой-то осмысленный результат. Чтобы все же его увидеть, нам придется загрузить файл с обычным текстом (например, `.txt`, `.json` и `.xml`). С помощью этой операции мы также можем отправить файл на сервер или обработать содержимое лог-файла.

Практическое занятие 14.1

Это упражнение продемонстрирует процесс загрузки и отображения локальных файлов изображений на вашей веб-странице. В качестве исходного шаблона можете использовать следующий HTML- и CSS-код:

```
<!doctype html>
<html>
<head>
    <title>Complete JavaScript Course</title>
    <style>
        .thumb {
            max-height: 100px;
        }
    </style>
</head>
<body>
    <input type="file" multiple accept="image/*" />
    <div class="output"></div>
    <script>

    </script>
</body>
</html>
```

Чтобы заполнить элемент `script`, выполните следующие шаги.

1. Выберите элементы вашей страницы в качестве значений внутри переменных в коде JavaScript.

2. Добавьте прослушиватель событий к полю `input`. Триггер события следует изменить таким образом, чтобы он немедленно вызывал функцию считывания.
3. Создайте функцию, управляющую считыванием выбранных файлов.
4. Используя объект события, выберите целевой элемент, который вызвал событие. Получите файлы, выбранные в рамках этого ввода, и назначьте их переменной `files`.
5. Переберите все выбранные файлы.
6. Установите для файлов значение переменной с именем `file`, используя индекс внутри цикла.
7. Установите файл изображения, выбранный из файлов поля ввода, в качестве файла в цикле.
8. Добавьте вновь созданный тег `img` на страницу. Создайте область на странице, в которую вы можете выводить содержимое, и отправьте туда этот новый элемент.
9. Создайте новый объект `FileReader`.
10. Добавьте прослушиватель событий `onload` объекту `fileReader` для создания и вызова анонимной функции, которая установит источник изображения в качестве результата, полученного от целевого элемента. Передайте объект изображения, который вы только что создали, в качестве аргумента в функцию.
11. Получите текущий объект файла и передайте его в объект чтения, чтобы файл можно было использовать после добавления на страницу. Примените `readAsDataURL()` для решения этой задачи.
12. Теперь можете выбрать несколько файлов изображений со своего компьютера и поместить их на своей веб-странице.

Использование функции GeoLocation для получения данных местоположения

Теперь познакомимся с навигатором объектов окна (свойством `navigator` объекта `window`) и поймем, сможем ли мы найти местоположение нашего пользователя. Это может быть полезным во многих случаях: например, для поиска ресторанов поблизости. Работу функции `GeoLocation` можно наблюдать, внедрив в код `navigator.geolocation`. Вот один из способов это сделать:

```
<!DOCTYPE html>
<html>
```

```
  <body>
    <script>
      window.onload = init;

      function init() {
        console.dir(navigator.geolocation);
      }
    </script>
  </body>
</html>
```

В логе вы увидите содержимое объекта `GeoLocation`, один из вариантов — текущее местоположение пользователя. Вот как его найти:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      window.onload = init;

      function init() {
        navigator.geolocation.getCurrentPosition(showGeoPosition);
      }

      function showGeoPosition(data) {
        console.dir(data);
      }
    </script>
  </body>
</html>
```

Все это может показаться немного более сложным, чем вы ожидали. А все потому, что метод `getCurrentPosition()` принимает другой метод в качестве аргумента. Данные о местоположении отправляются в эту функцию, принимающую данные в качестве входящих. Следовательно, нам придется обернуть `console.dir()` во внешнюю функцию (`showGeoPosition()`), которая принимает параметр и выводит данные на экран. Затем мы можем отправить эту функцию в `getCurrentPosition()` и просмотреть данные.

При запуске этого процесса вы должны получить объект `GeolocationPosition` со свойством `coords`, содержащим ваши широту и долготу. Браузер может попросить вашего согласия на предоставление местоположения. Если он ничего не показывает, убедитесь, что настройки вашего компьютера позволяют браузеру работать с местоположением.

Используя такой подход, вы можете получить местоположение пользователей и на этой основе показывать им персонализированный контент или собирать данные для других целей, таких как анализ местоположения посетителей.

HTML5-элемент canvas

Мы уже говорили, что элемент `<canvas>` является новым в HTML5, правда? Это удивительный инструмент, который поможет вам создавать динамические веб-приложения. Вот как прописывается `canvas`:

```
<!DOCTYPE html>
<html>
  <body>
    <canvas id="c1"></canvas>
    <script></script>
  </body>
</html>
```

Когда вы откроете страницу, она окажется пустой. Почему? Что ж, по умолчанию элемент `canvas` представляет собой белый прямоугольник, который вы не увидите на белом фоне. Вы можете добавить CSS, чтобы нарисовать границу объекта или задать цвет основного фона, тогда ваша канва будет заметна.

Но мы, вероятно, хотим что-то туда добавить, и для этого нам понадобится JavaScript. Создадим на этом холсте «рисунок»:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="c1"></canvas>
    <script>
      let canvas = document.getElementById("c1");
      let ctx = canvas.getContext("2d");
      canvas.width = 500; // пикселей
      canvas.height = 500; // пикселей
      ctx.fillRect(20, 40, 100, 100);
    </script>
  </body>
</html>
```

Контекст объекта `canvas` считывается и сохраняется в переменной `ctx` (общепринятое сокращение слова context). Это нужно для рисования на канве. Задаем размеры канвы `500` на `500` пикселей — и здесь мы не сможем использовать параметры CSS для ширины и высоты. Нам понадобятся атрибуты HTML `width` и `height`.

С помощью метода `fillRect()` в контексте `canvas` мы можем нарисовать прямоугольник. Он принимает четыре параметра. Первые два — координаты x и y того места на канве, куда фигура должна быть добавлена. Последние два — это ширина и высота прямоугольника (на самом деле в нашем случае — квадрата). На рис. 14.1 показано, как будет выглядеть результат.

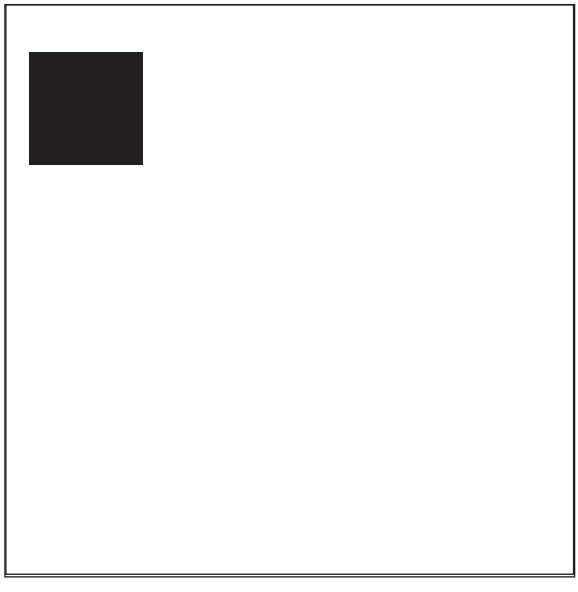


Рис. 14.1. Результат применения метода fillRect() на канве размером 500 на 500 пикселей

Также можно изменить цвет, которым мы рисуем. Вместо черного квадрата вы можете получить розовый, заменив содержание скрипта предыдущего HTML-документа:

```
<script>
  let canvas = document.getElementById("c1");
  let ctx = canvas.getContext("2d");
  canvas.width = 500; // пикселей
  canvas.height = 500; // пикселей
  ctx.fillStyle = "pink";
  ctx.fillRect(20, 40, 100, 100);
</script>
```

Сейчас мы использовали слово pink — но, прописывая fillStyle, вы также можете работать с шестнадцатеричными цветовыми кодами (для розового цвета, например, код будет выглядеть так: #FFC0CB). Первые два символа указывают на количество красного цвета (FF), третий и четвертый — зеленого (C0), последние два — голубого (CB). Диапазон значений — от 00 до FF (от 0 до 255 в десятичной системе).

Существует множество вещей, которые вы можете сделать в элементе `canvas`, помимо простого рисования. Рассмотрим добавление текста.

Практическое занятие 14.2

В данном упражнении мы создадим фигуру и используем элемент `canvas` для рисования на веб-странице с JavaScript. Результат выполнения задания будет выглядеть следующим образом (рис. 14.2).

Рис. 14.2. Результат выполнения упражнения

Выполните следующие действия.

1. Добавьте элемент `canvas` на страницу.
2. Установите ширину и высоту элемента равными 640 пикселям и, используя CSS, добавьте границу толщиной 1 пиксель.
3. В JavaScript выберите элемент `canvas` и установите для `Context` значение `2d`.
4. Задайте прямоугольнику красную заливку.
5. Создайте вывод фигуры с помощью `fillRect`.
6. Установите контур прямоугольника.
7. Очистите внутреннюю заливку прямоугольника, чтобы сделать его прозрачным. Он должен принять цвет фона.

Динамический элемент canvas

В `canvas` мы можем рисовать более сложные фигуры, добавлять изображения и текст. Это позволяет вывести наши навыки работы с элементом на новый уровень.

Добавление линий и кругов элементу canvas

Рассмотрим, как нарисовать линию и круг. Вот фрагмент простого кода, который создает линию:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      #canvas1 {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="canvas1"></canvas>
    <script>
      let canvas = document.getElementById("canvas1");
      let ctx = canvas.getContext("2d");
      canvas.width = 100;
      canvas.height = 100;
      ctx.lineWidth = 2;
      ctx.moveTo(0, 20);
      ctx.lineTo(50, 100);
      ctx.stroke();
    </script>
  </body>
</html>
```

Заданная ширина линии — два пикселя. В этот раз наша канва меньше: `100` на `100` пикселей. Скрипт сначала фокусирует внимание на точке с координатами `0` *(x)* и `20` *(y)*. Это означает, что точка находится на левом краю холста, в `20` пикселях от верха. Вторая точка расположена в координатах `50` *(x)* и `100` *(y)*. Вот как выглядит линия (рис. 14.3).

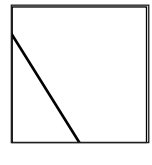


Рис. 14.3. Результат нанесения линии на элемент canvas

А вот как рисуется круг.

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <style>
      #canvas1 {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="canvas1"></canvas>
    <script>
      let canvas = document.getElementById("canvas1");
      let ctx = canvas.getContext("2d");
      canvas.width = 150;
      canvas.height = 200;
      ctx.beginPath();
      ctx.arc(75, 100, 50, 0, Math.PI * 2);
      ctx.stroke();
    </script>
  </body>
</html>
```

Метод `arc()` применяется для создания круга или кривой линии. Он принимает пять параметров, таких как:

- начальная позиция *x* на холсте;
- начальная позиция *y* на холсте;
- радиус круга;
- начальный угол в радианах;
- конечный угол в радианах.

Итак, если вам понадобится не круг, а, например, полукруг, придется указать разные начальный и конечный углы в радианах. На этот раз для рисования мы использовали метод `stroke()` вместо метода `fill()` (рис. 14.4).

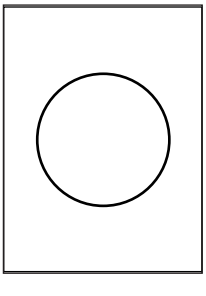


Рис. 14.4. Результат отображения окружности с использованием метода arc()

Метод `stroke()` только рисует линию, в то время как `fill()` закрашивает всю форму.

В элементе `canvas` фигуры и линии добавляются друг на друга в зависимости от очередности их создания. Первая фигура, которую вы рисуете, находится под последующей: то же самое происходит и при рисовании на настоящем холсте. Рассмотрим это в следующем практическом упражнении.

Практическое занятие 14.3

В этом упражнении вы нарисуете человечка-палочку с помощью элемента `canvas`.

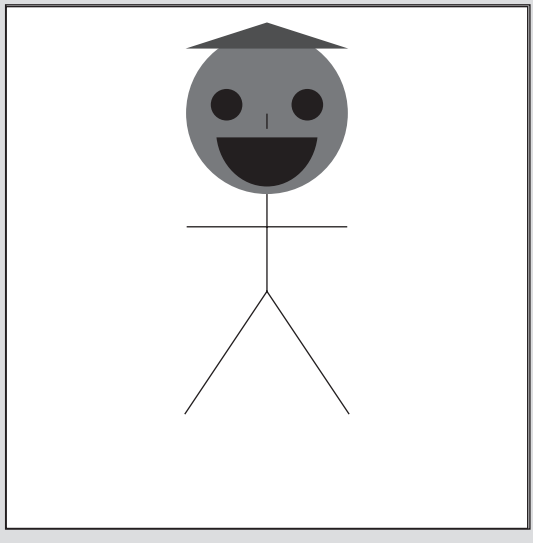


Рис. 14.5. Результат выполнения упражнения в элементе canvas веб-браузера

1. Создайте элементы страницы и подготовьтесь к рисованию на канве.
2. Начните с дуги примерно в центре верхней границы вашего объекта `canvas`.
3. Используя `arc()`, установите положение левого глаза (примерно в верхнем левом углу относительно центра дуги, которую вы только что нарисовали), затем добавьте еще одну дугу для правого глаза. Создайте половину дуги для рта (радиальный угол для полукруга равен пи) и заполните все цветом.
4. Переместите позицию для рисования в центр и создайте линию для носа.
5. Нарисуйте тело линией вниз от центра дуги, создайте левую руку, а затем переместите позицию для рисования, чтобы нарисовать правую руку, которая будет в два раза шире левой. Вернитесь к центру и продолжайте движение вниз, чтобы нарисовать левую ногу, после чего снова вернитесь к центру и нарисуйте линию для правой ноги.
6. Переместитесь наверх, установите синий цвет и нарисуйте треугольник-шляпу.

Добавление текста в элемент canvas

Аналогичным образом мы можем добавить на канву текст. В этом примере мы устанавливаем шрифт и его размер, а затем записываем наш текст:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      #canvas1 {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="canvas1"></canvas>
    <script>
      let canvas = document.getElementById("canvas1");
      let ctx = canvas.getContext("2d");
      canvas.width = 200;
      canvas.height = 200;
      ctx.font = "24px Arial";
      let txt = "Hi canvas!";
      ctx.fillText(txt, 10, 35);
    </script>
  </body>
</html>
```

Для добавления текста используется метод `fillText()`. Мы должны указать три параметра: текст, позицию *x* и позицию *y*. На рис. 14.6 представлен результат.

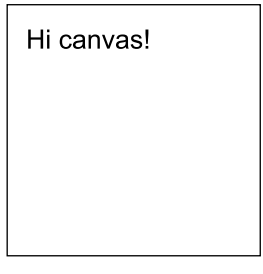


Рис. 14.6. Результат работы метода fillText()

Мы написали, что текст должен отступать на 35 пикселей сверху. Можно указать и другие свойства текста, например:

```
ctx.textAlign = "center";
```

Здесь мы применяем свойство `textAlign`, чтобы указать, каким должно быть выравнивание текста.

Практическое занятие 14.4

В следующем упражнении будет продемонстрировано, как динамически добавить текст и разместить его внутри элемента `canvas`. Результат выполнения кода будет выглядеть таким образом (рис. 14.7).

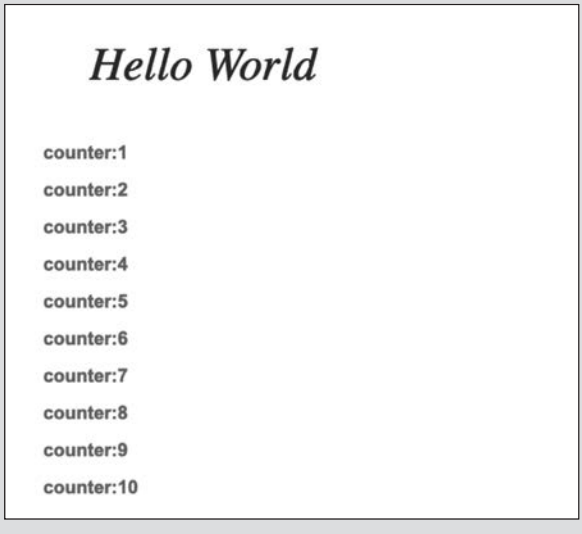


Рис. 14.7. Результат выполнения упражнения

Выполните следующие действия.

1. Создайте простой HTML-документ и добавьте туда элемент `canvas`. Установите высоту и ширину канвы равными 640 пикселей, добавьте границу толщиной 1 пиксель, чтобы вы могли видеть элемент на странице.
2. Выберите элементы страницы в качестве переменных JavaScript.
3. Создайте строковую переменную с сообщением `Hello World`.
4. Задайте стиль шрифта с помощью свойства `font` и синий цвет надписи с помощью свойства `fillStyle`. Можете выровнять текст по левому краю.
5. С помощью `fillText` добавьте текст на канву и задайте позиции его x и y.
6. Определите новый шрифт и задайте ему красный цвет.
7. Создайте цикл и, используя значение переменной цикла, добавьте текст на канву страницы.

Добавление и загрузка изображений в элемент canvas

В элемент `canvas` мы также можем добавить изображение. Получим изображение со страницы и добавим его на канву:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="c1"></canvas>
    <img id="flower" src="flower.jpg" />
    <script>
      window.onload = function () {
        let canvas = document.getElementById("c1");
        canvas.height = 300;
        canvas.width = 300;
        let ctx = canvas.getContext("2d");
        let myImage = document.getElementById("flower");
        ctx.drawImage(myImage, 10, 10);
      };
    </script>
  </body>
</html>
```

Здесь мы добавляем прослушиватель событий `onload`: мы хотим быть уверены, что изображение загружено, прежде чем получать его из DOM, иначе холст останется пустым. Для добавления изображения на канву используем метод `drawImage()`. Он принимает три аргумента: *изображение*, положение *x*, положение *y*.

Таким же образом мы можем создать один элемент `canvas` внутри другого. Это очень мощное свойство, потому что оно позволяет нам использовать, например, не весь предложенный пользователем рисунок, а только его часть. Вот как это делается:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
```

```
    <canvas id="canvas1"></canvas>
    <canvas id="canvas2"></canvas>
    <canvas id="canvas3"></canvas>
    <script>
      let canvas1 = document.getElementById("canvas1");
      let ctx1 = canvas1.getContext("2d");
      ctx1.strokeRect(5, 5, 150, 100);

      let canvas2 = document.getElementById("canvas2");
      let ctx2 = canvas2.getContext("2d");
      ctx2.beginPath();
      ctx2.arc(60, 60, 20, 0, 2 * Math.PI);
      ctx2.stroke();

      let canvas3 = document.getElementById("canvas3");
      let ctx3 = canvas3.getContext("2d");
      ctx3.drawImage(canvas1, 10, 10);
      ctx3.drawImage(canvas2, 10, 10);
    </script>
  </body>
</html>
```

Мы создаем три холста, к двум добавляем фигуры по отдельности, а на третьем будет отображаться комбинация первых двух фигур. Выглядит все это следующим образом (рис. 14.8).

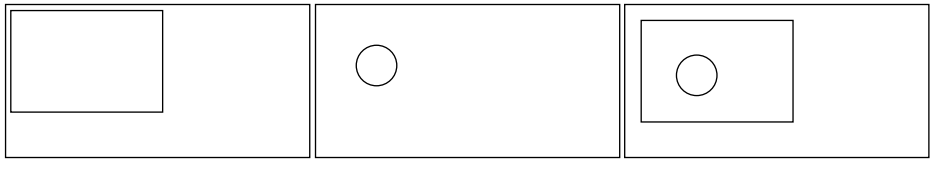


Рис. 14.8. Результат: три объекта canvas с фигурами

На канву также можно загружать изображения. Это может быть полезно в тех случаях, когда вы, например, хотите показать вашему пользователю превью чего-то, что только что было загружено (скажем, фото профиля). Подобное происходит при захвате элемента `<img>` с веб-страницы и его дальнейшем использовании. Но на сей раз нам нужно считать данные из загруженного файла, создать новый элемент изображения, а затем отобразить изображение на канве.

Следующий код делает именно это:

```
<html>
  <head>
    <style>
      canvas {
```

```
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <input type="file" id="imgLoader" />
    <br>
    <canvas id="canvas"></canvas>
    <script>
      let canvas = document.getElementById("canvas");
      let ctx = canvas.getContext("2d");
      let imgLoader = document.getElementById("imgLoader");
      imgLoader.addEventListener("change", upImage, false);
      function upImage() {
        let fr = new FileReader();
        fr.readAsDataURL(event.target.files[0]);
        fr.onload = function (e) {
          let img = new Image();
          img.src = event.target.result;
          img.onload = function () {
            canvas.width = img.width;
            canvas.height = img.height;
            ctx.drawImage(img, 0, 0);
          };
          console.log(fr);
        };
      }
    </script>
  </body>
</html>
```

Каждый раз, когда данные в поле ввода изменяются, выполняется метод `upImage()`. Он решает несколько задач. Прежде всего, мы создаем новый `FileReader` и добавляем загруженный файл (в нашем случае только один, поэтому используем индекс `0`). Вместо уже знакомого нам `readAsText()` берем `readAsDataURL()`, который также подходит для чтения изображений.

Как результат, будет вызвано событие `onload`. В нашем случае это создает новое изображение, которое позже можно добавить на холст. В качестве источника берем результат чтения и, когда изображение загрузится, меняем размер холста на размер картинки, а затем добавляем туда саму картинку.

Полученные новые навыки позволят вам работать с существующими изображениями на канве, рисовать свои собственные, загружать картинки из других источников и даже повторно использовать их на веб-странице. Это пригодится во многих ситуациях: например, для создания базовой анимации или разработки функции загрузки нового фото в профиль пользователя.

Практическое занятие 14.5

В следующем упражнении мы попробуем загрузить изображение с вашего компьютера и поместить его в `canvas` вашего браузера.

1. Установите элементы страницы и добавьте поле ввода для загрузки изображения. Добавьте `canvas` на страницу.
2. Выберите поле ввода и элементы `canvas` в качестве объектов JavaScript.
3. Добавьте прослушиватель событий для вызова функции загрузки при изменении содержимого поля ввода.
4. Напишите вышеупомянутую функцию. С помощью `FileReader` создайте новый объект `FileReader`. В событии `reader.onload` укажите новый объект изображения.
5. Добавьте прослушиватель событий `onload` к объекту изображения, чтобы при загрузке изображения установить высоту и ширину канвы равными половине высоты и ширины изображения. Добавьте изображение в `canvas` страницы с помощью `ctx.drawImage()`.
6. Установите результат из входного значения в качестве источника `img`.
7. Используйте объект `reader` и вызовите `readAsDataURL()` для преобразования входного значения файла в читаемый формат данных изображения (base64), который можно использовать в `canvas`.

Добавление анимации в элемент canvas

Методов, которые мы уже изучили, достаточно для того, чтобы начать работу над анимацией. Мы реализуем ее с помощью циклов и рекурсии в сочетании с `timeout()`. Серия рисунков, сменяющих друг друга через (короткие) временные интервалы, и создает анимацию. Начнем с самой простой:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>

  <body>
    <canvas id="canvas"></canvas>
    <script>
```

```
      window.onload = init;
      var canvas = document.getElementById("canvas");
      var ctx = canvas.getContext("2d");
      canvas.height = 500;
      canvas.width = 500;
      var pos = {
        x: 0,
        y: 50,
      };

      function init() {
        draw();
      }

      function draw() {
        pos.x = pos.x + 5;
        if (pos.x > canvas.width) {
          pos.x = 0;
        }
        if (pos.y > canvas.height) {
          pos.y = 0;
        }

        ctx.fillRect(pos.x, pos.y, 100, 100);
        window.setTimeout(draw, 50);
      }
    </script>
  </body>
</html>
```

Этот скрипт начнет рисовать квадрат в позиции `5`, `50`. Через 50 миллисекунд начнется рисование другого квадрата в позиции `10`, `50`, еще через 50 — в позиции `15`, `50`. Скрипт продолжит изменять значение `x` на `5` до тех пор, пока оно не станет больше ширины элемента `canvas`. Тогда `x` вернется на ноль. Таким образом, последний участок белого холста на этой линии тоже будет окрашен в черный цвет.

В данном виде все это больше похоже, скорее, на линию, чем на движущийся квадрат. Дело в том, что мы продолжаем добавлять цветную фигуру на холст, но не сбрасываем значение цвета фигуры из предыдущей итерации. Исправить это можно с помощью метода `clearRect()`. Он принимает четыре параметра. Первые два являются отправной точкой для рисования очищаемого прямоугольника (x и y). Третий — это `width` (ширина) прямоугольника, а последний — `height` (высота). Чтобы очистить весь элемент `canvas`, следует написать:

```
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

Добавление этого метода в начало функции из предыдущего примера приводит к тому, что вместо жирной линии на экране появляется движущийся квадрат: при выполнении функции канва каждый раз очищается и новый квадрат рисуется с нуля.

Практическое занятие 14.6

Попрактикуемся в оживлении фигур на странице. В этом упражнении мы рассмотрим, как перемещать объекты с помощью элемента `canvas` и JavaScript (рис. 14.9).

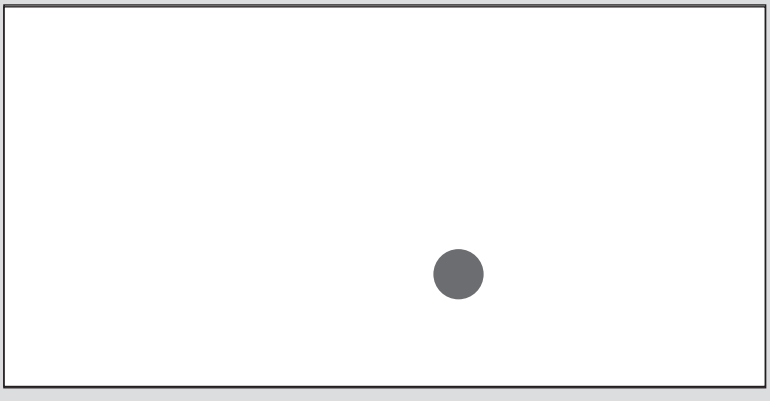


Рис. 14.9. Красный круг, движущийся в пределах границ элемента canvas

Чтобы создать красный круг, который затем будет перемещаться в пределах границ холста, как бы отскакивая от них, выполните следующие действия.

1. Создайте холст и задайте ему обводку толщиной 1 пиксель.
2. Выберите элементы `canvas` с помощью JavaScript и подготовьтесь к рисованию на холсте.
3. Задайте переменные для отслеживания положений x и y, а также скорости перемещения в направлениях x и y. В качестве значений по умолчанию для скорости перемещения можете установить значение `1`. Начальные позиции x и y могут составлять половину размеров холста.
4. Создайте функцию, которая нарисует шар, начав с красной дуги в положениях x и y. Размер шара должен быть задан как переменная, чтобы по нему можно было вычислять границы. Замкните контур и заполните фигуру.
5. Создайте функцию, перемещающую мяч, и установите для нее интервал, равный 10 миллисекундам.
6. В вышеупомянутой функции очистите текущий прямоугольник и нарисуйте шар, используя соответствующую функцию.
7. Проверьте положение мяча на странице. Если мяч находится за пределами границ холста, нужно изменить направление его движения (умножив значение направления на -1). Обновите позиции x и y.

Рисование на холсте с помощью мыши

У нас уже есть все ингредиенты для канвы, на которой можно рисовать с помощью мыши. Так давайте создадим ее. Начнем с настроек холста:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <canvas id="canvas"></canvas>
    <input type="color" id="bgColor" />
    <script>
      let canvas = document.getElementById("canvas");
      let ctx = canvas.getContext("2d");
      canvas.width = 700;
      canvas.height = 700;
    </script>
  </body>
</html>
```

В элементе `script` мы собираемся добавить метод для определения момента загрузки окна. Когда окно загрузится, нам нужно добавить несколько прослушивателей событий:

```
window.onload = init; // добавьте эту строку в начало скрипта

function init() {
  canvas.addEventListener("mousemove", draw);
  canvas.addEventListener("mousemove", setPosition);
  canvas.addEventListener("mouseenter", setPosition);
}
```

Мы хотим рисовать движением мыши. Когда мышь движется, необходимо изменять ее текущее положение на холсте. То, что мы хотим сделать, мы зададим также в событии `mouseenter`. Напишем код для установки позиции. Данный фрагмент также нужно указать в элементе `script`. Необходимо добавить переменную `position`, которая должна быть объявлена в начале скрипта:

```
let pos = {
  x: 0,
  y: 0,
};
```

Сама функция установки положения будет выглядеть следующим образом:

```
function setPosition(e) {
  pos.x = e.pageX;
  pos.y = e.pageY;
}
```

Эта функция активизируется событиями `mousemove` и `mouseenter`. Событие, запускающее процесс, имеет свойства `pageX` и `pageY`, которые мы можем использовать для получения текущего положения указателя мыши.

Последней обязательной составляющей для рисования на канве является метод `draw()`. Вот как могла бы выглядеть его реализация:

```
function draw(e) {
    if (e.buttons !== 1) return;
    ctx.beginPath();
    ctx.moveTo(pos.x, pos.y);
    setPosition(e);
    ctx.lineTo(pos.x, pos.y);
    ctx.lineWidth = 10;
    ctx.lineCap = "round";
    ctx.stroke();
}
```

Мы начинаем с чего-то странного, но вообще эта отличная уловка позволяет убедиться, что кнопка мыши действительно нажата. Программа не должна рисовать, если нажатия не произошло, — в таком случае метод предотвращает подобное развитие событий, возвращаясь из метода.

Затем мы начинаем прокладывать путь. У нас всегда есть текущие x и y — они задаются как координаты. Далее мы задаем их снова и используем новые координаты для линии. Мы задаем параметр `linecap`, чтобы добиться плавности линий, и ширину линии со значением `10`. Затем рисуем линию, и, пока мышь движется, функция `draw()` вызывается снова.

Теперь приложение можно открыть и использовать в качестве работающего инструмента рисования. Мы также можем предоставить пользователю дополнительные возможности — например, средство выбора цвета. Добавим его в HTML:

```
<input type="color" id="bgColor" />
```

Обновите цвет в JavaScript, добавив прослушиватель событий, реагирующий на изменение значения в этом поле ввода:

```
let bgColor = "pink";
let bgC = document.getElementById("bgColor");
bgC.addEventListener("change", function () {
    bgColor = event.target.value;
});
```

Мы начинаем с розового и перезаписываем его тем цветом, который пользователь выбирает в палитре цветов.

Практическое занятие 14.7

В следующем упражнении мы создадим интерактивную доску для рисования и присвоим динамические значения для ширины и цвета, а также возможности стирания текущего рисунка. Используйте следующий HTML-код в качестве шаблона, добавив туда код JavaScript:

```
<!doctype html>
<html>
<head>
  <style>
    canvas {
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <div class="controls">
    <button class="clear">Clear</button> <span>Color
      <input type="color" value="#ffff00" id="penColor"></span><span>Width
      <input type="range" min="1" max="20" value="10" id="penWidth"></span>
  </div>
  <canvas id="canvas"></canvas>
  <script>
  </script>
</body>
</html>
```

Выполните следующие шаги.

1. Выберите элементы страницы как переменные JavaScript. Задайте поле ввода и возьмите кнопку в качестве объекта.
2. Добавьте к кнопке прослушиватель событий, который будет запускать функцию очистки канвы. В рамках функции используйте метод `confirm()`, чтобы проверить, действительно ли пользователь хочет стереть рисунок на холсте. Если он подтвердит это с помощью `clearRect()`, удалите содержимое элемента `canvas`.
3. Создайте глобальный объект `position` для *x* и *y* и, добавив прослушиватель событий к событиям мыши, обновите его. Если мышь начинает перемещаться, вызывайте функцию рисования. Задайте значение позиции, чтобы обновить положение мыши, установив глобальные значения для *x* и *y*.
4. В функции рисования проверьте, нажата ли кнопка мыши; если нет, добавьте `return`. Если кнопка нажата, мы можем рисовать на холсте. Установите новый путь и переместитесь в позиции *x* и *y*. Начните новую линию: получите значение `strokestyle` из поля ввода цвета и установите `linewidth` из поля ввода ширины. Добавьте метод `stroke()`, чтобы отобразить новую линию на странице.

Сохранение динамических изображений

Мы можем преобразовать канву в изображение и сохранить его. Для этого в наш элемент скрипта нужно добавить следующее:

```
let dataURL = canvas.toDataURL();
document.getElementById("imageId").src = dataURL;
```

Мы меняем холст на URL данных, который станет источником изображения. Так должно происходить всякий раз, когда нажимается кнопка сохранения. Вот код кнопки:

```
<input type="button" id="save" value="save" />
```

И прослушиватель событий:

```
document.getElementById("save").addEventListener("click", function () {
  let dataURL = canvas.toDataURL();
  document.getElementById("holder").src = dataURL;
});
```

Теперь всякий раз, когда нажимается кнопка сохранения, изображение в URL данных, сгенерированных с холста, будет обновляться. Независимо от содержимого `canvas` оно будет преобразовано в изображение с типом данных base64 и добавлено на страницу в теге `img`.

В следующем примере есть канва размером 200 на 200 пикселей и пустое изображение того же размера. При выборе цвета на холсте рисуется квадрат размером 100 на 100 пикселей этого цвета. При нажатии кнопки Сохранить холст преобразуется в изображение, которое позже можно сохранить. Вот код для примера:

```
<!doctype html>
<html>
<head>
    <style>
        canvas {
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <canvas id="canvas"></canvas>
    <input type="color" id="squareColor" />
    <br>
    <img src="" width="200" height="200" id="holder" />
```

```
    <input type="button" id="save" value="save" />
    <script>
        const canvas = document.getElementById("canvas");
        const ctx = canvas.getContext("2d");
        canvas.width = 200;
        canvas.height = 200;
        const penColor = document.getElementById("squareColor");
        penColor.addEventListener("change", function () {
            color = event.target.value;
            draw(color);
        });
        document.getElementById("save").addEventListener("click", function () {
            let dataURL = canvas.toDataURL();
            document.getElementById("holder").src = dataURL;
        });
        function draw(color) {
            ctx.fillStyle = color;
            ctx.fillRect(70, 70, 100, 100);
        }
    </script>
</body>
</html>
```

Вот как страница выглядит после сохранения изображения (рис. 14.10).

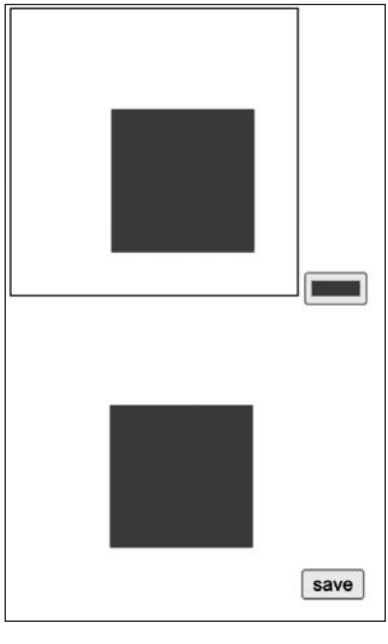


Рис. 14.10. Результат сохранения изображения

Мультимедийный контент на странице

Для размещения медиаконтента на странице существуют особые элементы JavaScript. Мы покажем, как добавить аудио и видео и встроить ролик из YouTube.

Добавить аудиопроигрыватель на страницу очень просто:

```
<!DOCTYPE html>
<html>
  <body>
    <audio controls>
      <source src="sound.ogg" type="audio/ogg">
      <source src="sound.mp3" type="audio/mpeg">
    </audio>
  </body>
</html>
```

Вы указываете атрибут `controls`, если хотите, чтобы пользователь мог управлять воспроизведением (ставить трек на паузу и снова его запускать), а также регулировать громкость. Если хотите, чтобы проигрыватель запускался автоматически, добавьте атрибут `autoplay`. С помощью элемента `source` вы указываете файлы, которые нужно воспроизвести. В браузере появится только один из них — тот, формат которого поддерживается и который в списке будет идти первым (по направлению сверху вниз).

Добавление видео на веб-страницу происходит по подобному принципу. Вот как это сделать:

```
<video width="1024" height="576" controls>
    <source src="movie.mp4" type="video/mp4">
    <source src="movie.ogg" type="video/ogg">
</video>
```

Хотя часто вместо этого вы будете ссылаться на видео в YouTube:

```
<iframe
    width="1024"
    height="576"
    src="https://www.youtube.com/embed/v6VTv7czb1Y"
>
</iframe>
```

Здесь вам необходим `iframe` — это специальный элемент, который позволяет использовать другую веб-страницу внутри текущей веб-страницы. Туда вы сможете добавить в качестве источника видео ссылку на YouTube. Остальной код после `embed` берется из URL-адреса видео.

Чтобы сделать размер видео бóльшим или меньшим, атрибуты высоты и ширины окна можно изменить. Если хотите показать его в полноэкранном режиме, можете задать ширину и высоту следующим образом:

```
<iframe
  width="100%"
  height="100%"
  src="https://www.youtube.com/embed/v6VTv7czb1Y"
>
</iframe>
```

Чтобы видео занимало только часть экрана, атрибуты ширины и высоты можно настроить соответствующим образом.

Автоматическое воспроизведение видео задается с помощью атрибута `autoplay`. Обратите внимание, что, если вы зададите автозапуск более чем для одного элемента, ни один из них сам не воспроизведется. Это предусмотрено для защиты пользователя от шума с веб-страницы. Если в браузере видео начнет шуметь само по себе, посетитель, скорее всего, не будет ему рад. Добавление атрибута `muted` предотвратит это.

Цифровая доступность в HTML

Цифровая доступность имеет огромное значение для людей с ослабленным зрением или тех, кто не может взять в руки мышь. Для комфортного использования интернета при слабом зрении или при его отсутствии существуют различные программы чтения с экрана, распознающие то, что там находится, или преобразующие и распечатывающие текст шрифтом Брайля с помощью подключенных к компьютеру специальных устройств. Люди, которые не могут пользоваться мышью, часто полагаются на голосовых помощников, чтобы давать задания компьютеру.

Ранние веб-приложения были ужасны с точки зрения доступности. К счастью, стандарт WAI-ARIA описал то, как технически расширить доступ к цифровой информации в интернете. Динамические части HTML могут быть распознаны внешними инструментами, если они правильно реализованы. Добавление семантики и метаданных также упрощает использование страницы.

Семантика играет здесь очень важную роль. Ключевой момент: нужно использовать правильный HTML-элемент для правильной цели. Например, элемент, который требуется нажать, лучше прописать с помощью `<button>`, а не `<span>`. Если это кнопка, то можно перейти к ней с помощью клавиши Tab и нажать ее, используя Enter.

То же самое относится и к заголовкам. Вы можете сформировать определенный макет и задать что-то похожее на заголовок, используя специальный класс, но программы чтения ищут именно `h1`, `h2` и `h3`. Поэтому для создания заголовков всегда стоит использовать теги заголовков. Это помогает программам чтения и повышает доступность сайта. А еще помогает вам занять более высокое место в результатах поиска Google, поскольку боты, определяя самое важное на вашем сайте, проверяют заголовки именно по тегам.

Также важно использовать ярлыки и давать развернутые описания ссылкам. Ссылка с текстом Нажмите здесь, по сути, бесполезна. А вот фраза в духе «Нажмите здесь, чтобы зарегистрироваться на летнее мероприятие» передает смысл намного лучше.

Везде в книге мы неправильно поступали с нашими полями ввода. Чтобы сделать их доступными, нужно добавить элемент `label`. Это облегчит программам чтения понимание того, о чем идет речь в поле ввода. Использование следующего синтаксиса, как правило, является плохой практикой:

```
<input type="text" id="address" />
```

А вот следующая запись намного лучше, потому что программы чтения ее распознают (следовательно, люди с нарушениями зрения смогут воспринять помещенный там контент):

```
<label for="address">Address:</label>
<input type="text" id="address" />
```

И еще один атрибут, который вы, возможно, уже знаете, — это атрибут `alt` для изображений. Если программа чтения обнаружит изображение, она прочитает описание к нему — `alt`. Поэтому убедитесь, что вы добавили туда какой-то текст, даже если изображение не имеет значения. Пользователь не будет знать, что изображение неважно, ведь попросту его не увидит. Зато он точно будет знать, что какое-то изображение недоступно для просмотра. Вот как добавить текст в `alt`:

```
<img src="umbrella.jpg" width="200" height="200" alt="rainbow colored
umbrella" />
```

Данные советы не так важны для практических целей и тестирования. Тем не менее при создании качественных и профессиональных приложений полезно учитывать перечисленные возможности — они сделают ваше приложение доступным для всех. И, как мы уже говорили, Google (по крайней мере сейчас) вознаградит вас за хорошее поведение, повысив ваш рейтинг, — следовательно, ваше приложение станет более прибыльным, ведь им сможет пользоваться больше людей!

Проекты текущей главы

Создание эффекта матрицы

В этом упражнении мы создадим анимацию текста, непрерывно движущегося сверху вниз. В конце мы увидим, как символы перемещаются вниз по экрану внутри элемента `canvas` и по мере приближения к нижней части экрана исчезают, а вместо них на холст добавляются новые символы. Случайный символ может быть либо `0`, либо `1`, и он расположится в соответствии с номером, который будет представлять вертикальную позицию уже нарисованного символа.

Холст заполнится черным фоном. Для создания эффекта затухания будут использованы настройки прозрачности (рис. 14.11).

Рис. 14.11. Желаемый результат матричного эффекта

Выполните следующие шаги.

1. Создайте простой HTML-документ. В JavaScript-коде создайте элемент `canvas` и добавьте `getContent` как `2d`.
2. Выберите элемент `canvas` и установите его высоту и ширину равными `500` и `400` пикселей соответственно. Добавьте его в `body` вашего кода.
3. Создайте пустой массив `colVal` и задайте цикл для добавления в массив ряда элементов, которые будут иметь значение `0`. Количество элементов, которые вам нужно добавить в массив, можно определить, разделив ширину на 10 — это расстояние между столбцами. Значения в массиве будут являться начальной вертикальной позицией содержимого для метода `fillText()`, который вы зададите.
4. Создайте основную матричную функцию, которая будет запускаться с интервалом 50 миллисекунд.
5. Установите для параметра `fillStyle` значение черного цвета с затененностью `0.05`, чтобы при наложении поверх существующих элементов создавался эффект затухания.
6. Установите в канве зеленый цвет шрифта.

7. Используя карту массива, выполните итерацию всех текущих элементов в `colVal`, хранящих значения вертикального положения выходного текста.
8. Внутри карты установите символы для отображения. Мы хотим, чтобы они чередовались между `0` и `1`, поэтому, используя `Math.random()`, сгенерируйте значение `0` или `1` для выводимого текста. Можете взять тернарный оператор для реализации этого действия.
9. Установите положение *x*, используя значение индекса, умноженное на 10, — это начало каждой новой буквы. Для создания отдельных столбцов движущихся символов возьмите индекс из массива `colVal`.
10. Создайте символ внутри канвы. С помощью контекстного метода `fillText()` задайте выходной символ — случайное значение, `0` или `1`. Используйте `posX`, чтобы задать позицию *x*, и `posY` — значение в массиве `colVal` для позиции *y*.
11. Добавьте условие, которое проверяет, является ли значение позиции *y* больше 100 плюс случайное значение от 0 до 300. Чем больше значение числа, тем дольше оно будет находиться в позиции *y*. Значение случайно, поэтому не все числа будут исчезать в одном и том же месте. Это создаст ступенчатый эффект после первоначального падения чисел на экране.
12. Если значение позиции *y* не превышает случайное плюс 100, увеличьте значение индекса на 10. Присвойте значение *y* элементу в массиве `colVal`, который можно использовать на следующей итерации, — это переместит букву на холсте на 10 пикселей вниз в следующем цикле рисования.

Таймер обратного отсчета

В следующем упражнении мы создадим таймер обратного отсчета, который будет отображать время в днях, часах, минутах и секундах, оставшееся до указанного значения в поле ввода даты. Изменение значения в поле приведет к обновлению таймера. Он также будет использовать локальное хранилище для захвата и хранения значения из поля ввода, поэтому при обновлении страницы значение в поле ввода останется прежним, а таймер продолжит отсчет до этого времени. Можете использовать данный HTML-шаблон:

```
<!doctype html>
<html>
<head>
    <title>JavaScript</title>
    <style>
        .clock {
            background-color: blue;
            width: 400px;
            text-align: center;
            color: white;
            font-size: 1em;
        }
```

```
        .clock>span {
            padding: 10px;
            border-radius: 10px;
            background-color: black;
        }
        .clock>span>span {
            padding: 5px;
            border-radius: 10px;
            background-color: red;
        }
        input {
            padding: 15px;
            margin: 20px;
            font-size: 1.5em;
        }
    </style>
</head>
<body>
    <div>
        <input type="date" name="endDate">
        <div class="clock"> <span><span class="days">0</span> Days</span>
          <span><span class="hours">0</span>
                Hours</span> <span><span class="minutes">0</span>Minutes</span>
          <span><span class="seconds">0</span>Seconds</span>
        </div>
    </div>
    <script>

    </script>
</body>
</html>
```

Мы создали элементы страницы, включая ввод с типом данных `date` и главный контейнер `clock`, а также добавили интервалы для `days`, `hours`, `minutes` и `seconds` (дни, часы, минуты и секунды). Они получили маркировку, и к ним были применены указанные стили CSS.

Выполните следующие действия.

1. Выберите элементы страницы в качестве объектов JavaScript. Задайте область вывода основного значения таймера как значение объекта JavaScript.
2. Создайте переменные для параметра `timeInterval` и глобального логического значения, которое можно использовать для остановки таймера `clock`.
3. Проверьте локальное хранилище: не задан ли там элемент для обратного отсчета. Если записанное значение существует, используйте его.
4. Создайте условие и функцию для запуска таймера с помощью сохраненного значения и напишите значение даты из поля ввода как значение, сохраненное в локальном хранилище.

5. Добавьте прослушиватель событий для вызова функции в случае, если значение поля ввода изменяется. Очистите интервал, если он стал другим, и установите новое значение `endDate` в локальном хранилище.
6. Запустите таймер с помощью соответствующей функции с этого нового входного значения `endDate`.
7. Создайте функцию, используемую для запуска отсчета. В ее рамках вы можете создать функцию, которая обновляет счетчик и выводит новые значения времени в область контейнера часов.
8. Внутри этой функции проверьте, не меньше ли значение `timeLeft` значения времени счетчика. Создайте отдельную функцию для обработки данного процесса. Если значение меньше — остановите таймер.
9. Если оставшееся время больше и значение находится в пределах объекта, выведите объект по именам свойств. Сопоставьте имена свойств, которые вы используете в `timeLeft`, с именами классов в элементах веб-страницы. В случае совпадения вы можете сэкономить время на их переписывании. Перебирайте все значения объекта и присваивайте значения элементу страницы `innerHTML`.
10. В функции оставшегося времени получите текущую дату. Используя `Date.parse()`, проанализируйте ее и вычислите общее количество миллисекунд до остановки счетчика. Верните значения количества дней, часов, минут и секунд в качестве ответа для использования в функции обновления.
11. Если счетчик получил значение `false` и время окончания наступило — очистите интервал. Если таймер все еще работает, установите интервал для запуска функции обновления через каждые 1000 миллисекунд.

Онлайн-приложение для рисования

Создадим приложение, где пользователь сможет рисовать мышью в элементе `canvas`. Когда курсор находится внутри `canvas`, а пользователь нажимает кнопку мыши, удержание этой кнопки в нажатом положении позволит проводить линии, создавая эффект рисования. Цвет и ширину пера для большей функциональности можно динамически изменять. Кроме того, в приложении будет кнопка для сохранения и загрузки изображения из `canvas`, а также для очистки текущего содержимого.

Можете использовать следующий шаблон, добавив в него код JavaScript.

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
    <style>
        #canvas {
```

```
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <canvas id="canvas" width="600" height="400"></canvas>
    <div>
        <button class="save">Save</button>
        <button class="clear">clear</button>
        <span>Color: <input type="color" value="#ffff00" id="penColor"></span>
        <span>Width: <input type="range" min="1" max="20" value="10"
         id="penWidth"></span>
    </div>
    <div class="output"></div>
    <script>

    </script>
</body>
</html>
```

Мы создали кнопку сохранения и кнопку очистки, ввод цвета с HTML5-типом `color` и ввод с типом `range` для получения числового значения ширины пера. Были также добавлены элементы холста и области вывода.

Выполните следующие действия.

1. Выберите все элементы страницы в качестве объектов JavaScript и настройте элемент `canvas`, который будет использован для рисования.
2. Задайте переменную для фиксации расположения пера.
3. Добавьте прослушиватель событий для отслеживания движения мыши на холсте. Обновите позицию пера до значений `lastX` и `lastY`, а затем установите положение на `clientX` и `clientY`. Создайте функцию для рисования в положении пера и вызовите ее.
4. Для `mousedown` задайте `draw` в значении `true`, а для `mouseup` и `mouseout` — `draw` в значении `false`.
5. В функции рисования начните путь со значений местоположения пера и задайте стилю обводки цвет пера, а ширине обводки — ширину пера. Их можно изменить, щелкнув на входных значениях и обновив их. Добавьте обводку и закройте контур.
6. Добавьте прослушиватель событий для кнопки очистки. Создайте функцию нажатия кнопки, которая подтвердит, что пользователь хочет удалить и очистить рисунок, а затем при полученном значении `true` вызовите `clearRect()`, чтобы очистить содержимое холста.
7. Добавьте еще один прослушиватель событий для сохранения изображения. При нажатии на кнопку должна быть вызвана функция, которая получает объект

canvas, используя toDataURL в качестве данных изображения формата base64. Можете вывести изображение на экран, чтобы посмотреть, как оно выглядит.

8. Создайте элемент img и добавьте его в область вывода. Задайте путь src для значения dataURL.
9. Чтобы настроить функцию загрузки изображения, создайте тег привязки, добавьте его в любое место внутри элементов HTML-страницы и напишите имя файла. Уникальные имена файлов можно задавать с помощью Math.random(). Установите гиперссылку для загружаемого атрибута и путь href для пути dataURL. Вызовите нажатие методом click(). После нажатия удалите элемент ссылки.

Вопросы для самопроверки

1. Какие из приведенных ниже записей являются правильными при подготовке к рисованию?

```
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");
var canvas = document.getElementById("canvas");
var ctx = getContext("canvas");
var ctx = canvas.getContext("canvas");
```

2. Что сделает следующий код:

```
<canvas id="canvas"></canvas>
<script>
    var canvas = document.getElementById("canvas");
    var ctx = canvas.getContext("2d");
    canvas.height = 600;
    canvas.width = 500;
    ctx.beginPath();
    ctx.fillStyle = "red";
    ctx.arc(50, 50, 50, 0, Math.PI * 2);
    ctx.fill();
    ctx.closePath();
</script>
```

   1) ничего, код содержит ошибки;
   2) нарисует красный квадрат;
   3) нарисует красный круг;
   4) нарисует половину круга?

3. Какие три метода требуются для рисования линии внутри canvas? В каком порядке они должны быть использованы?

Резюме

В этой главе мы обсудили множество замечательных дополнений к нашему JavaScript-инструментарию, которые можно задействовать благодаря использованию HTML5. Новые навыки действительно расширят ваши возможности при создании интерактивных веб-приложений. Мы начали с локального объекта, который позволил нам загружать и читать файлы с помощью нескольких методов (таких как `readAsText()`). Затем рассмотрели, как получить местоположение пользователя, — данная опция может быть полезна для персонализации предложений, например при поиске ресторанов или парковочных мест.

Принципы работы с элементом `canvas` были еще одним удивительным дополнением к тому, что мы уже можем делать с веб-страницами. Холсты позволяют нам рисовать, писать текст, добавлять изображения (путем рисования и загрузки) и создавать полноценные анимации. Все это можно сделать с помощью методов, прописываемых в `canvas`.

Затем мы рассмотрели медиа на странице: добавление аудио и видео. Наконец, мы обсудили тему цифровой доступности, чтобы ваш веб-сайт был открыт и понятен для всех как с программой чтения, так и без нее.

И сейчас мы наконец-то можем сказать: ура, вы сделали это! Вы изучили так много базовых и продвинутых тем веб-разработки. В заключительной главе мы подскажем следующие шаги, сделав которые вы выведете свои навыки на новый уровень, за рамки чистого JavaScript, которому и посвящена эта книга.

15 Дальнейшие шаги

Как далеко вы зашли! Вы уже прекрасно работаете со строительными блоками JavaScript, умеете создавать приложения, писать умные скрипты и читать много кода. Это прекрасная основа для дальнейшего роста. Давайте теперь перенесем все то, чему вы научились, на следующий уровень, попрактиковавшись и выяснив, что конкретно вас интересует из бесконечных возможностей, которые предлагает JavaScript.

В этой главе мы не будем давать материал подробно: детали скоро устареют, к тому же в интернете вы найдете бесконечный запас актуальной информации и прекрасную подборку хорошо продуманных руководств. Скорее всего, к тому времени, как вы взяли в руки эту книгу, рекомендуемые нами фреймворки и библиотеки безнадежно устарели. Хотя велика вероятность, что следующие крупные проекты все равно будут базироваться на концепциях, описанных здесь. И это хорошая новость.

Данная глава покажет вам, куда еще вы можете вырасти в сфере JavaScript. Мы рассмотрим следующие темы:

- библиотеки и фреймворки;
- основы бэкенда;
- дальнейшие шаги.

Решения упражнений и проектов, а также ответы на вопросы для самопроверки находятся в приложении.

Библиотеки и фреймворки

Библиотеки — это предварительно запрограммированные модули JavaScript, которые вы можете использовать для ускорения процесса разработки. Обычно они заточены под одну конкретную задачу. Фреймворки очень похожи на библиотеки: они тоже прописаны заранее, но вместо того, чтобы делать за вас только что-то одно, они решают целый ряд задач. Вот почему они и называются фреймворками: они действительно предоставляют вам надежную базу, но, прежде чем начать ими

пользоваться, файлы, как правило, нужно определенным образом структурировать. Фреймворк — это обычно набор библиотек, которые дают вам «все в одном» (или по крайней мере «много в одном»). В будущем вы даже наверняка обнаружите, что используете внешние библиотеки поверх фреймворков.

Приведем пример из жизни. Автомобиль можно сконструировать с нуля и создать абсолютно каждую его деталь самостоятельно — этим мы в значительной степени и занимались в данной книге до сих пор. Благодаря библиотекам мы получаем готовые детали — полностью собранные автомобильные кресла, которые можно сразу установить на раму автомобиля. Фреймворк дал бы нашему автомобилю каркас со всеми необходимыми деталями — вероятно, такая машина даже могла бы ездить. Нам в таком случае оставалось бы только убедиться, что в автомобиле есть все необходимое, при надобности подстраиваясь под уже имеющийся у нас фреймворк и продолжая работу в том же духе.

Используя библиотеки и фреймворки, мы бы закончили наш автомобильный проект гораздо быстрее. Кроме того, в процессе работы возникало бы намного меньше проблем, поскольку готовые части хорошо протестированы многими разработчиками. Если бы мы конструировали собственные автомобильные кресла с нуля, после года вождения они наверняка бы стали неудобными, чего бы не случилось с тщательно проверенным стандартным решением.

Таким образом, библиотеки и фреймворки не просто ускоряют процесс разработки, а еще и дают нам для этого более стабильные и проверенные решения. Неужели у них нет никаких недостатков? Что ж, они, конечно, есть. Наиболее важным из них, вероятно, является отсутствие гибкости, ведь при работе с фреймворками вам придется придерживаться их структуры. Но в какой-то степени данное свойство можно считать преимуществом: в таком случае от вас наверняка потребуется хорошо структурированный код, что улучшит ваш стиль кодирования.

Еще один недостаток — всякий раз, когда обновляется используемая вами платформа или библиотека, вам придется обновлять и свое приложение. Это очень важно, особенно когда обновления касаются проблем безопасности. С одной стороны, фреймворки и библиотеки очень надежны, но, поскольку они так распространены, хакеры быстро находят в них слабые места. Найденные лазейки дают им доступ ко многим приложениям, включая ваше. С другой стороны, собственноручно написанный код, вероятно, все равно был бы намного слабее среднего фреймворка.

Тем не менее во многих случаях взлом созданного вами с нуля приложения может оказаться для хакеров слишком дорогостоящим. Вы, скорее всего, не собираетесь платить им огромную сумму выкупа за простой хобби-проект в интернете, в котором и данных особых-то нет. В то время как скрипты в составе часто применяемого фреймворка, которые пытаются использовать его слабость на случайном количестве веб-сайтов, являются обычным явлением. Чтобы свести к минимуму риск, своевременно обновляйте связи и зависимости и следите за сообщениями владельца вашей библиотеки или фреймворка.

Библиотеки

Технически с фреймворками и библиотеками можно сделать не больше, чем без них, но только если оставить вне уравнения значение потраченного времени. Фреймворки и библиотеки позволяют нам реализовывать проекты намного быстрее и качественнее, и именно поэтому они так популярны.

Далее мы обсудим несколько наиболее распространенных библиотек. Это определенно не полный список: через год популярными могут стать уже другие библиотеки или фреймворки. Тем не менее он даст вам прочную основу для дальнейшего продвижения в карьере разработчика.

Многие библиотеки можно включить в страницу, добавив тег `script` в начало HTML-кода:

```
<script src="https://linktolibrary.com/librarycode.js"></script>
```

Рассмотрим несколько известных библиотек.

jQuery

jQuery, пожалуй, самая популярная библиотека JavaScript. Раньше использовать ее было очень удобно, ведь она внедрялась в последнюю версию JavaScript конкретного браузера. Теперь это просто иной способ написания некоторых знакомых вам операций. Вы можете легко распознать jQuery по количеству знаков доллара в коде. А еще можно ввести в консоль веб-сайта `$` или `jQuery` — если сайт использует данную библиотеку, он вернет объект jQuery. Библиотека jQuery в основном ориентирована на выбор HTML-элементов из DOM, взаимодействие и управление ими. Применение выглядит как-то так:

```
$(selector).action();
```

Знаком доллара вы указываете, что хотите запустить jQuery, а с помощью селектора выбираете элемент в HTML. Знаки здесь немного похожи на CSS.

1. Простое строковое значение нацелено на HTML-элемент: `$("p")`.
2. Точка перед словом или фразой значит то, что вы хотите выбрать все элементы с определенным классом: `$(".special")`.
3. Хештег указывает целью элемент с определенным идентификатором: `$("#unique")`.
4. Вы также можете использовать любой другой селектор CSS, в том числе более сложные связанные селекторы.

Вот пример, где библиотека jQuery импортируется в `script`:

```
<html>
  <head>
```

```
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/
    jquery.min.js"></script>
  </head>
  <body>
    <p>Let's play a game!</p>
    <p>Of hide and seek...</p>
    <p class="easy">I'm easy to find!</p>
    <button id="hidebutton">Great idea</button>
    <button id="revealbutton">Found you!</button>
    <script>
      $(document).ready(function () {
        $("#hidebutton").click(function () {
          $("p").hide();
        });
        $("#revealbutton").click(function () {
          $(".easy").show();
        });
      });
    </script>
  </body>
</html>
```

Страница выглядит следующим образом (рис. 15.1).

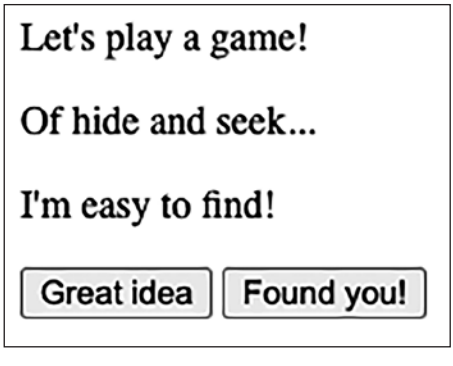


Рис. 15.1. Страница с простым скриптом jQuery

Когда вы нажмете кнопку Great idea, все абзацы будут скрыты. Это делается внутри события, которое было добавлено с помощью jQuery. Сначала мы выбрали кнопку с идентификатором `hidebutton`, затем вызвали на ней функцию `click`, определяющую, что произойдет при нажатии. В этой функции мы заявляем, что выделим все элементы `p` и скроем их. `hide` — это специальная функция jQuery, которая добавляет стиль `display:none` к элементу HTML.

Следовательно, после нажатия кнопки все абзацы исчезают. Когда мы нажимаем на Found you!, возвращается только один, последний абзац с надписью I'm easy to find. Так происходит, потому что при нажатии кнопки с идентификатором `revealbutton`

код выбирает все элементы с классом `easy` и удаляет стиль `display:none` с помощью функции jQuery `show`.

Библиотека jQuery на самом деле:

1) позволяет писать код без селекторов;
2) имеет дополнительные функции или другие (обычно более короткие) названия функций для управления HTML-элементами.

Вы можете использовать jQuery в своем коде, но это не расширит ваши JavaScript-возможности, а лишь позволит написать одно и то же действие меньшим количеством кода. jQuery представляла высокую ценность, когда браузеры были менее стандартизированы, поэтому она была так популярна. Тогда использование данной библиотеки фактически обеспечило стандартизацию одного и того же JavaScript-кода для нескольких браузеров. В настоящее время от нее мало пользы: если вы собираетесь писать новый код, лучше сразу работать с JavaScript. Однако велика вероятность того, что при работе со старым кодом вы все же столкнетесь с jQuery, поэтому знание особенностей ее работы определенно поможет вам в подобных случаях.

На момент написания книги документы по jQuery можно было найти здесь: https://api.jquery.com/.

D3

D3 означает *data-driven documents* (документы, управляемые данными). Это библиотека JavaScript, которая помогает управлять документами на основе данных и которую можно использовать для визуализации данных с помощью HTML, SVG и CSS. Очень удобный способ работы с информационными панелями, которые должны содержать какое-либо представление данных.

Используя D3, вы можете создать практически любой вид графика со множеством функций. Это может выглядеть довольно устрашающе, потому что необходимо задать все настройки для графической фигуры. Погрузившись в изучение графика и разбив его на части, вы гарантированно преодолеете любые трудности. Ниже вы найдете очень простой пример добавления трех сфер в SVG с помощью D3:

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://d3js.org/d3.v7.min.js"></script>
    <style>
        svg {
            background-color: lightgrey;
        }
```

```
    </style>
</head>
<body>
    <svg id="drawing-area" height=100 width=500></svg>
    <script>
        let svg = d3.select("#drawing-area");
        svg.append("circle")
            .attr("cx", 100).attr("cy", 50).attr("r", 20).style("fill", "pink");
        svg.append("circle")
            .attr("cx", 200).attr("cy", 20).attr("r", 20).style("fill", "black");
        svg.append("circle")
            .attr("cx", 300).attr("cy", 70).attr("r", 20).style("fill", "grey");
    </script>
</body>

</html>
```

Библиотека D3 импортируется в первом теге `script`. Переменная `svg` создается с помощью метода `d3.select`, применяемого к `svg` с идентификатором `drawing-area`.

Мы не отдаем должное возможностям D3. В данном случае D3 действительно ненамного полезнее, чем простая реализация аналогичного функционала с помощью элемента `canvas`. Тем не менее вы можете создавать красивые анимации данных: эффект масштабирования, сортируемую гистограмму, вращающуюся сферу и многое другое. Однако этот код занял бы несколько страниц.

На момент написания книги документы по D3 можно было найти здесь: https://devdocs.io/d3~4/.

Underscore

Underscore — это библиотека JavaScript, которую можно охарактеризовать как инструментарий для функционального программирования. Функциональное программирование можно считать парадигмой программирования, вращающейся вокруг использования описательных функций в последовательности, а не по отдельности. *Объектно-ориентированное программирование (ООП)* также является парадигмой, полностью посвященной объектам и их состоянию, а также данным, которые могут быть инкапсулированы и скрыты от внешнего кода. В функциональном программировании функции настолько важны, что разработчики должны беспокоиться об их корректной работе, но в этой парадигме есть и иные аспекты, достойные внимания. Эти функции постоянно выполняют одно и то же с разными аргументами, и их можно легко связать в последовательность.

Библиотека Underscore предлагает множество функций для повседневного программирования: например, `map`, `filter`, `invoke` и функции для тестирования. Вот небольшой фрагмент кода, демонстрирующий одну из функций Underscore, которая

создает всплывающие уведомления для всех элементов в массиве (в нашем случае для 1, 2 и 3):

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://cdn.jsdelivr.net/npm/underscore@1.13.1/
    underscore-umd-min.js"></script>
</head>

<body>
    <script>
        _.each([1, 2, 3], alert);
    </script>
</body>

</html>
```

Существует множество других функций для сортировки, группировки и преобразования элементов, получения случайного значения, значения текущего времени и многого другого.

Данный фрагмент, вероятно, пояснил вам, почему библиотека называется именно так: мы получаем доступ к функциям Underscore с помощью символа подчеркивания. Однако сначала вам все же придется установить библиотеку: в противном случае интерпретатор не поймет синтаксис.

На момент написания книги документы по Underscore можно было найти здесь: https://devdocs.io/underscore/.

React

React — это последняя библиотека, которую мы обсудим. Если вы думаете, что React — фреймворк, вы и правы и не правы одновременно. Причина, по которой мы считаем React библиотекой, заключается в следующем: чтобы достичь точки, в которой все это будет похоже на фреймворк, вам придется использовать дополнительные библиотеки.

React применяется для создания красивых и динамичных пользовательских интерфейсов. Она разбивает страницы на разные компоненты; данные передаются и обновляются между компонентами по мере их изменения. Вот очень простой пример, охватывающий самую малость того, что может сделать React. Следующий HTML-код отобразит такое предложение на странице — Hi Emile, what's up?:

```
<div id="root"></div>
```

Это будет возможно в связке со следующим JavaScript-кодом:

```
ReactDOM.render(
  <p> Hi Emile, what's up?</p>,
  document.getElementById('root');
);
```

Код будет работать только в том случае, когда библиотека React доступна. Он будет отображать DOM, заменив `innerHTML` элемента `div` первым аргументом функции `render`. Мы можем сделать это, добавив React в элемент `script` и ничего не устанавливая в нашей системе. Завершенный скрипт выглядит следующим образом:

```
<!DOCTYPE html>
<html>

<head>
    <script src="https://unpkg.com/react@17/umd/react.development.js"
      crossorigin></script>
    <script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
      crossorigin></script>
</head>

<body>
    <div id="root"></div>
    <script>
        let p = React.createElement("p", null, "Hi Emile, what's up?");
        ReactDOM.render(
            p,
            document.getElementById("root");
        );
    </script>
</body>
</html>
```

Этот код выдаст Hi Emile, what's up? на странице с помощью элементов React, прописанных вручную в теге `script`. Однако это не тот способ, который вам стоит применять в крупных проектах. Гораздо полезнее настроить React и все, что вам нужно, с помощью такого менеджера пакетов, как *Node Package Manager (NPM)*. Это позволит вам легко управлять всеми зависимостями и держать ваш код в чистоте.

На момент написания книги более подробная информация о React находилась здесь: https:// reactjs.org/docs/getting-started.html.

Фреймворки

Фреймворки более сложны и чаще всего требуют их установки на свой компьютер. (Как это сделать, рассказывается в онлайн-документации конкретной платформы.) И всякий раз, когда вы закончите разработку и захотите запустить программу, вам придется выполнить команду, перерабатывающую код во что-то понятное браузеру. Мы «обслуживаем» приложение, когда делаем это.

Vue.js

Vue.js — это легкий JavaScript-фреймворк. Его можно использовать для создания пользовательских интерфейсов и одностраничных приложений — *single-page applications (SPAs)*. В способе написания пользовательских интерфейсов с помощью Vue.js может быть непросто разобраться с первого раза. Взгляните на следующий образец кода:

```
<!DOCTYPE html>
<html>
  <script src="https://cdn.jsdelivr.net/npm/vue"></script>

  <body>
    <div id="app">
      <p v-if="!hide">
        Let's play hide and seek. <br />
        Go to the console and type: <br />
        obj._data.hide = true <br />
      </p>
    </div>

    <script>
      let obj = new Vue({
        el: "#app",
        data: {
          hide: false,
        },
      });
    </script>
  </body>
</html>
```

Это простая HTML-страница, импортирующая JavaScript-ссылку из Vue. В HTML-коде тега `<p>` происходит что-то странное — там присутствует элемент `v-if`. Данный элемент будет отображаться только в том случае, если условие в нем равно `true`. Тогда он просматривает свойство `hide` объекта данных в нашем экземпляре Vue. Если изменить значение `hide` на `true`, отрицаемый оператор `hide` примет значение `false` — и элемент исчезнет. Мы могли бы сделать это и без Vue, но тогда нам бы пришлось указать событие JavaScript для изменения значения и использовать JavaScript для редактирования CSS, чтобы скрыть абзац.

Вам также могут встретиться новые HTML-элементы. Все потому, что это не обычные HTML-элементы, а, скорее, полученные из Vue, который позволяет вам определять свои собственные компоненты. Вы можете столкнуться с HTML, который выглядит следующим образом:

```
<div id="custom-component">
  <maaike></maaike>
</div>
```

Когда вы откроете веб-страницу, связанную с этим кодом, увидите следующее:

```
Maaike says: good job!
```

Причина не в том, что HTML знает, как это сделать. Так происходит потому, что есть фрагмент, определяющий компонент `maaike`:

```
<script>
    Vue.component("maaike", {
        template: "<p>Maaike says: good job!</p>",
    });

    new Vue({ el: "#app" });
</script>
```

Мы создали новый компонент Vue, и он действительно может содержать данные и иметь функцию. Но перед вами компонент базового уровня: он нужен для того, чтобы проиллюстрировать возможность добавления HTML-шаблонов в свойство `template`. Там есть определенный абзац. Когда веб-страница будет загружена, компонент `<maaike>` будет заменен тем, что есть в шаблоне.

Содержимое одной страницы может собираться из многих файлов. Обычно все эти компоненты содержатся каждый в своем файле. Официальных инструментов Vue существует гораздо больше; вы познакомитесь с ними, как только погрузитесь в Vue.js. На самом деле для начала работы с фреймворками это отличный вариант, поскольку достаточно хорошо видно, что в нем происходит. Vue является отличной отправной точкой для знакомства с принципами работы фреймворков в целом.

На момент написания книги все документы по Vue можно было найти здесь: https://v3.vuejs.org/guide/introduction.html.

Angular

Angular — это фреймворк, созданный и (в настоящее время) поддерживаемый Google. Angular намного объемнее, чем Vue.js, и его можно считать полным пакетом. Angular занимает больше места на диске, а чем больше занято места на диске, тем медленнее фреймворк компилируется и устанавливается. Код Angular на самом

деле не так уж сильно отличается от Vue.js; тем не менее он использует TypeScript вместо JavaScript — это расширенный вариант JavaScript и транспилируется на JavaScript, но он более строг и также имеет другой синтаксис.

Angular можно распознать по атрибутам `ng` в HTML. Вот фрагмент кода, который покажет все задачи в списке дел (при условии, конечно, что остальной код написан верно):

```
<ul>
  <li ng-repeat="task in tasks">
    {{task}}<span ng-click="deleteTask($index)">Done</span>
  </li>
</ul>
```

Атрибут `ng-repeat` указывает повторное действие, которое для каждой задачи в списке дел должно создавать элемент `<li>`. Также `task` можно использовать в качестве переменной внутри `<li>`, указав следующим образом: `{{ task }}`.

Есть еще одна специфичная для Angular функция — `ng-click`, которая сообщает фреймворку, что делать при щелчке на элементе. Чем-то похоже на событие `onclick` в JavaScript, только теперь его можно привязывать динамически. Иными словами, при написании кода вам не обязательно знать об `onclick`. Вы, конечно, можете добиться того же и в JavaScript, указав события, которые приведут к изменениям атрибута `onclick` (и всего элемента, если необходимо), но для этого потребуется намного больше кода. В принципе, данное утверждение верно для всего, что есть в Angular, — его содержимое можно написать с помощью одного только JavaScript-функционала, но для этого потребуется намного больше усилий (и это еще, в зависимости от сложности ситуации, мягко сказано).

На момент написания книги все документы по Angular можно было найти здесь: https://angular.io/docs.

Научиться работать с библиотеками и фреймворками, такими как React, Angular или Vue, — очень логичный и даже обязательный шаг, если вы хотите стать разработчиком интерфейса. По нашему мнению, сложность этих вариантов на самом деле не так уж сильно различается. Какой из них вы посчитаете наилучшим, зависит от места, где вы хотите работать, и региона, в котором вы находитесь (потому что для этих фреймворков и библиотек существуют и региональные предпочтения).

Изучение бэкенда

До сих пор мы имели дело только с интерфейсом. Внешний интерфейс (фронтенд) — это часть, выполняемая на стороне клиента. Это может быть любое пользовательское устройство: телефон, ноутбук или планшет. Однако, чтобы сайты могли делать что-то интересное, нам также нужна программная часть сервера (бэкенд).

Например, когда вы входите в личный кабинет на каком-либо веб-сайте, этот веб-сайт должен проверить, существует ли пользователь с введенными вами данными.

Такую работу как раз и проводит бэкенд. Код выполняется не на устройстве пользователя, а на каком-то другом сервере, который часто принадлежит или арендуется компанией, размещающей веб-сайт. «Размещение» в данном контексте обычно означает, что сайт делают доступным для Всемирной паутины. И вот как раз размещение на сервере дает возможность принимать внешние запросы по URL-адресу.

Код на сервере выполняет множество функций — все они связаны с углубленной логикой и данными. Например, в интернет-магазине есть множество товаров, которые поступают из базы данных. Сервер получает элементы базы данных, анализирует HTML-шаблон и отправляет код HTML, CSS и JavaScript клиенту.

То же самое относится и ко входу в систему — когда вы вводите имя пользователя и пароль и нажимаете Войти, запускается код на сервере. Он будет сверять введенные вами данные с записями в базе. Если ваши данные правильные, он отправит вас обратно на страницу для зарегистрированных пользователей. Если же вы ввели неверные данные, код вернет клиенту сообщение об ошибке.

Далее мы рассмотрим основы взаимодействия между интерфейсом и серверной частью и покажем вам, как можно использовать JavaScript для написания серверного кода, в том числе с помощью Node.js.

API

API (*Application Programming Interface*, или *интерфейс программирования приложений*) — это, по сути, интерфейс для кода, написанный с использованием большего количества кода. Запрос может быть отправлен в API с использованием (скажем) URL-адреса, что вызовет определенный фрагмент кода, который даст определенный ответ.

Да, описание довольно абстрактное, поэтому давайте воспользуемся примером. Если бы у нас был сайт отеля, разумно, чтобы там была функция онлайн-бронирования. Для ее реализации потребуется какой-то API. Всякий раз, когда пользователь будет заполнять все поля и нажимать Подтвердить бронирование, система будет связываться с API. Будет вызван определенный URL-адрес — конечная точка, куда отправятся введенные пользователем данные (в нашем случае пусть это будет www.api.hotelname.com/rooms/book). API обработает и проверит наши данные; когда все будет готово, он сохранит бронирование номера в базе и, возможно, отправит нашему гостю подтверждение по электронной почте.

Когда один из служащих отеля проверит бронирование, API будет вызван еще раз с помощью одной из конечных точек (она может выглядеть, например, так: www.api.hotelname.com/reservations). Сначала это позволит убедиться, есть ли

у сотрудника, вошедшего в систему, нужные права доступа; если есть, код извлечет все бронирования для выбранного диапазона дат из базы данных и отправит страницу с результатами обратно сотруднику. Таким образом, API-интерфейсы являются точками соединения между логикой, базой данных и интерфейсом.

API работают с вызовами *HTTP* (*Hypertext Transfer Protocol*, или *протокол передачи гипертекста*). HTTP — это просто протокол для связи между двумя сторонами: клиентом и сервером или сервером и другим сервером (где запрашивающий сервер действует как клиент). Протокол должен придерживаться определенных конвенций и правил, которых ожидает другая сторона и на которые она будет реагировать соответствующим образом. В эти правила входит, например, использование определенного формата для указания заголовков, методов GET — для получения информации, методов POST — для создания новой информации на сервере, и методов PUT — для изменения этой информации.

С помощью API можно решить больше задач: например, ваш компьютер и принтер также взаимодействуют через API. Однако с точки зрения JavaScript это не слишком актуально.

Вы увидите, как использовать API, в подразделе «AJAX». Вы также можете написать свои собственные API: базовую информацию о том, как это сделать, можно найти в подразделе «Node.js».

AJAX

AJAX расшифровывается как *Asynchronous JavaScript and XML* (асинхронный JavaScript и XML), что на самом деле не совсем верно, поскольку в настоящее время вместо XML чаще используется JSON. Мы применяем AJAX для выполнения вызовов из интерфейса в бэкенд без обновления страницы (асинхронно). AJAX — это не язык программирования или библиотека; это сочетание встроенного объекта `XMLHttpRequest` в браузере и языка JavaScript.

В настоящее время вы, будучи фронтенд-разработчиком, вероятно, не будете использовать чистый AJAX. Однако он применяется в фоновом режиме, поэтому все же полезно узнать, как все это работает. Вот как выглядит вызов бэкенда с помощью AJAX:

```
let xhttp = new XMLHttpRequest();
let url = "some valid url";
xhttp.load = function () {
    if (this.status == 200 && this.readyState == 4) {
        document.getElementById("content").innerHTML = this.responseText;
    }
};
```

```
xhttp.open("GET", url, true);
xhttp.send();
```

Этот пример не рабочий (у нас нет действующего URL-адреса), тем не менее он отлично демонстрирует функционирование AJAX. Код устанавливает, что ему нужно сделать, когда запрос загружен, в нашем случае заменяя HTML внутри элемента `content` на то, что возвращает ссылка (это может быть ссылка на файл или на какой-либо API, вызывающий базу данных). Он может давать разные ответы, если в базе есть (или нет) другие данные. Ответ написан в формате JSON, но он также может быть в формате XML (в зависимости от того, как был закодирован сервер).

В настоящее время более распространенным является использование для AJAX-запросов *Fetch API*. Мы можем сделать что-то подобное с помощью `XMLHttpRequest`, но в данном случае наш набор функций будет более гибким и мощным. Например, в следующем коде мы получаем данные, размещенные по URL-адресу, преобразуем их в JSON с помощью метода `json()` и выводим на экран:

```
let url = "some valid url";
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data));
```

Fetch API работает с промисами, которые на данный момент уже должны выглядеть для вас знакомо. После того как промис будет выполнен, с помощью `then` создается новый, и после его обработки выполняется следующий `then`.

На момент написания книги больше информации по AJAX можно было найти здесь: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started.

Практическое занятие 15.1

Создайте файл JSON и, используя `fetch`, верните результаты в виде объекта, который можно применить в вашем коде JavaScript.

1. Создайте объект JSON и сохраните его в файле `list.json`.
2. Используя JavaScript, назначьте имя файла и путь переменной `url`.
3. С помощью `fetch` сошлитесь на URL-адрес файла. Верните результат в виде JSON.
4. Как только объект ответа будет готов, выполните итерацию по данным и выведите на экран результаты по каждому элементу в файле JSON.

Node.js

Мы можем писать API на JavaScript, используя Node.js. Node.js — это очень умная виртуальная машина, которая взяла движок Google JavaScript, расширила его и позволила запускать JavaScript на серверах, взаимодействуя с протоколом файловой системы и HTTP как раз с помощью JavaScript. Благодаря этому решению мы можем использовать JavaScript для бэкенда. Следовательно, вы можете писать на одном языке как бэкенд, так и фронтенд (наряду с HTML и CSS). Без Node.js вам пришлось бы воспользоваться в серверной части другим языком, например PHP, Java или C#.

Чтобы запустить Node.js, нужно настроить его, а затем выполнить команду `node nameOfFile.js`. (В официальной документации Node.js есть все инструкции по настройке.) Часто для полной готовности требуется что-то загрузить и установить.

На момент написания книги инструкции по загрузке Node.js были доступны по адресу: https://nodejs.org/en/download/.

Вот пример кода для Node.js, получающего HTTP-вызовы:

```
const http = require("http");

http.createServer(function(req, res){
    res.writeHead(200, {"Content-Type": "text/html"}); //статус заголовка
    let name = "Rob";
    res.write(`Finally, hello ${name}`); //тело
    res.end();
}).listen(8080); //слушаем порт 8080

console.log("Listening on port 8080... ");
```

Мы начинаем с импорта модуля `http`. Это файл внешнего кода, который необходимо импортировать, чтобы код запустился. Модуль `http` поставляется с Node.js, но, возможно, потребуется установить дополнительные модули. Для этого вы будете использовать менеджер пакетов (например, NPM), который поможет установить все зависимости и сможет управлять всеми различными версиями внешних модулей.

Приведенный выше код настраивает сервер, который прослушивает порт `8080` и всякий раз, когда к нему обращаются, возвращает `Finally, hello Rob`. Мы создаем сервер с помощью метода `createServer` на импортированном модуле `http`. Затем говорим, что должно произойти, чтобы сервер был вызван. После чего отвечаем статусом `200` (с указанием `OK`) и пишем `Finally, hello Rob` в ответе. Далее указываем порт по умолчанию `8080` в качестве порта прослушивания.

В этом примере используется встроенный модуль `http` для Node.js, который очень эффективен при создании API. Вам определенно стоит с ним поработать. Возмож-

ность писать собственные API позволит разрабатывать полноценные приложения. А все станет еще проще, когда мы добавим фреймворк Express в эту смесь.

Использование фреймворка Express Node.js

Node.js — это не фреймворк и не библиотека. Это виртуальная машина. Следовательно, она может запускать и интерпретировать JavaScript-код. Существуют различные фреймворки для Node.js, и в настоящее время Express является самым популярным из них.

Вот очень простое приложение с применением Express. Снова вначале требуется настроить Node.js, затем добавить модуль Express (если вы используете NPM, вам поможет команда `npm install express`) и запустить его действием `node nameOfRootFile.js`:

```
const express = require('express');
const app = express();

app.get('/', (request, response) => {
  response.send('Hello Express!');
});

app.listen(3000, () => {
  console.log('Express app at http://localhost:3000');
});
```

После запуска этого кода и перехода к `localhost:3000` (при условии, конечно, что вы запускаете его на `localhost`) вы получите сообщение Hello Express! в вашем браузере. В терминале, где вы запускаете приложение `Node`, после загрузки оно выведет сообщение, содержащееся в операторе `console.log`.

Больше информации можно найти в документации Node.js, которая на момент написания книги находилась по следующему адресу: https://nodejs.org/en/docs/.

Для информации по модулю Express перейдите сюда: https://expressjs.com/en/5x/api.html.

Дальнейшие шаги

Из этой книги вы многое узнали о JavaScript. Благодаря последней главе у вас должно было появиться представление о возможных дальнейших шагах, которые можно предпринять. Мы не рассматривали подробно темы текущей главы, поскольку о каждой из них можно написать (и уже написаны) целые книги. Но вы наверняка уже поняли, куда двигаться и что учитывать, принимая решение о подъеме на следующую ступень.

Лучший способ обучения — практика. Поэтому настоятельно рекомендуем просто придумать интересный проект и попытаться реализовать его. А может, получив знания из этой книги, вы даже почувствуете себя готовыми к начальной должности JavaScript-разработчика! Вы также можете выполнять учебные задания из онлайн-пособий или поработать в проектной команде в качестве младшего специалиста. Используйте фриланс-платформы, такие как Upwork или Fiverr, для поиска проектов — найти их, правда, бывает довольно трудно, поэтому все же вам стоило бы для начала изучить какой-либо фреймворк или расширить свой опыт с Node.js. Тем не менее это можно сделать и в процессе работы, если сможете убедить нанимателя в своих навыках и потенциале.

Проекты текущей главы

Работа с JSON

Создайте локальный файл JSON, подключитесь к данным и JSON. Выведите данные из файла JSON на экран.

1. Создайте файл с расширением JSON и назовите его `people.json`.
2. Создайте в `people.json` массив, который будет содержать множество объектов. Каждый элемент массива должен представлять собой объект с одинаковой структурой. Назовите свойства `first`, `last` и `topic`. Убедитесь, что вы используете двойные кавычки вокруг имен и значений свойств, так как это правильный синтаксис JSON.
3. Добавьте три или более записи в массив, используя одну и ту же структуру объекта для каждого элемента.
4. Создайте HTML-файл и добавьте JavaScript-файл к нему. В файле JavaScript используйте `people.json` в качестве URL. Подключитесь к URL и получите данные с помощью `fetch`. Поскольку это файл в формате JSON, получив данные ответа, их можно отформатировать в JSON с помощью метода `.json()` в `fetch`.
5. Выведите полное содержимое данных на экран.
6. В цикле переберите элементы данных и выведите значения на экран с помощью `foreach`. Можете использовать шаблонный литерал и выводить каждое значение.

Создание списков

Создайте список, который будет находиться в локальном хранилище, чтобы даже при обновлении страницы данные сохранялись в браузере. Если локальное хранилище пусто при первой загрузке страницы, задайте файл JSON, который будет загружен в локальное хранилище и сохранен как список по умолчанию.

1. Настройте HTML-файл, добавив `div` для вывода результатов и поле ввода с кнопкой, которую можно нажать.

2. С помощью JavaScript добавьте элементы страницы в качестве объектов, которые можно использовать в коде.
3. Создайте файл JSON по умолчанию (может быть пустым) и добавьте в свой JavaScript-код путь к нему, используя переменную `url`.
4. Добавьте прослушиватель событий к кнопке, которая будет запускать функцию `addToList()`.
5. В функции `addToList()` проверьте длину значения входного поля: равно ли оно `3` или более. Если это так, создайте объект с именем и значением поля ввода. Напишите глобальную переменную `myList` для хранения списка. В функции `addToList()` вставьте новый объект данных в `myList`.
6. Напишите функцию `maker()`. Она создаст элемент страницы и добавит в него текст, который, в свою очередь, будет присоединен к выходному элементу. Вызовите `maker()`, чтобы добавить новый элемент в функцию `addToList()`.
7. Сохраните также элемент в локальном хранилище, чтобы визуальное содержимое `myList` было синхронизировано со значением, которое в хранилище уже есть. Чтобы сделать это, создайте функцию `savetoStorage()` и вызывайте ее каждый раз, когда обновляете `myList` в скрипте.
8. В функции сохранения значений `savetoStorage()` задайте значение `myList` в `localStorage` с помощью `setItem`. Чтобы сохранить его в `localStorage`, вам нужно будет преобразовать `myList` в строку.
9. Добавьте в код функцию `getItem()`, чтобы получить значение `myList` из хранилища `localStorage`. Настройте глобальную переменную для массива `myList`.
10. Добавьте прослушиватель событий для `DOMContentLoaded`. В рамках этой функции проверьте, загрузил ли `localstorage` значение. Если загрузил, получите `myList` из локального хранилища и преобразуйте его из строки в объект JavaScript. Очистите содержимое вывода. Переберите в цикле элементы `myList` и добавьте их на страницу с помощью ранее созданной функции `maker()`.
11. Если в хранилище `localStorage` отсутствует содержимое, загрузите файл JSON со значениями по умолчанию с помощью `fetch`. Как только данные загружены, добавьте их к глобальному значению `myList`. Переберите в цикле элементы `myList` и выведите их на страницу с помощью `maker()`. Не забудьте после этого вызвать `savetoStorage()`, чтобы хранилище содержало те же элементы списка, что и на странице.

Вопросы для самопроверки

1. Что такое библиотеки и фреймворки JavaScript?
2. Как определить, использует ли веб-страница библиотеку jQuery?
3. Какая библиотека содержит множество функциональных возможностей для управления данными?
4. Как вы можете запустить файл Node.js, когда Node.js установлен?

Резюме

В этой главе мы рассмотрели несколько возможностей для того, чтобы вы продолжили ваше путешествие по JavaScript. Мы начали с обсуждения фронтенда, а также библиотек и фреймворков. Библиотеки и фреймворки — это готовый код, который вы можете использовать в своем проекте. Библиотеки обычно решают одну проблему. Фреймворки же предоставляют стандартное решение, под которое вам, скорее всего, придется подстраивать способ структурирования приложения и которое имеет некоторые ограничения. Тем не менее фреймворки отлично подходят для очень многих решений, которые вы, возможно, захотите применить в своих веб-разработках.

Затем мы перешли к рассмотрению бэкенда. Бэкенд — это код, который выполняется на сервере. Мы можем написать этот код на JavaScript с использованием Node.js. Node.js — это виртуальный движок, который может обрабатывать JavaScript и предоставляет некоторые дополнительные функции, которых не дает обычное использование JavaScript в браузере.

На этом все. Теперь вы должны отлично разбираться в JavaScript. Вы видели все нужные строительные блоки и активно практиковались в небольших упражнениях и более крупных проектах. Тем не менее можно сказать наверняка: как JavaScript-программист, вы никогда не закончите учиться и продолжите удивляться тому, что еще можно сделать, развиваясь в этой сфере.

Не забывайте экспериментировать!

Приложение
Ответы на практические занятия, проекты и вопросы для самопроверки

Глава 1. Начало работы с JavaScript

Практические занятия

Практическое занятие 1.1

```
4 + 10
14

console.log("Laurence");
Laurence
undefined
```

Практическое занятие 1.2

```
<!DOCTYPE html>
<html>

<head>
  <title>Tester</title>
</head>

<body>
  <script>
    console.log("hello world");
```

```
  </script>
</body>

</html>
```

Практическое занятие 1.3

```
<!DOCTYPE html>
<html>

<head>
  <title>Tester</title>
</head>

<body>
  <script src="app.js"></script>
</body>

</html>
```

Практическое занятие 1.4

```
let a = 10; // присвоить переменной значение 10
console.log(a); // эта команда выведет 10 на экран
/*
Это многострочный
комментарий
*/
```

Проект

Создание HTML-файла и привязка JavaScript-файла

```
<!doctype html>
<html>
  <head>
    <title>JS Tester</title>
  </head>
  <body>
    <script src="myJS.js"></script>
  </body>
</html>

// console.log("Laurence");
/*
Это мой комментарий
Лоренс Свекис
*/
```

Вопросы для самопроверки

1. `<script src="myJS.js"></script>`
2. Нет.
3. Открыв с помощью `/*` и `*/`.
4. Закомментировать строку с помощью `//`.

Глава 2. Основы JavaScript

Практические занятия

Практическое занятие 2.1

```
console.log(typeof(str1));
console.log(typeof(str2));
console.log(typeof(val1));
console.log(typeof(val2));
console.log(typeof(myNum));
```

Практическое занятие 2.2

```
const myName = "Maaike";
const myAge = 29;
const coder = true;
const message = "Hello, my name is " + myName + ", I am " + myAge+" years old
and I can code JavaScript: " + coder + ".";
console.log(message);
```

Практическое занятие 2.3

```
let a = window.prompt("Value 1?");
let b = window.prompt("Value 2?");
a = Number(a);
b = Number(b);
let hypotenuseVal = ((a * a) + (b * b))**0.5;
console.log(hypotenuseVal);
```

Практическое занятие 2.4

```
let a = 4;
let b = 11;
let c = 21;
a = a + b;
```

```
a = a / c;
c = c % b;
console.log(a, b, c);
```

Проекты

Конвертер миль в километры

```
// Переведите мили в километры.
// 1 миля равна 1,60934 км.
let myDistanceMiles = 130;
let myDistanceKM = myDistanceMiles * 1.60934;
console.log("The distance of " + myDistanceMiles + " miles is equal to
" + myDistanceKM + " kilometers");
```

Вычислитель индекса массы тела (ИМТ)

```
// 1 дюйм = 2.54 см
// 2,2046 фунта в килограмме
let inches = 72;
let pounds = 180;
let weight = pounds / 2.2046; // в кг
let height = inches * 2.54; // рост в см.
console.log(weight, height);
let bmi = weight/(height/100*height/100);
console.log(bmi);
```

Вопросы для самопроверки

1. Строка.
2. Число.
3. Строка 2.
4. `world`
5. `Hello world!`
6. Все, что введет пользователь.
7. `71`
8. `4`
9. `16` и `536`.
10. `true`
    `false`
    `true`
    `true`
    `false`

Глава 3. Множественные значения JavaScript

Практические занятия

Практическое занятие 3.1

```
const myList = ["Milk", "Bread", "Apples"];
console.log(myList.length);
myList[1] = "Bananas";
console.log(myList);
```

Практическое занятие 3.2

```
const myList = [];
myList.push("Milk", "Bread", "Apples");
myList.splice(1, 1, "Bananas", "Eggs");
const removeLast = myList.pop();
console.log(removeLast);
myList.sort();
console.log(myList.indexOf("Milk"));
myList.splice(1, 0, "Carrots", "Lettuce");
const myList2 = ["Juice", "Pop"];
const finalList = myList.concat(myList2, myList2);
console.log(finalList.lastIndexOf("Pop"));
console.log(finalList);
```

Практическое занятие 3.3

```
const myArr = [1, 2, 3];
const bigArr = [myArr, myArr, myArr];
console.log(bigArr[1][1]);
console.log(bigArr[0][1]);
console.log(bigArr[2][1]);
```

Практическое занятие 3.4

```
const myCar = {
    make: "Toyota",
    model: "Camry",
    tires: 4,
    doors: 4,
    color: "blue",
    forSale: false
};
```

```
let propColor = "color";
myCar[propColor] = "red";
propColor = "forSale";
myCar[propColor] = true;
console.log(myCar.make + " " + myCar.model);
console.log(myCar.forSale);
```

Практическое занятие 3.5

```
const people = {friends:[]};
const friend1 = {first: "Laurence", last: "Svekis", id: 1};
const friend2 = {first: "Jane", last: "Doe", id: 2};
const friend3 = {first: "John", last: "Doe", id: 3};
people.friends.push(friend1, friend2, friend3);
console.log(people);
```

Проекты

Управление массивом

```
theList.pop();
theList.shift();
theList.unshift("FIRST");
theList[3] = "hello World";
theList[2] = "MIDDLE";
theList.push("LAST");
console.log(theList);
```

Каталог продукции компании

```
const inventory = [];
const item3 = {
    name: "computer",
    model: "imac",
    cost: 1000,
    qty: 3
}
const item2 = {
    name: "phone",
    model: "android",
    cost: 500,
    qty: 11
}
const item1 = {
    name: "tablet",
    model: "ipad",
    cost: 650,
    qty: 1
}
```

```
inventory.push(item1, item2, item3);
console.log(inventory);
console.log(inventory[2].qty);
```

Вопросы для самопроверки

1. Да. Вы можете заново присваивать значения внутри массива, объявленного с помощью `const`, но не можете повторно объявить сам массив.
2. Длина.
3. Вывод будет следующим:

```
-1
1
```

4. Можно поступить так:

```
const myArr = [1,3,5,6,8,9,15];
myArr.splice(1,1,4);
console.log(myArr);
```

5. Запись в консоли будет такой:

```
[empty × 10, "test"]
undefined
```

6. Вывод будет выглядеть следующим образом:

```
undefined
```

Глава 4. Логические операторы

Практические занятия

Практическое занятие 4.1

```
const test = false;
console.log(test);
if(test){
    console.log("It's True");
}
if(!test){
    console.log("False now");
}
```

Практическое занятие 4.2

```
let age = prompt("How old are you?");
age = Number(age);
let message;
if(age >= 21){
    message = "You can enter and drink.";
}else if(age >= 19){
    message = "You can enter but not drink.";
}else{
    message = "You are not allowed in!";
}
console.log(message);
```

Практическое занятие 4.3

```
const id = true;
const message = (id) ? "Allowed In" : "Denied Entry";
console.log(message);
```

Практическое занятие 4.4

```
const randomNumber = Math.floor(Math.random() * 6);
let answer = "Something went wrong";
let question = prompt("Ask me anything");
switch (randomNumber) {
    case 0:
        answer = "It will work out";
        break;
    case 1:
        answer = "Maybe, maybe not";
        break;
    case 2:
        answer = "Probably not";
        break;
    case 3:
        answer = "Highly likely";
        break;
    default:
        answer = "I don't know about that";
}
let output = "You asked me " + question + ". I think that " + answer;
console.log(output);
```

Практическое занятие 4.5

```
let prize = prompt("Pick a number 0-10");
prize = Number(prize);
let output = "My Selection: ";
switch (prize){
    case 0:
        output += "Gold ";
    case 1:
        output += "Coin ";
        break;
    case 2:
        output += "Big ";
    case 3:
        output += "Box of ";
    case 4:
        output += "Silver ";
    case 5:
        output += "Bricks ";
        break;
    default:
        output += "Sorry Try Again";
}
console.log(output);
```

Проекты

Игра в рулетку

```
let val = prompt("What number?");
val = Number(val);
let num = 100;
let message = "nothing";
if (val > num) {
    message = val + " was greater than " + num;
} else if (val === num) {
    message = val + " was equal to " + num;
} else {
    message = val + " is less than " + num;
}
console.log(message);
console.log(message);
```

Игра «Проверь друга»

```
let person = prompt("Enter a name");
let message;
switch (person) {
    case "John" :
    case "Larry" :
    case "Jane" :
    case "Laurence" :
    message = person + " is my friend";
    break;
    default :
    message = "I don't know " + person;
}
console.log(message);
```

Игра «Камень — ножницы — бумага»

```
const myArr = ["Rock", "Paper", "Scissors"];
let computer = Math.floor(Math.random() * 3);
let player = Math.floor(Math.random() * 3);
let message = "player " + myArr[player] + " vs computer " +
  myArr[computer] + " ";
if (player === computer) {
    message += "it's a tie";
} else if (player > computer) {
    if (computer == 0 && player == 2) {
        message += "Computer Wins";
    } else {
        message += "Player Wins";
    }
} else {
    if (computer == 2 && player == 0) {
        message += "Player Wins";
    } else {
        message += "Computer Wins";
    }
}
console.log(message);
```

Вопросы для самопроверки

1. one
2. this is the one
3. login
4. Welcome, that is a user: John

5. `Wake up, it's morning`
6. Результат:
   - `true`;
   - `false`;
   - `true`;
   - `true`.
7. Результат:

```
100 was LESS or Equal to 100
100 is Even
```

Глава 5. Циклы

Практические занятия

Практическое занятие 5.1

```
const max = 5;
const ranNumber = Math.floor(Math.random() * max) + 1;
// console.log(ranNumber);
let correct = false;
while (!correct) {
    let guess = prompt("Guess a Number 1 - " + max);
    guess = Number(guess);
    if (guess === ranNumber) {
        correct = true;
        console.log("You got it " + ranNumber);
    } else if (guess > ranNumber) {
        console.log("Too high");
    } else {
        console.log("Too Low");
    }
}
```

Практическое занятие 5.2

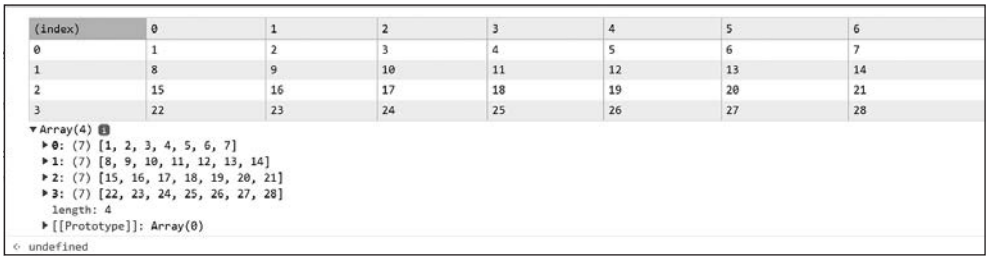
```
let counter = 0;
let step = 5;
do {
    console.log(counter);
    counter += step;
}
while (counter <= 100);
```

Практическое занятие 5.3

```
const myWork = [];
for (let x = 1; x < 10; x++) {
    let stat = x % 2 ? true : false;
    let temp = {
        name: `Lesson ${x}`, status: stat
    };
    myWork.push(temp);
}
console.log(myWork);
```

Практическое занятие 5.4

```
const myTable = [];
const rows = 4;
const cols = 7;
let counter = 0;
for (let y = 0; y < rows; y++) {
    let tempTable = [];
    for (let x = 0; x < cols; x++) {
        counter++;
        tempTable.push(counter);
    }
    myTable.push(tempTable);
}
console.table(myTable);
```



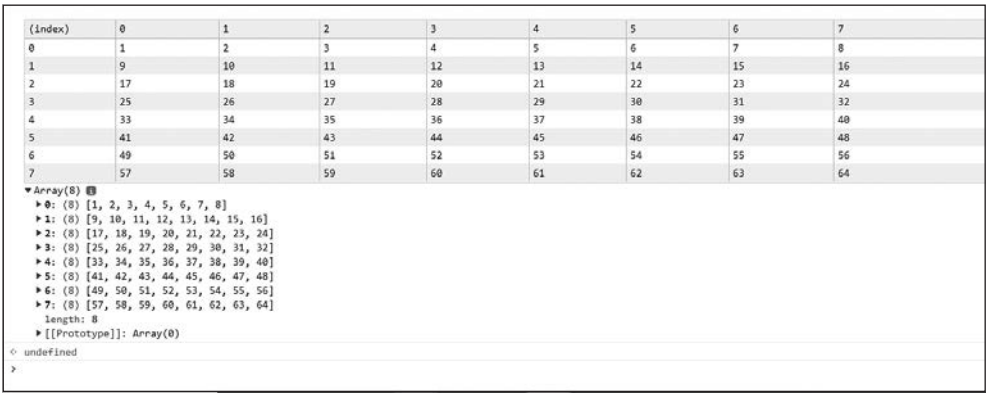
(index)	0	1	2	3	4	5	6
0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14
2	15	16	17	18	19	20	21
3	22	23	24	25	26	27	28

Практическое занятие 5.5

```
const grid = [];
const cells = 64;
let counter = 0;
let row;
for (let x = 0; x < cells + 1; x++) {
    if (counter % 8 == 0) {
        if (row != undefined) {
            grid.push(row);
        }
```

```
        row = [];
    }
    counter++;
    let temp = counter;
    row.push(temp);

}
console.table(grid);
```



(index)	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64

Практическое занятие 5.6

```
const myArray = [];
for (let x = 0; x < 10; x++) {
    myArray.push(x + 1);
}
console.log(myArray);

for (let i = 0; i < myArray.length; i++) {
    console.log(myArray[i]);
}
for (let val of myArray) {
    console.log(val);
}
```

Практическое занятие 5.7

```
const obj = {
    a: 1,
    b: 2,
    c: 3
};
for (let prop in obj) {
    console.log(prop, obj[prop]);
}
```

```
const arr = ["a", "b", "c"];
for (let w = 0; w < arr.length; w++) {
    console.log(w, arr[w]);
}

for (el in arr) {
    console.log(el, arr[el]);
}
```

Практическое занятие 5.8

```
let output = "";
let skipThis = 7;
for (let i = 0; i < 10; i++) {
  if (i === skipThis) {
    continue;
  }
  output += i;
}
console.log(output);
```

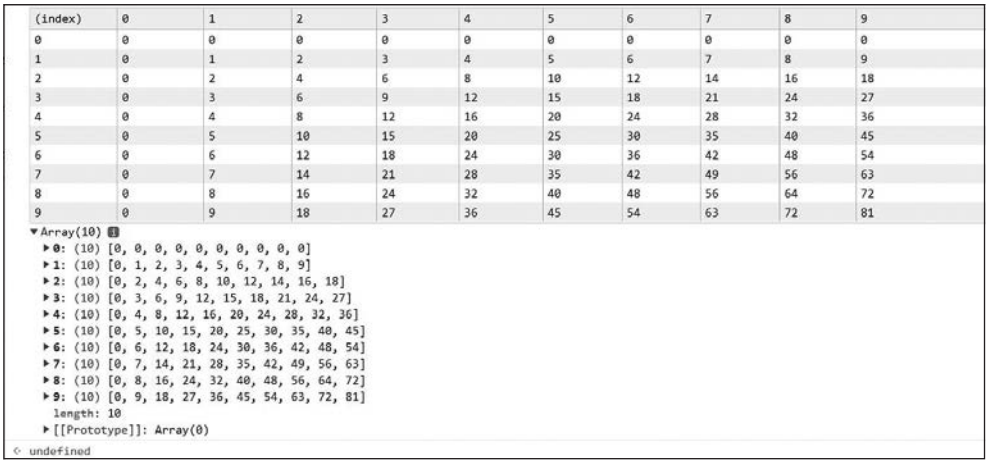
В качестве альтернативы можно использовать следующий код, заменив `continue` на `break`:

```
let output = "";
let skipThis = 7;
for (let i = 0; i < 10; i++) {
  if (i === skipThis) {
    break;
  }
  output += i;
}
console.log(output);
```

Проект

Математическая таблица умножения

```
const myTable = [];
const numm = 10;
for(let x=0; x<numm; x++){
    const temp = [];
    for(let y = 0; y<numm; y++){
        temp.push(x*y);
    }
    myTable.push(temp);
}
console.table(myTable);
```



(index)	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

```
▼Array(10) i
  ▶0: (10) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶1: (10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  ▶2: (10) [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
  ▶3: (10) [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
  ▶4: (10) [0, 4, 8, 12, 16, 20, 24, 28, 32, 36]
  ▶5: (10) [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
  ▶6: (10) [0, 6, 12, 18, 24, 30, 36, 42, 48, 54]
  ▶7: (10) [0, 7, 14, 21, 28, 35, 42, 49, 56, 63]
  ▶8: (10) [0, 8, 16, 24, 32, 40, 48, 56, 64, 72]
  ▶9: (10) [0, 9, 18, 27, 36, 45, 54, 63, 72, 81]
    length: 10
  ▶[[Prototype]]: Array(0)
◇ undefined
```

Вопросы для самопроверки

1. Результат:

```
0
3
6
9
```

2. Итог:

```
0
5
1
6
2
7
[1, 5, 7]
```

Глава 6. Функции

Практические занятия

Практическое занятие 6.1

```
function adder(a, b) {
    return a + b;
}
```

```
const val1 = 10;
const val2 = 20;
console.log(adder(val1, val2));
console.log(adder(20, 30));
```

Практическое занятие 6.2

```
const adj = ["super", "wonderful", "bad", "angry", "careful"];

function myFun() {
    const question = prompt("What is your name?");
    const nameAdj = Math.floor(Math.random() * adj.length);
    console.log(adj[nameAdj] + " " + question );
}
myFun();
```

Практическое занятие 6.3

```
const val1 = 10;
const val2 = 5;
let operat = "-";
function cal(a, b, op) {
  if (op == "-") {
    console.log(a – b);
  } else {
    console.log(a + b);
  }
}
cal(val1, val2, operat);
```

Практическое занятие 6.4

```
const myArr = [];

for(let x=0; x<10; x++){
  let val1 = 5 * x;
  let val2 = x * x;
  let res = cal(val1, val2, "+");
  myArr.push(res);
}
console.log(myArr);
function cal(a, b, op) {
  if (op == "-") {
    return a - b;
  } else {
    return a + b;
  }
}
```

Практическое занятие 6.5

```
let val = "1000";

(function () {
    let val = "100"; // переменная локальной области видимости
    console.log(val);
})();

let result = (function () {
    let val = "Laurence";
    return val;
})();
console.log(result);
console.log(val);

(function (val) {
    console.log(`My name is ${val}`);
})("Laurence");
```

Практическое занятие 6.6

```
function calcFactorial(nr) {
    console.log(nr);
    if (nr === 0) {
        return 1;
    }
    else {
        return nr * calcFactorial(--nr);
    }
}
console.log(calcFactorial(4));
```

Практическое занятие 6.7

```
let start = 10;
function loop1(val) {
    console.log(val);
    if (val < 1) {
        return;
    }
    return loop1(val - 1);
}
loop1(start);
function loop2(val) {
    console.log(val);
    if (val > 0) {
        val--;
```

```
        return loop2(val);
    }
    return;
}
loop2(start);
```

Практическое занятие 6.8

```
const test = function(val){
    console.log(val);
}
test('hello 1');

function test1(val){
    console.log(val);
}
test1("hello 2");
```

Проекты

Создание рекурсивной функции

```
const main = function counter(i) {
    console.log(i);
    if (i < 10) {
        return counter(i + 1);
    }
    return;
}
main(0);
```

Использование функции setTimeout()

```
const one = ()=> console.log('one');
const two = ()=> console.log('two');
const three = () =>{
    console.log('three');
    one();
    two();
}
const four = () =>{
    console.log('four');
    setTimeout(one,0);
    three();
}
four();
```

Вопросы для самопроверки

1. `10`
2. `Hello`
3. Ответ:

```
Welcome
Laurence
My Name is Laurence
```

4. `19`
5. `16`

Глава 7. Классы

Практические занятия

Практическое занятие 7.1

```
class Person {
    constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
    }
}
let person1 = new Person("Maaike", "van Putten");
let person2 = new Person("Laurence", "Svekis");
console.log("hello " + person1.firstname);
console.log("hello " + person2.firstname);
```

Практическое занятие 7.2

```
class Person {
    constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
    }
    fullname(){
        return this.firstname + " " + this.lastname;
    }
}
let person1 = new Person("Maaike", "van Putten");
let person2 = new Person("Laurence", "Svekis");
console.log(person1.fullname());
console.log(person2.fullname());
```

Практическое занятие 7.3

```
class Animal {
    constructor(species, sounds) {
        this.species = species;
        this.sounds = sounds;
    }
    speak() {
        console.log(this.species + " " + this.sounds);
    }
}

Animal.prototype.eat = function () {
    return this.species + " is eating";
}
let cat = new Animal("cat", "meow");
let dog = new Animal("dog", "bark");
cat.speak();
console.log(dog.eat());
console.log(dog);
```

Проекты

Приложение для контроля сотрудников

```
class Employee {
    constructor(first, last, years) {
        this.first = first;
        this.last = last;
        this.years = years;
    }
}
const person1 = new Employee("Laurence", "Svekis", 10);
const person2 = new Employee("Jane", "Doe", 5);
const workers = [person1, person2];

Employee.prototype.details = function(){
    return this.first + " " + this.last + " has worked here " +
           this.years + " years";
}

workers.forEach((person) => {
    console.log(person.details());
});
```

Расчет стоимости заказов

```
class Menu {
    #offer1 = 10;
    #offer2 = 20;
    constructor(val1, val2) {
        this.val1 = val1;
        this.val2 = val2;
    }
    calTotal(){
        return (this.val1 * this.#offer1) + (this.val2 * this.#offer2);
    }
    get total(){
        return this.calTotal();
    }
}
const val1 = new Menu(2,0);
const val2 = new Menu(1,3);
const val3 = new Menu(3,2);
console.log(val1.total);
console.log(val2.total);
console.log(val3.total);
```

Вопросы для самопроверки

1. `class`
2. Синтаксис будет следующим:

```
class Person {
    constructor(firstname, lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }
}
```

3. Наследование.
4. Ответы:
   - `True`;
   - `False`;
   - `True`;
   - `True`;
   - `False`.
5. 2.

Глава 8. Встроенные методы JavaScript

Практические занятия

Практическое занятие 8.1

```
const secretMes1 = "How's%20it%20going%3F";
const secretMes2 = "How's it going?";
const decodedComp = decodeURIComponent(secretMes1);
console.log(decodedComp);
const encodedComp = encodeURIComponent(secretMes2);
console.log(encodedComp);
const uri = "http://www.basescripts.com?=Hello World";
const encoded = encodeURI(uri);
console.log(encoded);
```

Практическое занятие 8.2

```
const arr = ["Laurence", "Mike", "Larry", "Kim", "Joanne", "Laurence",
"Mike", "Laurence", "Mike", "Laurence", "Mike"];
const arr2 = arr.filter ( (value, index, array) => {
    console.log(value,index,array.indexOf(value));
    return array.indexOf(value) === index;
});
console.log(arr2);
```

Практическое занятие 8.3

```
const myArr = [1,4,5,6];
const myArr1 = myArr.map(function(ele){
    return ele * 2;
});
console.log(myArr1);

const myArr2 = myArr.map((ele)=> ele*2);
console.log(myArr2);
```

Практическое занятие 8.4

```
const val = "thIs will be capiTalized for each word";
function wordsCaps(str) {
    str = str.toLowerCase();
    const tempArr = [];
    let words = str.split(" ");
```

```
    words.forEach(word => {
        let temp = word.slice(0, 1).toUpperCase() + word.slice(1);
        tempArr.push(temp);
    });
    return tempArr.join(" ");
}
console.log(wordsCaps(val));
```

Практическое занятие 8.5

```
let val = "I love JavaScript";
val = val.toLowerCase();
let vowels = ["a","e","i","o","u"];
vowels.forEach((letter,index) =>{
    console.log(letter);
    val = val.replaceAll(letter,index);
});
console.log(val);
```

Практическое занятие 8.6

```
console.log(Math.ceil(5.7));
console.log(Math.floor(5.7));
console.log(Math.round(5.7));
console.log(Math.random());
console.log(Math.floor(Math.random()*11)); // 0-10
console.log(Math.floor(Math.random()*10)+1); // 1-10;
console.log(Math.floor(Math.random()*100)+1); // 1-100;
function ranNum(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
for (let x = 0; x < 100; x++) {
    console.log(ranNum(1, 100));
}
```

Практическое занятие 8.7

```
let future = new Date(2025, 5, 15);
console.log(future);
const months = ["January", "February", "March", "April", "May", "June",
"July", "August", "September", "October", "November", "December"];
let day = future.getDate();
let month = future.getMonth();
let year = future.getFullYear();
let myDate = `${months[month-1]} ${day} ${year}`;
console.log(myDate);
```

Проекты

Скремблер слов

```
let str = "JavaScript";

function scramble(val) {
    let max = val.length;
    let temp = "";
    for(let i=0;i<max;i++){
        console.log(val.length);
        let index = Math.floor(Math.random() * val.length);
        temp += val[index];
        console.log(temp);
        val = val.substr(0, index) + val.substr(index + 1);
        console.log(val);
    }
    return temp;
}
console.log(scramble(str));
```

Таймер обратного отсчета

```
const endDate = "Sept 1 2022";

function countdown() {
    const total = Date.parse(endDate) - new Date();
    const days = Math.floor(total / (1000 * 60 * 60 * 24));
    const hrs = Math.floor((total / (1000 * 60 * 60)) % 24);
    const mins = Math.floor((total / 1000 / 60) % 60);
    const secs = Math.floor((total / 1000) % 60);
    return {
        days,
        hrs,
        mins,
        secs
    };
}

function update() {
    const temp = countdown();
    let output = "";
    for (const property in temp) {
        output += (`${property}: ${temp[property]} `);
    }
    console.log(output);
    setTimeout(update, 1000);
}

update();
```

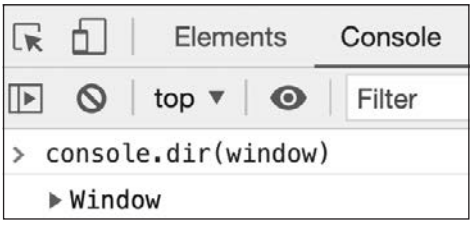
Вопросы для самопроверки

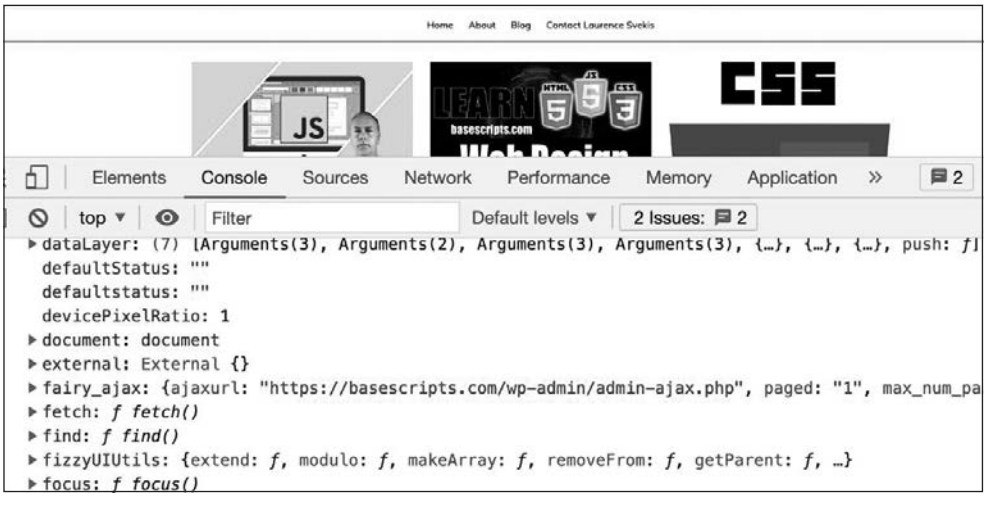
1. `decodeURIComponent(e)`
2. `4`
3. `["Hii", "hi", "hello", "Hii", "hi", "hi World", "Hi"]`
4. `["hi", "hi World"]`

Глава 9. Объектная модель документа

Практические занятия

Практическое занятие 9.1





Практическое занятие 9.2

```
console.log(window.location.protocol);
console.log(window.location.href);
```

Практическое занятие 9.3

```
<script>
    const output = document.querySelector('.output');
    output.textContent = "Hello World";
    output.classList.add("red");
    output.id = "tester";
    output.style.backgroundColor = "red";
    console.log(document.URL);
    output.textContent = document.URL;
</script>
```

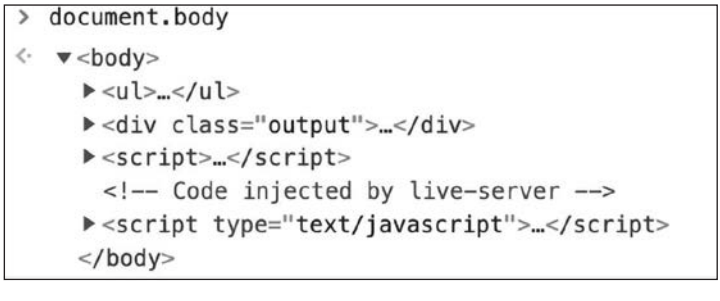
Проект

Управление элементами HTML с помощью JavaScript

```
const output = document.querySelector(".output");
const mainList = output.querySelector("ul");
mainList.id = "mainList";
console.log(mainList);
const eles = document.querySelectorAll("div");
for (let x = 0; x < eles.length; x++) {
    console.log(eles[x].tagName);
    eles[x].id = "id" + (x + 1);
    if (x % 2) {
        eles[x].style.color = "red";
    } else {
        eles[x].style.color = "blue";
    }
}
```

Вопросы для самопроверки

1. Объект, представляющий список элементов, содержащихся в `body` HTML-страницы.



2. `document.body.textContent = "Hello World";`
3. Код будет следующим:

```
for (const property in document) {
    console.log(`${property}: ${document[property]}`);
}
```

4. Код будет таким:

```
for (const property in window) {
    console.log(`${property}: ${document[window]}`);
}
```

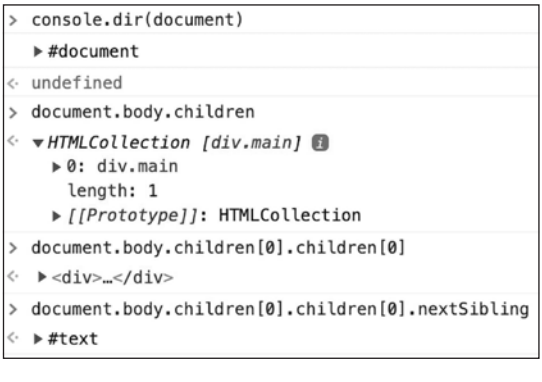
5. Код будет выглядеть следующим образом:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <h1>Test</h1>
    <script>
        const output = document.querySelector('h1');
        output.textContent = "Hello World";
    </script>
</body>
</html>
```

Глава 10. Управление динамическими элементами с помощью DOM

Практические занятия

Практическое занятие 10.1



Практическое занятие 10.2

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
</head>
<body>
    <div id="one">Hello World</div>
    <script>
        const myEle = document.getElementById("one");
        console.log(myEle);
    </script>
</body>
</html>
```

Практическое занятие 10.3

```
<!doctype html>
<html>
<head>
    <title>Dynamic event manipulation</title>
</head>
<body>
    <div>Hello World 1</div>
    <div>Hello World 2</div>
    <div>Hello World 3</div>
    <script>
        const myEles = document.getElementsByTagName("div");
        console.log(myEles[1]);
    </script>
</body>
</html>
```

Практическое занятие 10.4

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
</head>
<body>
    <h1 class="ele">Hello World</h1>
    <div class="ele">Hello World 1</div>
    <div class="ele">Hello World 3</div>
    <script>
        const myEles = document.getElementsByClassName("ele");
```

```
        console.log(myEles[0]);
    </script>
</body>
</html>
```

Практическое занятие 10.5

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
</head>
<body>
    <h1 class="ele">Hello World</h1>
    <div class="ele">Hello World 1</div>
    <div class="ele">Hello World 3</div>
    <p class="ele">Hello World 4</p>
    <script>
        const myEle = document.querySelector(".ele");
        console.log(myEle);
    </script>
<body>
</html>
```

Практическое занятие 10.6

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <div class="container">
        <div class="myEle">One</div>
        <div class="myEle">Two</div>
        <div class="myEle">Three</div>
        <div class="myEle">Four</div>
        <div class="myEle">Five</div>
    </div>
    <script>
        const eles = document.querySelectorAll(".myEle");
        console.log(eles);
        eles.forEach((el) => {
            console.log(el);
        });
    </script>
</body>
</html>
```

Практическое занятие 10.7

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <div>
        <button onclick="message(this)">Button 1</button>
        <button onclick="message(this)">Button 2</button>
    </div>
    <script>
        function message(el) {
            console.dir(el.textContent);
        }
    </script>
</body>
</html>
```

Практическое занятие 10.8

```
<script>
    const message = document.querySelector("#message");
    const myArray = ["Laurence", "Mike", "John", "Larry", "Kim",
                     "Joanne", "Lisa", "Janet", "Jane"];
    build();
    // addClicks();
    function build() {
        let html = "<h1>My Friends Table</h1><table>";
        myArray.forEach((item, index) => {
            html += `<tr class="box" data-row="${index+1}"
                    data-name="${item}" onclick="getData(this)">
                    <td>${item}</td>`;
            html += `<td >${index + 1}</td></tr>`;
        });
        html += "</table>";
        document.getElementById("output").innerHTML = html;
    }

    function getData(el) {
        let temp = el.getAttribute("data-row");
        let tempName = el.getAttribute("data-name");
        message.innerHTML = `${tempName } is in row #${temp}`;
    }
</script>
```

Практическое занятие 10.9

```
<script>
    const btns = document.querySelectorAll("button");
    btns.forEach((btn)=>{
        function output(){
            console.log(this.textContent);
        }
        btn.addEventListener("click",output);
    });
</script>
```

Практическое занятие 10.10

```
<script>
    document.getElementById("addNew").onclick = function () {
        addOne();
    }
    function addOne() {
        var a = document.getElementById("addItem").value;
        var li = document.createElement("li");
        li.appendChild(document.createTextNode(a));
        document.getElementById("sList").appendChild(li);
    }
</script>
```

Проекты

Сворачиваемый компонент-аккордеон

```
<script>
    const menus = document.querySelectorAll(".title");
    const openText = document.querySelectorAll(".myText");
    menus.forEach((el) => {
        el.addEventListener("click", (e) => {
            console.log(el.nextElementSibling);
            remover();
            el.nextElementSibling.classList.toggle("active");
        })
    })
    function remover() {
        openText.forEach((ele) => {
            ele.classList.remove("active");
        })
    }
</script>
```

Интерактивная система голосования

```
<script>
    window.onload = build;
    const myArray = ["Laurence", "Mike", "John", "Larry"];
    const message = document.getElementById("message");
    const addNew = document.getElementById("addNew");
    const newInput = document.getElementById("addFriend");
    const output = document.getElementById("output");
    addNew.onclick = function () {
        const newFriend = newInput.value;
        adder(newFriend, myArray.length, 0);
        myArray.push(newFriend);
    }
    function build() {
        myArray.forEach((item, index) => {
            adder(item, index, 0);
        });
    }
    function adder(name, index, counter) {
        const tr = document.createElement("tr");
        const td1 = document.createElement("td");
        td1.classList.add("box");
        td1.textContent = index + 1;
        const td2 = document.createElement("td");
        td2.textContent = name;
        const td3 = document.createElement("td");
        td3.textContent = counter;
        tr.append(td1);
        tr.append(td2);
        tr.append(td3);
        tr.onclick= function () {
            console.log(tr.lastChild);
            let val = Number(tr.lastChild.textContent);
            val++;
            tr.lastChild.textContent = val;
        }
        output.appendChild(tr);
    }
</script>
```

Игра «Виселица»

```
<script>
    const game = { cur: "", solution: "", puzz: [], total: 0 };
    const myWords = ["learn Javascript", "learn html",
                     "learn css"];
    const score = document.querySelector(".score");
    const puzzle = document.querySelector(".puzzle");
    const letters = document.querySelector(".letters");
```

```
const btn = document.querySelector("button");
btn.addEventListener("click", startGame);
function startGame() {
    if (myWords.length > 0) {
        btn.style.display = "none";
        game.puzz = [];
        game.total = 0;
        game.cur = myWords.shift();
        game.solution = game.cur.split("");
        builder();
    } else {
        score.textContent = "No More Words.";
    }
}
function createElements(elType, parentEle, output, cla) {
    const temp = document.createElement(elType);
    temp.classList.add("boxE");
    parentEle.append(temp);
    temp.textContent = output;
    return temp;
}
function updateScore() {
    score.textContent = `Total Letters Left : ${game.total}`;
    if (game.total <= 0) {
        console.log("game over");
        score.textContent = "Game Over";
        btn.style.display = "block";
    }
}
function builder() {
    letters.innerHTML = "";
    puzzle.innerHTML = "";
    game.solution.forEach((lett) => {
        let div = createElements("div", puzzle, "-", "boxE");
        if (lett == " ") {
            div.style.borderColor = "white";
            div.textContent = " ";
        } else {
            game.total++;
        }
        game.puzz.push(div);
        updateScore();
    })
    for (let i = 0; i < 26; i++) {
        let temp = String.fromCharCode(65 + i);
        let div = createElements("div", letters, temp,"box");
        let checker = function (e) {
            div.style.backgroundColor = "#ddd";
            div.classList.remove("box");
            div.classList.add("boxD");
            div.removeEventListener("click", checker);
```

```
                checkLetter(temp);
            }
            div.addEventListener("click", checker);
        }
    }
    function checkLetter(letter) {
        console.log(letter);
        game.solution.forEach((ele, index) => {
            if (ele.toUpperCase() == letter) {
                game.puzz[index].textContent = letter;
                game.total--;
                updateScore();
            };
        };
        )
    }
</script>
```

Вопросы для самопроверки

1. `Hello <br> World`
2. `Hello`
   `World`
3. `Hello World`
4. При нажатии на `three` будет выведено `three`. При нажатии на `one` будет выведено:

   ```
   one
   two
   three
   ```

5. `btn.removeEventListener("click", myFun);`

Глава 11. Интерактивный контент и прослушиватели событий

Практические занятия

Практическое занятие 11.1

```
<!DOCTYPE html>
<html>

<head>
    <title>Laurence Svekis</title>
```

```
</head>

<body>
    <script>
        let darkMode = false;
        window.onclick = () => {
            console.log(darkMode);
            if (!darkMode) {
                document.body.style.backgroundColor = "black";
                document.body.style.color = "white";
                darkMode = true;
            } else {
                document.body.style.backgroundColor = "white";
                document.body.style.color = "black";
                darkMode = false;
            }
        }
    </script>
</body>

</html>
```

Практическое занятие 11.2

```
<!doctype html>
<html>
<body>
    <div>red</div>
    <div>blue</div>
    <div>green</div>
    <div>yellow</div>
    <script>
        const divs = document.querySelectorAll("div");
        divs.forEach((el)=>{
            el.addEventListener("click",()=>{
                document.body.style.backgroundColor = el.textContent;
            });
        })
    </script>
</body>
</html>
```

Практическое занятие 11.3

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
```

```
    <script>
        document.addEventListener("DOMContentLoaded", (e) => {
            message("Document ready", e);
        });
        window.onload = (e) => {
            message("Window ready", e);
        }
        function message(val, event) {
            console.log(event);
            console.log(val);
        }
    </script>
</body>
</html>
```

Практическое занятие 11.4

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <div class="output"></div>
    <script>
        const output = document.querySelector(".output");
        output.textContent = "hello world";
        output.style.height = "200px";
        output.style.width = "400px";
        output.style.backgroundColor = "red";
        output.addEventListener("mousedown", function (e) {
            message("green", e);
        });
        output.addEventListener("mouseover", function (e) {
            message("red", e);
        });
        output.addEventListener("mouseout", function (e) {
            message("yellow", e);
        });
        output.addEventListener("mouseup", function (e) {
            message("blue", e);
        });
        function message(elColor, event) {
            console.log(event.type);
            output.style.backgroundColor = elColor;
        }
    </script>
</body>
</html>
```

Практическое занятие 11.5

```
<script>
    const myInput = document.querySelector("input[name='message']");
    const output = document.querySelector(".output");
    const btn1 = document.querySelector(".btn1");
    const btn2 = document.querySelector(".btn2");
    const btn3 = document.querySelector(".btn3");
    const log = [];
    btn1.addEventListener("click", tracker);
    btn2.addEventListener("click", tracker);
    btn3.addEventListener("click", (e) => {
        console.log(log);
    });
    function tracker(e) {
        output.textContent = myInput.value;
        const ev = e.target;
        console.dir(ev);
        const temp = {
            message: myInput.value,
            type: ev.type,
            class: ev.className,
            tag: ev.tagName
        };
        log.push(temp);
        myInput.value = "";
    }
</script>
```

Практическое занятие 11.6

```
<script>
    const main = document.querySelector(".container");
    const boxes = document.querySelectorAll(".box");
    main.addEventListener("click", (e) => {
        console.log("4");
    },false);
    main.addEventListener("click", (e) => {
        console.log("1");
    },true);

    boxes.forEach(ele => {
        ele.addEventListener("click", (e) => {
            console.log("3");
            console.log(e.target.textContent);
        },false);
        ele.addEventListener("click", (e) => {
            console.log("2");
```

```
            console.log(e.target.textContent);
        },true);

    });
</script>
```

Практическое занятие 11.7

```
<script>
    const output = document.querySelector(".output1");

    const in1 = document.querySelector("input[name='first']");
    const in2 = document.querySelector("input[name='last']");
    in1.addEventListener("change", (e) => {
        console.log("change");
        updater(in1.value);
    });
    in1.addEventListener("blur", (e) => {
        console.log("blur");
    });
    in1.addEventListener("focus", (e) => {
        console.log("focus");
    });
    in2.addEventListener("change", (e) => {
        console.log("change");
        updater(in2.value);
    });
    in2.addEventListener("blur", (e) => {
        console.log("blur");
    });
    in2.addEventListener("focus", (e) => {
        console.log("focus");
    });
    function updater(str) {
        output.textContent = str;
    }
</script>
```

Практическое занятие 11.8

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <div class="output"></div>
        <input type="text" name="myNum1">
        <input type="text" name="myNum2">
    <script>
```

```
        const eles = document.querySelectorAll("input");
        const output = document.querySelector(".output");
        eles.forEach(el => {
            el.addEventListener("keydown", (e) => {
                if (!isNaN(e.key)) {
                    output.textContent += e.key;
                }
            });
            el.addEventListener("keyup", (e) => {
                console.log(e.key);
            });
            el.addEventListener("paste", (e) => {
                console.log('pasted');
            });
        });
    </script>
</body>
</html>
```

Практическое занятие 11.9

```
    <script>
        const dragme = document.querySelector("#dragme");
        dragme.addEventListener("dragstart", (e) => {
            dragme.style.opacity = .5;
        });
        dragme.addEventListener("dragend", (e) => {
            dragme.style.opacity = "";
        });
        const boxes = document.querySelectorAll(".box");
        boxes.forEach(box => {
            box.addEventListener("dragenter", (e) => {
                e.target.classList.add('red');
            });
            box.addEventListener("dragover", (e) => {
                e.preventDefault();
            });
            box.addEventListener("dragleave", (e) => {
                // console.log("leave");
                e.target.classList.remove('red');
            });
            box.addEventListener("drop", (e) => {
                e.preventDefault();
                console.log("dropped");
                e.target.appendChild(dragme);
            });
        });
        function dragStart(e) {
            console.log("Started");
        }
    </script>
```

Практическое занятие 11.10

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>

<body>
    <form action="index2.html" method="get">
        First: <input type="text" name="first">
        <br>Last: <input type="text" name="last">
        <br>Age: <input type="number" name="age">
        <br><input type="submit" value="submit">
    </form>

    <script>
        const form = document.querySelector("form");
        const email = document.querySelector("#email");
        form.addEventListener("submit", (e) => {
            let error = false;
            if (checker(form.first.value)) {
                console.log("First Name needed");
                error = true;
            }
            if (checker(form.last.value)) {
                console.log("Last Name needed");
                error = true;
            }
            if (form.age.value < 19) {
                console.log("You must be 19 or over");
                error = true;
            }
            if (error) {
                e.preventDefault();
                console.log("please review the form");
            }
        });
        function checker(val) {
            console.log(val.length);
            if (val.length < 6) {
                return true;
            }
            return false;
        }
    </script>
</body>
</html>
```

Практическое занятие 11.11

```
<!doctype html>
<html>
<style>
    div {
        background-color: purple;
        width: 100px;
        height: 100px;
        position: absolute;
    }
</style>
<body>
    <div id="block"></div>
    <script>
        const main = document.querySelector("#block");
        let mover = { speed: 10, dir: 1, pos: 0 };
        main.addEventListener("click", moveBlock);
        function moveBlock() {
            let x = 30;
            setInterval(function () {
                if (x < 1) {
                    clearInterval();
                } else {
                    if (mover.pos > 800 || mover.pos < 0) {
                        mover.dir *= -1;
                    }
                    x--;
                    mover.pos += x * mover.dir;
                    main.style.left = mover.pos + "px";
                    console.log(mover.pos);
                }
            }, 2);
        }
    </script>
</body>
</html>
```

Проекты

Ваша собственная аналитика

```
<!doctype html >
<html>
<head>
    <title>JS Tester</title>
    <style>.box{width:200px;height:100px;border:1px solid black}</style>
```

```
</head>
<body>
    <div class="container">
        <div class="box" id="box0">Box #1</div>
        <div class="box" id="box1">Box #2</div>
        <div class="box" id="box2">Box #3</div>
        <div class="box" id="box3">Box #4</div>
    </div>
    <script>
        const counter = [];
        const main = document.querySelector(".container");
        main.addEventListener("click",tracker);
        function tracker(e){
            const el = e.target;
            if(el.id){
            const temp = {};
            temp.content = el.textContent;
            temp.id = el.id;
            temp.tagName = el.tagName;
            temp.class = el.className;
            console.dir(el);
            counter.push(temp);
            console.log(counter);
            }
        }
    </script>
</body>
</html>
```

Звездная рейтинговая система

```
    <script>
        const starsUL = document.querySelector(".stars");
        const output = document.querySelector(".output");
        const stars = document.querySelectorAll(".star");
        stars.forEach((star, index) => {
            star.starValue = (index + 1);
            star.addEventListener("click", starRate);
        });
        function starRate(e) {
            output.innerHTML =
                `You Rated this ${e.target.starValue} stars`;
            stars.forEach((star, index) => {
                if (index < e.target.starValue) {
                    star.classList.add("orange");
                } else {
                    star.classList.remove("orange");
                }
            });
        }
    </script>
```

Отслеживание позиции мыши

```
<!DOCTYPE html>
<html>
<head>
    <title>Complete JavaScript Course</title>
    <style>
        .holder {
            display: inline-block;
            width: 300px;
            height: 300px;
            border: 1px solid black;
            padding: 10px;
        }

        .active {
            background-color: red;
        }
    </style>
</head>
<body>
    <div class="holder">
        <div id="output"></div>
    </div>
    <script>
        const ele = document.querySelector(".holder");
        ele.addEventListener("mouseover",
            (e) => { e.target.classList.add("active"); });
        ele.addEventListener("mouseout",
            (e) => { e.target.classList.remove("active"); });
        ele.addEventListener("mousemove", coordin);
        function coordin() {
            let html = "X:" + event.clientX + " | Y:" + event.clientY;
            document.getElementById("output").innerHTML = html;
        }
    </script>
</body>
</html>
```

Игра на скорость со щелчками кнопкой мыши

```
    <script>
        const output = document.querySelector('.output');
        const message = document.querySelector('.message');
        message.textContent = "Press to Start";
        const box = document.createElement('div');
        const game = {
            timer: 0,
            start: null
        };
```

```
        box.classList.add('box');
        output.append(box);

        box.addEventListener('click', (e) => {
            box.textContent = "";
            box.style.display = 'none';
            game.timer = setTimeout(addBox, ranNum(3000));
            if (!game.start) {
                message.textContent = 'Loading....';
            } else {
                const cur = new Date().getTime();
                const dur = (cur - game.start) / 1000;
                message.textContent = `It took ${dur} seconds to click`;
            }
        });

        function addBox() {
            message.textContent = 'Click it...';
            game.start = new Date().getTime();
            box.style.display = 'block';
            box.style.left = ranNum(450) + 'px';
            box.style.top = ranNum(450) + 'px';
        }

        function ranNum(max) {
            return Math.floor(Math.random() * max);
        }
    </script>
```

Вопросы для самопроверки

1. В модели объекта `window`.
2. Метод `preventDefault()` отменяет событие, если оно может быть отменено. Действие по умолчанию, относящееся к событию, не будет выполнено.

Глава 12. Средний уровень JavaScript

Практические занятия

Практическое занятие 12.1

```
    <script>
        const output = document.getElementById("output");
        const findValue = document.getElementById("sText");
        const replaceValue = document.getElementById("rText");
        document.querySelector("button").addEventListener("click", lookUp);
```

```
        function lookUp() {
            const s = output.textContent;
            const rt = replaceValue.value;
            const re = new RegExp(findValue.value, "gi");
            if (s.match(re)) {
                let newValue = s.replace(re, rt);
                output.textContent = newValue;
            }
        }
    </script>
```

Практическое занятие 12.2

```
    <script>
        const output = document.querySelector(".output");
        const emailVal = document.querySelector("input");
        const btn = document.querySelector("button");
        const emailExp =
            /([A-Za-z0-9._-]+@[A-Za-z0-9._-]+\.[A-Za-z0-9]+)\w+/;
        btn.addEventListener("click", (e) => {
            const val = emailVal.value;
            const result = emailExp.test(val);
            let response = "";
            if (!result) {
                response = "Invalid Email";
                output.style.color = "red";
            } else {
                response = "Valid Email";
                output.style.color = "green";
            }
            emailVal.value = "";
            output.textContent = response;
        });
    </script>
```

Практическое занятие 12.3

```
function showNames() {
    let lastOne = "";
    for (let i = 0; i < arguments.length; i++) {
        lastOne = arguments[i];
    }
    return lastOne;
}
console.log(showNames("JavaScript", "Laurence", "Mike", "Larry"));
```

Практическое занятие 12.4

```
15
45
```

Практическое занятие 12.5

```
function test(val) {
    try {
        if (isNaN(val)) {
            throw "Not a number";
        } else {
            console.log("Got number");
        }
    } catch (e) {
        console.error(e);
    } finally {
        console.log("Done " + val);
    }
}
test("a");
test(100);
```

Практическое занятие 12.6

```
    <script>
        console.log(document.cookie);
        console.log(rCookie("test1"));
        console.log(rCookie("test"));
        cCookie("test1", "new Cookie", 30);
        dCookie("test2");
        function cCookie(cName, value, days) {
            if (days) {
                const d = new Date();
                d.setTime(d.getTime() + (days * 24 * 60 * 60 * 1000));
                let e = "; expires=" + d.toUTCString();
                document.cookie = cName + "=" + value + e + "; path=/";
            }
        }
        function rCookie(cName) {
            let cookieValue = false;
            let arr = document.cookie.split("; ");
            arr.forEach(str => {
                const cookie = str.split("=");
                if (cookie[0] == cName) {
                    cookieValue = cookie[1];
                }
            });
            return cookieValue;
        }
        function dCookie(cName) {
            cCookie(cName, "", -1);
        }
    </script>
```

Практическое занятие 12.7

```
<script>
    const userTask = document.querySelector(".main input");
    const addBtn = document.querySelector(".main button");
    const output = document.querySelector(".output");
    const tasks = JSON.parse(localStorage.getItem("tasklist")) || [];
    addBtn.addEventListener("click", createListItem);
    if (tasks.length > 0) {
        tasks.forEach((task) => {
            genItem(task.val, task.checked);
        });
    }
    function saveTasks() {
        localStorage.setItem("tasklist", JSON.stringify(tasks));
    }
    function buildTasks() {
        tasks.length = 0;
        const curList = output.querySelectorAll("li");
        curList.forEach((el) => {
            const tempTask = {
                val: el.textContent,
                checked: false
            };
            if (el.classList.contains("ready")) {
                tempTask.checked = true;
            }
            tasks.push(tempTask);
        });
    saveTasks();
}
function genItem(val, complete) {
    const li = document.createElement("li");
    const temp = document.createTextNode(val);
    li.appendChild(temp);
    output.append(li);
    userTask.value = "";
    if (complete) {
        li.classList.add("ready");
    }
        li.addEventListener("click", (e) => {
            li.classList.toggle("ready");
            buildTasks();
        });
        return val;
    }
    function createListItem() {
        const val = userTask.value;
        if (val.length > 0) {
```

```
                const myObj = {
                    val: genItem(val, false),
                    checked: false
                };
                tasks.push(myObj);
                saveTasks();
            }
        }
    </script>
```

Практическое занятие 12.8

```
let myList = [{
        "name": "Learn JavaScript",
        "status": true
    },
    {
        "name": "Try JSON",
        "status": false
    }
];

reloader();
function reloader() {
    myList.forEach((el) => {
    console.log(`${el.name} = ${el.status}`);
    });
}
```

Практическое занятие 12.9

```
let myList = [{
        "name": "Learn JavaScript",
        "status": true
    },
    {
        "name": "Try JSON",
        "status": false
    }
];

const newStr = JSON.stringify(myList);
const newObj = JSON.parse(newStr);
newObj.forEach((el)=>{
    console.log(el);
});
```

Проекты

Сборщик адресов электронной почты

```
<script>
    const firstArea = document.querySelector(
        "textarea[name='txtarea']");
    const secArea = document.querySelector(
        "textarea[name='txtarea2']");
    document.querySelector("button").addEventListener("click", lookUp);
    function lookUp() {
        const rawTxt = firstArea.value;
        const eData = rawTxt.match(
            /([a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)/gi);
        const holder = [];
        for (let x = 0; x < eData.length; x++) {
            if (holder.indexOf(eData[x]) == -1) {
                holder.push(eData[x]);
            }
        }
        secArea.value = holder.join(',');
    }
</script>
```

Валидатор форм

```
<script>
    const myForm = document.querySelector("form");
    const inputs = document.querySelectorAll("input");
    const errors = document.querySelectorAll(".error");
    const required = ["email", "userName"];
    myForm.addEventListener("submit", validation);
    function validation(e) {
        let data = {};
        e.preventDefault();
        errors.forEach(function (item) {
            item.classList.add("hide");
        });
        let error = false;
        inputs.forEach(function (el) {
            let tempName = el.getAttribute("name");
            if (tempName != null) {
                el.style.borderColor = "#ddd";
                if (el.value.length == 0 &&
                required.includes(tempName)) {
                    addError(el, "Required Field", tempName);
```

```
                    error = true;
                }
                if (tempName == "email") {
                    let exp = /([A-Za-z0-9._-]+@[A-Za-z0-9._-]+\.
                              [A-Za-z0-9]+)\w+/;
                    let result = exp.test(el.value);
                    if (!result) {
                        addError(el, "Invalid Email", tempName);
                        error = true;
                    }
                }
                if (tempName == "password") {
                    let exp = /[A-Za-z0-9]+$/;
                    let result = exp.test(el.value);
                    if (!result) {
                        addError(el, "Only numbers and Letters",
                                 tempName);
                        error = true;
                    }
                    if (!(el.value.length > 3 &&
                    el.value.length < 9)) {
                        addError(el, "Needs to be between 3-8 " +
                                 "characters", tempName);
                        error = true;
                    }
                }
                data[tempName] = el.value;
            }
        });
        if (!error) {
            myForm.submit();
        }
    }

    function addError(el, mes, fieldName) {
        let temp = el.nextElementSibling;
        temp.classList.remove("hide");
        temp.textContent = fieldName.toUpperCase() + " " + mes;
        el.style.borderColor = "red";
        el.focus();
    }
</script>
```

Простой математический вопросник

```
<!doctype html>
<html>
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
    <span class="val1"></span> <span>+</span>
```

```
    <span class="val2"></span> = <span>
        <input type="text" name="answer"></span><button>Check</button>
    <div class="output"></div>
    <script>
        const app = function () {
            const game = {};
            const val1 = document.querySelector(".val1");
            const val2 = document.querySelector(".val2");
            const output = document.querySelector(".output");
            const answer = document.querySelector("input");
            function init() {
                document.querySelector("button").addEventListener(
                    "click", checker);
                loadQuestion();
            }
            function ranValue(min, max) {
                return Math.floor(Math.random() * (max - min + 1) +
                                  min);
            }
            function loadQuestion() {
                game.val1 = ranValue(1, 100);
                game.val2 = ranValue(1, 100);
                game.answer = game.val1 + game.val2;
                val1.textContent = game.val1;
                val2.textContent = game.val2;
            }
            function checker() {
                let bg = answer.value == game.answer ? "green" : "red";
                output.innerHTML +=
                    `<div style="color:${bg}">${game.val1} +
                    ${game.val2} = ${game.answer} (${answer.value})
                    </div>`;
                answer.value = "";
                loadQuestion();
            }
            return {
                init: init
            };
        }();
        document.addEventListener('DOMContentLoaded', app.init);
    </script>
</body>
</html>
```

Вопросы для самопроверки

1. Диапазон совпадений расположен между `a` и `e` и чувствителен к регистру. Он вернет остальную часть слова: `enjoy avaScript`.
2. Да.
3. Это удалит файлы cookie с сайта.

4. `hello world`
5. `a is not defined`
6. `a`
   `c`
   `b`

Глава 13. Параллелизм

Практические занятия

Практическое занятие 13.1

```
function greet(fullName){
    console.log(`Welcome, ${fullName[0]} ${fullName[1]}`)
}
function processCall(user, callback){
    const fullName = user.split(" ");
    callback(fullName);
}

processCall("Laurence Svekis", greet);
```

Практическое занятие 13.2

```
const myPromise = new Promise((resolve, reject) => {
    resolve("Start Counting");
});

function counter(val){
    console.log(val);
}

myPromise
  .then(value => {counter(value); return "one"})
  .then(value => {counter(value); return "two"})
  .then(value => {counter(value); return "three"})
  .then(value => {counter(value);});
```

Практическое занятие 13.3

```
    let cnt = 0;
    function outputTime(val) {
        return new Promise(resolve => {
            setTimeout(() => {
                cnt++;
                resolve(`x value ${val} counter:${cnt}`);
```

```
            }, 1000);
        });
    }
    async function aCall(val) {
        console.log(`ready ${val} counter:${cnt}`);
        const res = await outputTime(val);
        console.log(res);
    }
    for (let x = 1; x < 4; x++) {
        aCall(x);
    }
```

Проект

Проверка паролей

```
const allowed = ["1234", "pass", "apple"];

function passwordChecker(pass) {
    return allowed.includes(pass);
}

function login(password) {
    return new Promise((resolve, reject) => {
        if (passwordChecker(password)) {
            resolve({
                status: true
            });
        } else {
            reject({
                status: false
            });
        }
    });
}

function checker(pass) {
    login(pass)
        .then(token => {
            console.log("Approve:");
            console.log(token);
        })
        .catch(value => {
            console.log("Reject:");
            console.log(value);
        })
}
checker("1234");
checker("wrong");
```

Вопросы для самопроверки

1. Обновленный код будет выглядеть так:

```
function addOne(val){
  return val + 1;
}
function total(a, b, callback){
  const sum = a + b;
  return callback(sum);
}
console.log(total(4, 5, addOne));
```

2. На экране появится сообщение об ошибке: `Error: Oh no`.
3. Обновленный код:

```
function checker(val) {
  return new Promise((resolve, reject) => {
    if (val > 5) {
      resolve("Ready");
    } else {
      reject(new Error("Oh no"));
    }
  });
}
checker(5)
.then((data) => {console.log(data); })
.catch((err) => {console.error(err); })
.finally(() => { console.log("done");});
```

4. Новый код будет выглядеть следующим образом:

```
async function myFun() {
  return "Hello";
}
myFun().then(
  function(val) { console.log(val); },
  function(err) { conole.log(err); }
```

Глава 14. HTML5, Canvas и JavaScript

Практические занятия

Практическое занятие 14.1

```
<script>
    const message = document.getElementById("message");
    const output = document.querySelector(".output");
    const myInput = document.querySelector("input");
```

```
        myInput.addEventListener("change", uploadAndReadFile);
        function uploadAndReadFile(e) {
            const files = e.target.files;
            for (let i = 0; i < files.length; i++) {
                const file = files[i];
                const img = document.createElement("img");
                img.classList.add("thumb");
                img.file = file;
                output.appendChild(img);
                const reader = new FileReader();
                reader.onload = (function (myImg) {
                    return function (e) {
                        myImg.src = e.target.result;
                    };
                })(img);
                reader.readAsDataURL(file);
            }
        }
    </script>
```

Практическое занятие 14.2

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
    <style>
        #canvas {
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <canvas id="canvas" width="640" height="640">Not Supported</canvas>
    <script>
        const canvas = document.querySelector('#canvas');
        const ctx = canvas.getContext("2d");
        ctx.fillStyle = "red";
        ctx.fillRect(100, 100, 500, 300); // закрашенная фигура
        ctx.strokeRect(90, 90, 520, 320); // контур
        ctx.clearRect(150, 150, 400, 200); // прозрачность
    </script>
</body>
</html>
```

Практическое занятие 14.3

```
<!doctype html>
<html>
<head>
```

```
    <title>Canvas HTML5</title>
    <style>
        #canvas {
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <canvas id="canvas" width="640" height="640">Not Supported</canvas>
    <script>
        const canvas = document.querySelector("#canvas");
        const ctx = canvas.getContext("2d");
        ctx.beginPath();
        ctx.fillStyle = "red";
        ctx.arc(300, 130, 100, 0, Math.PI * 2);
        ctx.fill();
        ctx.beginPath();
        ctx.fillStyle = "black";
        ctx.arc(250, 120, 20, 0, Math.PI * 2);
        ctx.moveTo(370, 120);
        ctx.arc(350, 120, 20, 0, Math.PI * 2);
        ctx.moveTo(240, 160);
        ctx.arc(300, 160, 60, 0, Math.PI);
        ctx.fill();
        ctx.moveTo(300, 130);
        ctx.lineTo(300, 150);
        ctx.stroke();
        ctx.beginPath();
        ctx.moveTo(300, 230);
        ctx.lineTo(300, 270);
        ctx.lineTo(400, 270);
        ctx.lineTo(200, 270);
        ctx.lineTo(300, 270);
        ctx.lineTo(300, 350);
        ctx.lineTo(400, 500);
        ctx.moveTo(300, 350);
        ctx.lineTo(200, 500);
        ctx.stroke();
        ctx.beginPath();
        ctx.fillStyle = "blue";
        ctx.moveTo(200, 50);
        ctx.lineTo(400, 50);
        ctx.lineTo(300, 20);
        ctx.lineTo(200, 50);
        ctx.fill();
        ctx.stroke();
    </script>
</body>
</html>
```

Практическое занятие 14.4

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
    <style>
        #canvas {
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <canvas id="canvas" width="640" height="640">Not Supported</canvas>
    <script>
        const canvas = document.querySelector("#canvas");
        const ctx = canvas.getContext("2d");
        ctx.beginPath();
        ctx.fillStyle = "red";
        ctx.arc(300, 130, 100, 0, Math.PI * 2);
        ctx.fill();
        ctx.beginPath();
        ctx.fillStyle = "black";
        ctx.arc(250, 120, 20, 0, Math.PI * 2);
        ctx.moveTo(370, 120);
        ctx.arc(350, 120, 20, 0, Math.PI * 2);
        ctx.moveTo(240, 160);
        ctx.arc(300, 160, 60, 0, Math.PI);
        ctx.fill();
        ctx.moveTo(300, 130);
        ctx.lineTo(300, 150);
        ctx.stroke();
        ctx.beginPath();
        ctx.moveTo(300, 230);
        ctx.lineTo(300, 270);
        ctx.lineTo(400, 270);
        ctx.lineTo(200, 270);
        ctx.lineTo(300, 270);
        ctx.lineTo(300, 350);
        ctx.lineTo(400, 500);
        ctx.moveTo(300, 350);
        ctx.lineTo(200, 500);
        ctx.stroke();
        ctx.beginPath();
        ctx.fillStyle = "blue";
        ctx.moveTo(200, 50);
        ctx.lineTo(400, 50);
        ctx.lineTo(300, 20);
```

```
        ctx.lineTo(200, 50);
        ctx.fill();
        ctx.stroke();
    </script>
</body>
</html>
```

Практическое занятие 14.5

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
    <style>
        #canvas {
            border: 1px solid black;
        }
    </style>
</head>
<body>
    <div><label>Image</label>
        <input type="file" id="imgLoader" name="imgLoader">
    </div>
    <div><canvas id="canvas"></canvas></div>
    <script>
        const canvas = document.querySelector("#canvas");
        const ctx = canvas.getContext("2d");
        const imgLoader = document.querySelector("#imgLoader");
        imgLoader.addEventListener("change", handleUpload);
        function handleUpload(e) {
            console.log(e);
            const reader = new FileReader();
            reader.onload = function (e) {
                console.log(e);
                const img = new Image();
                img.onload = function () {
                    canvas.width = img.width / 2;
                    canvas.height = img.height / 2;
                    ctx.drawImage(img, 0, 0, img.width / 2,
                                  img.height / 2);
                }
                img.src = e.target.result;
            }
            reader.readAsDataURL(e.target.files[0]);
        }
    </script>
</body>
</html>
```

Практическое занятие 14.6

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
    <style>
        #canvas {
            border: 1px solid black;
        }
    </style>
</head>

<body>
    <div><canvas id="canvas"></canvas></div>
    <script>
        const canvas = document.getElementById("canvas");
        const ctx = canvas.getContext("2d");
        const ballSize = 10;
        let x = canvas.width / 2;
        let y = canvas.height / 2;
        let dirX = 1;
        let dirY = 1;

        function drawBall() {
            ctx.beginPath();
            ctx.arc(x, y, ballSize, 0, Math.PI * 2);
            ctx.fillStyle = "red";
            ctx.fill();
            ctx.closePath();
        }

        function move() {
            ctx.clearRect(0, 0, canvas.width, canvas.height);
            drawBall();
            if (x > canvas.width - ballSize || x < ballSize) {
                dirX *= -1;
            }
            if (y > canvas.height - ballSize || y < ballSize) {
                dirY *= -1;
            }
            x += dirX;
            y += dirY;
        }
        setInterval(move, 10);
    </script>
</body>
</html>
```

Практическое занятие 14.7

```
<script>
    window.onload = init;
    const canvas = document.getElementById("canvas");
    const ctx = canvas.getContext("2d");
    canvas.style.border = "1px solid black";
    const penColor = document.querySelector("#penColor");
    const penWidth = document.querySelector("#penWidth");
    document.querySelector(".clear").addEventListener(
        "click", clearImg);
    canvas.width = 700;
    canvas.height = 700;
    let pos = {
        x: 0,
        y: 0,
    };

    function init() {
        canvas.addEventListener("mousemove", draw);
        canvas.addEventListener("mousemove", setPosition);
        canvas.addEventListener("mouseenter", setPosition);
    }

    function draw(e) {
        if (e.buttons !== 1) return;
        ctx.beginPath();
        ctx.moveTo(pos.x, pos.y);
        setPosition(e);
        ctx.lineTo(pos.x, pos.y);
        ctx.strokeStyle = penColor.value;
        ctx.lineWidth = penWidth.value;
        ctx.lineCap = "round";
        ctx.stroke();
    }

    function setPosition(e) {
        pos.x = e.pageX;
        pos.y = e.pageY;
    }

    function clearImg() {
        const temp = confirm("Clear confirm?");
        if (temp) {
            ctx.clearRect(0, 0, canvas.offsetWidth,
                          canvas.offsetHeight);
        }
    }
</script>
```

Проекты

Создание эффекта матрицы

```
<!doctype html>
<html>
<head>
    <title>Canvas HTML5</title>
</head>
<body>
    <div class="output"></div>
    <script>
        const canvas = document.createElement("canvas");
        const ctx = canvas.getContext("2d");
        canvas.setAttribute("width", "500");
        canvas.setAttribute("height", "300");
        document.body.prepend(canvas);
        const colVal = [];
        for(let x=0;x<50;x++){
            colVal.push(0);
        }

        function matrix() {
            ctx.fillStyle = "rgba(0,0,0,.05)";
            ctx.fillRect(0, 0, canvas.width, canvas.height);
            ctx.fillStyle = "green";
            colVal.map((posY, index) => {
                let output = Math.random()<0.5?0:1;
                let posX = (index * 10) + 10;
                ctx.fillText(output, posX, posY);
                if (posY > 100 + Math.random() * 300) {
                    colVal[index] = 0;
                } else {
                    colVal[index] = posY + 10;
                }
            });
        }
        setInterval(matrix, 50);
    </script>
</body>
</html>
```

Таймер обратного отсчета

```
    <script>
            const endDate = document.querySelector("input[name='endDate']");
            const clock = document.querySelector(".clock");
            let timeInterval;
```

```
        let timeStop = true;
        const savedValue = localStorage.getItem("countdown") || false;
        if (savedValue) {
            startClock(savedValue);
            let inputValue = new Date(savedValue);
            endDate.valueAsDate = inputValue;
        }
        endDate.addEventListener("change", function (e) {
        e.preventDefault();
        clearInterval(timeInterval);
        const temp = new Date(endDate.value);
        localStorage.setItem("countdown", temp);
        startClock(temp);
        timeStop = true;
    });
    function startClock(d) {
        function updateCounter() {
            let tl = (timeLeft(d));
            if (tl.total <= 0) {
                timeStop = false;
            }
            for (let pro in tl) {
                let el = clock.querySelector("." + pro);
                if (el) {
                    el.innerHTML = tl[pro];
                }
            }
        }
        updateCounter();
        if (timeStop) {
            timeInterval = setInterval(updateCounter, 1000);
        } else {
            clearInterval(timeInterval);
        }
    }
    function timeLeft(d) {
        let currentDate = new Date();
        let t = Date.parse(d) - Date.parse(currentDate);
        let seconds = Math.floor((t / 1000) % 60);
        let minutes = Math.floor((t / 1000 / 60) % 60);
        let hours = Math.floor((t / (1000 * 60 * 60)) % 24);
        let days = Math.floor(t / (1000 * 60 * 60 * 24));
        return {
            "total": t,
            "days": days,
            "hours": hours,
            "minutes": minutes,
            "seconds": seconds
        };
    }
</script>
```

Онлайн-приложение для рисования

```
<script>
    const canvas = document.querySelector("#canvas");
    const ctx = canvas.getContext("2d");
    const penColor = document.querySelector("#penColor");
    const penWidth = document.querySelector("#penWidth");
    const btnSave = document.querySelector(".save");
    const btnClear = document.querySelector(".clear");
    const output = document.querySelector(".output");
    const mLoc = {
        draw: false,
        x: 0,
        y: 0,
        lastX: 0,
        lastY: 0
    };
    canvas.style.border = "1px solid black";
    btnSave.addEventListener("click", saveImg);
    btnClear.addEventListener("click", clearCanvas);
    canvas.addEventListener("mousemove", (e) => {
        mLoc.lastX = mLoc.x;
        mLoc.lastY = mLoc.y;
        // console.log(e);
        mLoc.x = e.clientX;
        mLoc.y = e.clientY;
        draw();
    });
    canvas.addEventListener("mousedown", (e) => {
        mLoc.draw = true;
    });
    canvas.addEventListener("mouseup", (e) => {
        mLoc.draw = false;
    });
    canvas.addEventListener("mouseout", (e) => {
        mLoc.draw = false;
    });
    function saveImg() {
        const dataURL = canvas.toDataURL();
        console.log(dataURL);
        const img = document.createElement("img");
        output.prepend(img);
        img.setAttribute("src", dataURL);
        const link = document.createElement("a");
        output.append(link);
        let fileName = Math.random().toString(16).substr(-8) +
            ".png"
        link.setAttribute("download", fileName);
        link.href = dataURL;
        link.click();
        output.removeChild(link);
```

```
        }
        function clearCanvas() {
            let temp = confirm("clear canvas?");
            if (temp) {
                ctx.clearRect(0, 0, canvas.width, canvas.height);
            }
        }
        function draw() {
            if (mLoc.draw) {
                ctx.beginPath();
                ctx.moveTo(mLoc.lastX, mLoc.lastY);
                ctx.lineTo(mLoc.x, mLoc.y);
                ctx.strokeStyle = penColor.value;
                ctx.lineWidth = penWidth.value;
                ctx.stroke();
                ctx.closePath();
            }
        }
    </script>
```

Вопросы для самопроверки

1. `const canvas = document.getElementById('canvas');`
   `const ctx = canvas.getContext('2d');`
2. Нарисует красный круг.
3. `moveTo()`, `lineTo()`, `stroke()`

```
    ctx.moveTo(100, 0);
    ctx.lineTo(100, 100);
    ctx.stroke();
```

Глава 15. Дальнейшие шаги

Практическое занятие

Практическое занятие 15.1

```
[
    {
        "name": "Learn JavaScript",
        "status" : true
    },
    {
        "name": "Try JSON",
        "status" : false
    }
]
```

```
const url = "list.json";
fetch(url).then(rep => rep.json())
.then((data) => {
data.forEach((el) => {
        console.log(`${el.name} = ${el.status}`);
    });
});
```

Проекты

Работа с JSON

```
<!DOCTYPE html>
<html>
    <head><title>Working with JSON Project</title></head>
    <body>
        <script src="myscript.js"></script>
    </body>
</html>

// myscript.js

let url = "people.json";
fetch(url)
.then(response => response.json())
.then(data => {
    console.log(data);
    data.forEach(person => {
        console.log(`${person.first} ${person.last} - ${person.topic}`);
    });
});

// people.json
[
    {
        "first": "Laurence",
        "last": "Svekis",
        "topic": "JavaScript"
    },
    {
        "first": "John",
        "last": "Smith",
        "topic": "HTML"
    },
    {
        "first": "Jane",
        "last": "Doe",
        "topic": "CSS"
    }
]
```

Создание списков

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript List Project</title>
</head>
<body>
    <div class="output"></div>
    <input type="text"><button>add</button>
    <script>
        const output = document.querySelector(".output");
        const myValue = document.querySelector("input");
        const btn1 = document.querySelector("button");
        const url = "list.json";
        btn1.addEventListener("click", addToList);
        let localData = localStorage.getItem("myList");
        let myList = [];
        window.addEventListener("DOMContentLoaded", () => {
            output.textContent = "Loading......";
            if (localData) {
                myList = JSON.parse(localStorage.getItem("myList"));
                output.innerHTML = "";
                myList.forEach((el, index) => {
                    maker(el);
                });
            } else {
                reloader();
            }
        });

        function addToList() {
            if (myValue.value.length > 3) {
                const myObj = {
                    "name": myValue.value
                }
                myList.push(myObj);
                maker(myObj);
                savetoStorage();
            }
            myValue.value = "";
        }
        function savetoStorage() {
            console.log(myList);
            localStorage.setItem("myList", JSON.stringify(myList));
        }
        function reloader() {
            fetch(url).then(rep => rep.json())
            .then((data) => {
                myList = data;
                myList.forEach((el, index) => {
```

```
                    maker(el);
                });
                savetoStorage();
            });
        }
        function maker(el) {
            const div = document.createElement("div");
            div.innerHTML = `${el.name}`;
            output.append(div);
        }
    </script>
</body>
</html>
```

Вопросы для самопроверки

1. Предварительно запрограммированные модули JavaScript, которые вы можете использовать для ускорения процесса разработки.
2. Откройте консоль и наберите `$` или `jQuery`. Если в качестве ответа вы получите объект `jQuery`, значит, на странице есть ссылка на `$` или `jQuery`.
3. UnderscoreJS.
4. Наберите `node` и название файла на компьютере в каталоге файлов.

Лоренс Ларс Свекис, Майке ван Путтен, Роб Персиваль

JavaScript с нуля до профи

Перевел с английского С. Черников

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*К. Тарасевич*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Терех*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 15.02.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1000. Заказ 0000.