

ЛЕГКОЕ  
ПРОГРАММИРОВАНИЕ

# JAVASCRIPT ДЛЯ ДЕТЕЙ

САМОУЧИТЕЛЬ ПО ПРОГРАММИРОВАНИЮ

НИК МОРГАН



МИФ  
ДЕТСТВО



**By Nick Morgan**

# JAVASCRIPT FOR KIDS

**A PLAYFUL  
INTRODUCTION TO PROGRAMMING**



San Francisco

**Ник Морган**

# **JAVASCRIPT ДЛЯ ДЕТЕЙ**

**САМОУЧИТЕЛЬ  
ПО ПРОГРАММИРОВАНИЮ**

Москва  
«Манн, Иванов и Фербер»  
2016

УДК 087.5:004.43  
ББК 76.1,62:32.973.412  
М79

Перевод с английского Станислава Ломакина

Издано с разрешения *No Starch Press, Inc., a California Corporation*

*На русском языке публикуется впервые*

Возрастная маркировка в соответствии  
с Федеральным законом № 436-ФЗ: 6+

**Морган, Ник**

М79 JavaScript для детей. Самоучитель по программированию / Ник Морган ;  
пер. с англ. Станислава Ломакина ; [науч. ред. Д. Абрамова]. — М. : Манн,  
Иванов и Фербер, 2016. — 288 с.

ISBN 978-5-00100-295-6

Эта книга позволит вам погрузиться в программирование и с легкостью  
освоить JavaScript. Вы напишете несколько настоящих игр — поиск сокро-  
вищ на карте, «Виселицу» и «Змейку». На каждом шаге вы сможете оценить  
результаты своих трудов — в виде работающей программы, а с понятными  
инструкциями, примерами и забавными иллюстрациями обучение будет  
только приятным. Книга для детей от 10 лет.

УДК 087.5:004.43  
ББК 76.1,62:32.973.412

Все права защищены. Никакая часть данной книги не  
может быть воспроизведена в какой бы то ни было форме  
без письменного разрешения владельцев авторских прав.  
Правовую поддержку издательства обеспечивает юриди-  
ческая фирма «Вегас-Лекс».

**VEGAS LEX**

ISBN 978-5-00100-295-6

Copyright © 2014 by Nick Morgan.  
Title of English-language original: *JavaScript for Kids*,  
ISBN 978-1-59327-408-5, published by No Starch Press.  
© Перевод на русский язык, издание на русском языке,  
оформление. ООО «Манн, Иванов и Фербер», 2016

# ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ .....	11
----------------	----

## ЧАСТЬ I. ОСНОВЫ

1. ЧТО ТАКОЕ JAVASCRIPT? .....	17
--------------------------------	----

Знакомьтесь: JavaScript .....	17
Зачем изучать JavaScript? .....	19
Пробуем JavaScript .....	19
Строение JavaScript-программы .....	21
Что мы узнали .....	24

2. ТИПЫ ДАННЫХ И ПЕРЕМЕННЫЕ .....	25
-----------------------------------	----

Числа и операторы .....	26
Переменные .....	28
Строки .....	35
Булевые значения .....	41
Undefined и null .....	48
Что мы узнали .....	48

3. МАССИВЫ .....	49
------------------	----

Зачем нужны массивы? .....	49
Создание массива .....	50

Доступ к элементам массива .....	52
Создание и изменение элементов .....	53
Разные типы данных в одном массиве .....	54
Работаем с массивами .....	55
Что полезного можно сделать с массивами .....	63
Что мы узнали.....	68
<b>4. ОБЪЕКТЫ .....</b>	<b>70</b>
Создание объектов .....	70
Доступ к значениям внутри объектов .....	72
Добавление элементов объекта.....	73
Массивы объектов .....	75
Исследование объектов в консоли .....	77
Что полезного можно сделать с объектами .....	79
Что мы узнали.....	81
<b>5. ОСНОВЫ HTML .....</b>	<b>83</b>
Текстовые редакторы .....	84
Наш первый HTML-документ .....	84
Теги и элементы .....	85
Полноценный HTML-документ .....	89
Иерархия HTML .....	90
Добавим в HTML ссылки .....	91
Что мы узнали.....	94
<b>6. УСЛОВИЯ И ЦИКЛЫ .....</b>	<b>95</b>
Внедрение JavaScript-кода в HTML .....	95
Условные конструкции .....	97
Циклы .....	101
Что мы узнали.....	107
<b>7. ПИШЕМ ИГРУ «ВИСЕЛИЦА» .....</b>	<b>110</b>
Взаимодействие с игроком .....	111
Проектирование игры .....	114
Программируем игру .....	117
Код игры .....	122
Что мы узнали.....	124

<b>8. ФУНКЦИИ</b>	126
Базовое устройство функции	126
Создаем простую функцию	127
Вызов функции	127
Передача аргументов в функцию	128
Возврат значения из функции	131
Вызов функции в качестве значения	132
Упрощаем код с помощью функций	133
Ранний выход из функции по return	136
Многократное использование return вместо конструкции if... else	137
Что мы узнали	139

## ЧАСТЬ II. ПРОДВИНУТЫЙ JAVASCRIPT

<b>9. DOM И JQUERY</b>	145
Поиск элементов DOM	146
Работа с деревом DOM через jQuery	149
Создание новых элементов через jQuery	150
Анимация элементов средствами jQuery	152
Цепной вызов и анимация на jQuery	152
Что мы узнали	154
<b>10. ИНТЕРАКТИВНОЕ ПРОГРАММИРОВАНИЕ</b>	156
Отложенное выполнение кода и setTimeout	156
Отмена действия таймера	158
Многократный запуск кода и setInterval	158
Анимация элементов с помощью setInterval	160
Реакция на действия пользователя	162
Что мы узнали	164
<b>11. ПИШЕМ ИГРУ «НАЙДИ КЛАД!»</b>	166
Проектирование игры	166
Создаем веб-страницу с HTML-кодом	167
Выбор случайного места для клада	168
Обработчик кликов	169
Код игры	173
Что мы узнали	175

<b>12. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ .....</b>	176
Простой объект .....	176
Добавление к объектам новых методов .....	177
Создание объектов с помощью конструкторов .....	180
Рисуем машины .....	182
Настройка объектов через прототипы .....	184
Что мы узнали .....	188
<b>ЧАСТЬ III. ГРАФИКА</b>	
<b>13. ЭЛЕМЕНТ CANVAS .....</b>	193
Создаем «холст» .....	193
Рисование на «холсте» .....	194
Выбор цвета .....	196
Рисование контуров прямоугольников .....	197
Рисование линий или путей .....	198
Заливка путей цветом .....	200
Рисование дуг и окружностей .....	201
Рисование нескольких окружностей с помощью функции .....	204
Что мы узнали .....	205
<b>14. АНИМАЦИИ С ПОМОЩЬЮ CANVAS .....</b>	208
Движение по странице .....	208
Изменение размера квадрата .....	210
Случайная пчела .....	211
Отскакивающий мяч .....	217
Что мы узнали .....	222
<b>15. УПРАВЛЕНИЕ АНИМАЦИЯМИ С КЛАВИАТУРЫ .....</b>	224
События клавиатуры .....	224
Управляем мячом с клавиатуры .....	227
Код программы .....	233
Запуск программы .....	235
Что мы узнали .....	235
<b>16. ПИШЕМ ИГРУ «ЗМЕЙКА»: ЧАСТЬ 1 .....</b>	237
Игровой процесс .....	237

Структура игры .....	238
Начинаем писать игру .....	240
Рисуем рамку .....	243
Отображение счета .....	245
Конец игры .....	249
Что мы узнали .....	250
<b>17. ПИШЕМ ИГРУ «ЗМЕЙКА»: ЧАСТЬ 2 .....</b>	<b>252</b>
Создаем конструктор Block .....	252
Создаем змейку .....	257
Перемещаем змейку .....	259
Управляем змейкой с клавиатуры .....	264
Создаем яблоко .....	266
Код игры .....	268
Что мы узнали .....	273
<b>ПОСЛЕСЛОВИЕ:</b>	
<b>КУДА ДВИГАТЬСЯ ДАЛЬШЕ .....</b>	<b>276</b>
Больше о JavaScript .....	276
Веб-программирование .....	277
Графическое программирование .....	278
3D-программирование .....	278
Программирование роботов .....	279
Программирование звука .....	279
Программирование игр .....	279
Обмен кодом с помощью JSFiddle .....	280
<b>ГЛОССАРИЙ .....</b>	<b>281</b>
<b>ОБ АВТОРЕ .....</b>	<b>286</b>
<b>БЛАГОДАРНОСТИ .....</b>	<b>287</b>

*Посвящается Филли  
(и Оладушку)*

## ВВЕДЕНИЕ

Эта книга научит вас писать программы на JavaScript — одном из популярных языков программирования. А освоив язык программирования, вы станете программистом — человеком, который не просто пользуется компьютерами, а управляет ими. Научившись программированию, вы сможете вертеть компьютерами как хотите, и они всегда будут послушно следовать вашим указаниям.

Изучить именно JavaScript — отличная идея, потому что этот язык используется повсюду. Его поддерживают браузеры Chrome, Firefox и Internet Explorer. Возможности JavaScript позволяют программистам делать из обычных веб-страниц полноценные интерактивные приложения и видеоигры. Но это еще не все: JavaScript также работает на интернет-серверах и даже может использоваться для управления роботами и другими устройствами.

### Для кого эта книга?

Эта книга предназначена для всех, кто хочет изучить именно JavaScript или же просто начать программировать с нуля. Она написана для детей, но может стать первым самоучителем по программированию для человека любого возраста.

Работая с книгой, вы будете постепенно узнавать новое, закреплять прочитанное и двигаться дальше и дальше. Начав с простых типов данных, вы перейдете к более сложным, по пути освоив управляющие конструкции и функции. После этого вы научитесь писать код, реагирующий на перемещения мышки или нажатия клавиш, и наконец познакомитесь с элементом canvas, который позволяет создавать рисунки и анимации — любые, какие только пожелаете!

## Как читать эту книгу

Самое главное, читайте по порядку! Может быть, этот совет звучит странно, однако нередко людям не терпится сразу перейти к чему-нибудь занимательному, например к созданию игр. Но поверьте — вам будет гораздо проще создать игру, если вы все-таки будете читать с начала, глава за главой, так как каждый новый раздел основывается на материале предыдущих.

Языки программирования похожи на обычные языки. Вы, наверное, знаете — чтобы овладеть языком, нужно выучить грамматику и запомнить достаточно много слов. Это требует времени. Это же правило работает и с JavaScript — чтобы научиться пользоваться этим языком, нужно постоянно исследовать код и писать на нем программы. По мере того как вы будете писать больше и больше, вы обнаружите, что пользуетесь командами все более естественно, и в конце концов сможете свободно выражать свои мысли в коде.

Я настоятельно рекомендую тестировать примеры кода по мере чтения книги. Если вам не до конца понятно, как код работает, попробуйте вносить в него небольшие изменения и смотреть, как изменится результат. Если же ваши правки не приводят к ожидаемому эффекту, постарайтесь выяснить, почему это происходит.

Обязательно выполняйте задания из разделов «Попробуйте сами» и «Упражнения». Вводить в компьютер код из книги — отличное начало, но по-настоящему вы станете понимать программирование только тогда, когда начнете писать собственный код. Если задания покажутся вам интересными, не останавливайтесь! Придумывайте свои задачи по усовершенствованию написанных вами программ.

Вы можете найти примеры выполнения заданий и исходный код игр по адресу [www.nostarch.com/javascriptforkids](http://www.nostarch.com/javascriptforkids) или на странице книги на сайте [www.mann-ivanov-ferber.ru](http://www.mann-ivanov-ferber.ru). Постарайтесь заглядывать в решения лишь после того, как выполните задания, чтобы сравнить свой подход с моим. И только если вы зашли в тупик, обратитесь за подсказкой. Однако помните, что это лишь варианты решения — в JavaScript существует множество способов выполнить одну и ту же задачу, так что не беспокойтесь, если ваше решение получится совсем не похожим на мое.

Если вы встретите слово, значение которого не понимаете, загляните в глоссарий в конце книги.

## Что вас ждет?

Глава 1 содержит краткое введение в JavaScript. Кроме того, вы узнаете, как писать код в консоли Google Chrome.

Глава 2 расскажет про переменные и основные типы данных в JavaScript: числа, строки и булевые значения.

**Глава 3** посвящена массивам, предназначенным для хранения наборов других элементов данных.

**Глава 4** расскажет об объектах, содержащих пары «ключ-значение».

**Глава 5** — это введение в HTML, язык для создания веб-страниц.

**Глава 6** научит, как управлять выполнением кода с помощью конструкций `if`, циклов `for` и других структур.

**Глава 7** покажет, как на основе изученного материала создать простую игру на отгадывание слов — «Виселицу».

**Глава 8** научит писать собственные функции, что позволит группировать фрагменты кода и использовать их повторно.

**Глава 9** — это введение в jQuery, инструмент, облегчающий управление веб-страницами из JavaScript-кода.

**Глава 10** научит, как использовать таймеры, интервалы и обработчики событий, делая код более интерактивным.

**Глава 11** использует функции, jQuery и обработчики событий для создания игры «Найди клад!».

**Глава 12** научит элементам объектно-ориентированного программирования.

**Глава 13** расскажет об элементе `canvas`, позволяющем создавать графические изображения на веб-страницах.

**Глава 14** на основе способов анимации из главы 10 покажет, как создавать анимации на «холсте» `canvas`,

тогда как

**Глава 15** научит, как управлять этими анимациями с клавиатуры.

В главах **16** и **17** вы создадите полноценную игру «Змейка», используя все знания, полученные в предыдущих пятнадцати главах!

**Послесловие** подскажет, куда двигаться дальше при изучении программирования.

**Глоссарий** даст определения множества новых слов, которые вам встретятся.

## **Повеселитесь!**

И еще один момент, о котором не стоит забывать: веселитесь! Программирование может быть увлекательным, творческим занятием, как рисование или игры (а работая с книгой, вы изрядно порисуете и поиграете). Как только вы научитесь программировать, для вас не будет иных препятствий, кроме собственного воображения. Добро пожаловать в потрясающий мир компьютерного программирования — и желаю вам отлично провести время!



ЧАСТЬ I

# Основы



# 1

## ЧТО ТАКОЕ JAVASCRIPT?

Компьютеры — необычайно мощные машины, способные делать потрясающие вещи. Например, они могут играть в шахматы, обслуживать тысячи интернет-страничек и менее чем за несколько секунд выполнять миллионы сложных вычислений. Однако сами по себе компьютеры неразумны, и делают они лишь то, что прикажут люди. Мы сообщаем компьютерам, что нам от них нужно, с помощью наборов инструкций, которые называются программами. Без программ компьютеры вообще ничего не умеют!



### Знакомьтесь: JavaScript

Более того, компьютеры не знают ни английского, ни русского, ни других естественных языков; и компьютерные программы создают на специальных языках *программирования*. Одним из таких языков является JavaScript. Даже если вы слышите про JavaScript впервые, вы определенно заходили на сайты, которые его используют. Например, JavaScript может управлять внешним видом странички или делать так, чтобы страница реагировала на нажатие клавиши или перемещение мышки.

Такие сайты, как Gmail, Facebook и Twitter, используют JavaScript для облегчения работы с почтой, отправки комментариев или улучшения навигации. К примеру, когда вы, читая в Twitter сообщения от @nostarch, проматываете страничку вниз и видите все больше и больше сообщений, это происходит благодаря JavaScript.

Чтобы понять, чем же так хорош JavaScript, достаточно посетить несколько сайтов:

- JavaScript позволяет проигрывать музыку и создавать яркие визуальные эффекты. Например, вы можете полетать в интерактивном видеоклипе от студии HelloEnjoy на песню Элли Голдинг Lights (<http://lights.helloenjoy.com/>), рис. 1.1.



Рис. 1.1. В клипе *Lights* нужно управлять искрящимся курсором

- С помощью JavaScript можно создавать инструменты для творчества. Patatap (<http://www.patatap.com/>) — это нечто вроде виртуальной драм-машины, которая издает всевозможные шумы и звуки, а также проигрывает забавные анимации, рис. 1.2.

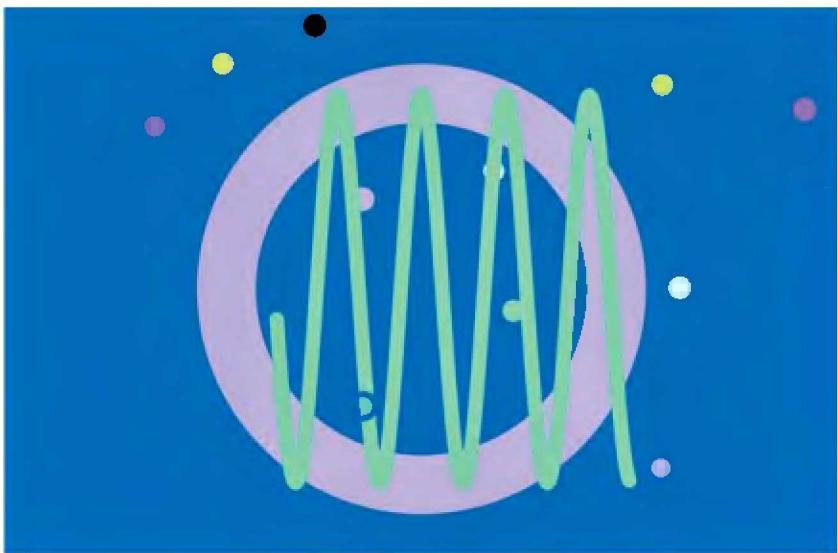


Рис. 1.2. Зайдя на страницу Patatap, нажимайте на разные клавиши, чтобы услышать разные звуки!

- JavaScript дает нам возможность играть в увлекательные игры. CubeSlam (<https://www.cubeslam.com/>) — это трехмерное подобие классической игры «Понг», похожее на аэрохоккей. Посоревнуйтесь с кем-нибудь из друзей или с медведем, за которого играет компьютер. См. рис. 1.3.



Рис. 1.3. Игра CubeSlam написана целиком на JavaScript!

## Зачем изучать JavaScript?

JavaScript — далеко не единственный язык программирования. В сущности, языков очень много, счет идет на сотни, однако есть немало причин выбрать именно JavaScript. Например, изучать его гораздо проще (и интереснее), чем многие другие языки. Но, пожалуй, самая веская причина такова: чтобы писать и выполнять JavaScript-программы, достаточно интернет-браузера — такого, как Internet Explorer, Mozilla Firefox или Google Chrome. В каждый из этих браузеров встроен интерпретатор JavaScript, который сможет выполнять JavaScript-программы. И никакого специального программного обеспечения вам не понадобится.

Написав программу на JavaScript, отправьте ссылку на нее другим людям, и они тоже смогут ее запустить — у себя на компьютере, в браузере (см. «Обмен кодом с помощью JSFiddle» на с. 280).

## Пробуем JavaScript

Давайте напишем простую JavaScript-программку с помощью браузера Google Chrome ([www.google.com/chrome](http://www.google.com/chrome)). Установите Chrome на свой компьютер (если он еще не установлен), запустите его и введите слова `about:blank` в адресной строке. Теперь нажмите ENTER — откроется пустая страничка, как на рис. 1.4.

Начнем с программирования в JavaScript-консоли Chrome (это секретный инструмент для тестирования коротких программ на JavaScript). Если ваш компьютер работает под управлением Microsoft Windows или Linux, нажмите и не отпускайте клавиши CTRL и SHIFT, а затем нажмите J. Если же вы пользуетесь системой MacOS, нажмите и удерживайте COMMAND и OPTION, а затем нажмите J.

Если вы все сделали правильно, то увидите пустую веб-страницу, под которой стоит значок угловой скобки (>), а после него мигает курсор (|). Здесь нам предстоит писать код на языке JavaScript!



Текст в консоли Chrome подсвечивается разными цветами в зависимости от типа данных. В этой книге код для ввода в консоль напечатан такими же цветами там, где это имеет значение. Но там, где разноцветный код будет вас только отвлекать, синим мы будем выделять то, что сами вводим в консоль, а данные, которые автоматически выдаст программа, будут цветными.

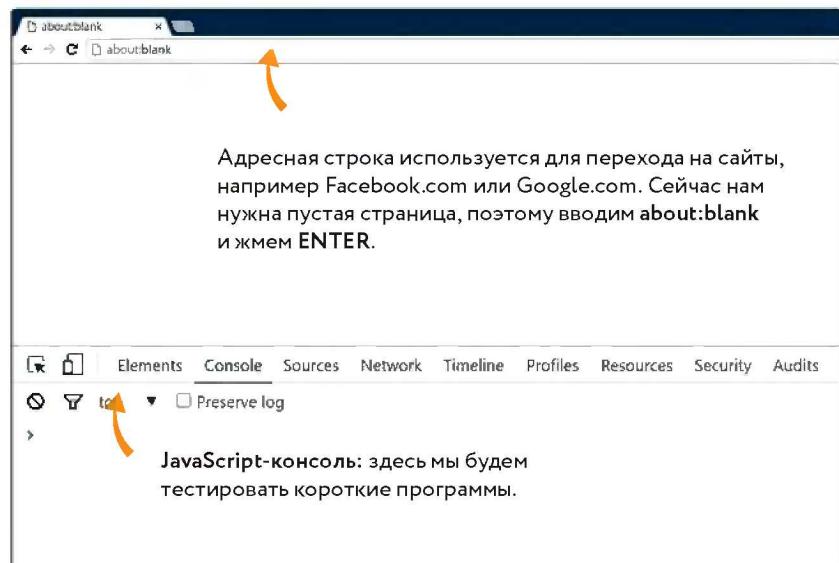


Рис. 1.4. JavaScript-консоль Google Chrome

Когда вы введете код и нажмете ENTER, JavaScript должен запустить (иначе говоря, выполнить) ваш код, показав на следующей строке результат (когда он есть). Например, введите в консоли:

---

```
3 + 4;
```

---

Теперь нажмите ENTER. JavaScript должен напечатать результат сложения (7) на следующей строке:

---

```
3 + 4;  
7
```

---

Как видите, ничего сложного. Но JavaScript — это нечто определенно большее, чем просто затейливый калькулятор. Давайте попробуем кое-что еще.

## Строение JavaScript-программы

Давайте позабавимся — напишем JavaScript-программу, которая печатает японские смайлики каомодзи в виде кошачьей мордочки:

---

```
=^ . ^=
```

---

В отличие от простого сложения, с которого мы начали, эта программа занимает несколько строк. Чтобы ввести ее в консоль, нужно будет в конце каждой строки переходить на новую строку **нажатием SHIFT-ENTER**. (Если нажать просто ENTER, Chrome попытается выполнить те команды, которые вы уже ввели, и программа не будет работать правильно. Сами по себе компьютеры ничего не соображают — я предупреждал!)

Введите в консоли браузера:

---

```
// Рисуем столько котиков, сколько захотим!  
var drawCats = function (howManyTimes) {  
    for (var i = 0; i < howManyTimes; i++) {  
        console.log(i + " =^ . ^=");  
    }  
};  
drawCats(10); // Вместо 10 тут может быть другое число
```

---



**Draw cats** —  
рисовать  
котиков

**Function** —  
функция

**How many times** —  
сколько раз

В конце последней строки нажмите ENTER, а не SHIFT-ENTER. Программа должна напечатать следующее:



```
0 =^ .^=
1 =^ .^=
2 =^ .^=
3 =^ .^=
4 =^ .^=
5 =^ .^=
6 =^ .^=
7 =^ .^=
8 =^ .^=
9 =^ .^=
```

Если при вводе программы вы где-то ошиблись, результат может оказаться другим — возможно, вы даже получите сообщение об ошибке. Это я и имел в виду, говоря, что компьютеры неразумны, — даже простейшая программа должна быть написана идеально, чтобы компьютер понял, что от него требуется!

Я не буду сейчас вдаваться в подробности, объясняя, как работает этот код (мы еще вернемся к нему в восьмой главе), однако давайте рассмотрим некоторые особенности этой программы, да и JavaScript-программ в целом.

## Синтаксис

В нашей программе встречается много символов, таких как скобки (), точки с запятой ;, фигурные скобки {}, знаки плюс +, а также некоторые таинственные на первый взгляд слова (например, var и console.log). Все это является частью *синтаксиса* JavaScript — то есть правил, указывающих, как объединять символы и слова, чтобы составить работающую программу.

Одна из главных сложностей при освоении нового языка программирования — запомнить правила написания команд. Поначалу легко пропустить какие-нибудь скобки или запутаться в очередности записи значений. Не волнуйтесь, с опытом вы привыкнете писать код правильно.

В этой книге мы будем изучать материал медленно, постепенно знакомясь с новыми командами языка, чтобы вы могли писать все более и более мощные программы.

## Комментарии

В первой строке нашей программы написано:

```
// Рисуем столько котиков, сколько захотим!
```

Это называется **комментарием**. Программисты пишут комментарии, чтобы другим программистам было легче читать и понимать их код. Компьютер же комментарии игнорирует. В JavaScript комментарии начинаются с двух символов наклонной черты (//). Все, что идет следом за ними (в той же строке), интерпретатор JavaScript пропускает, поэтому комментарии не оказывают влияния на выполнение программы — это всего лишь пояснение.

В примерах кода, которые встретятся вам в этой книге, комментарии описывают, что и как там происходит. При написании своего кода тоже добавляйте комментарии — когда вы заглянете в программу некоторое время спустя, они напомнят вам, как работает код и что происходит на том или ином этапе.

В конце нашей программы-примера есть еще один комментарий. Напоминаю: все, что записано после символов //, компьютер игнорирует!

---

```
drawCats(10); // Вместо 10 тут может быть другое число
```

---

Комментарии могут занимать отдельную строку или следовать сразу после кода. Но если вы поставите // перед кодом, вот так:

---

```
// drawCats(10);
```

---

...то не произойдет вообще ничего! Chrome решит, что вся эта строка — комментарий, хоть там и записаны инструкции на языке JavaScript.

Когда вы, помимо примеров в этой книге, начнете изучать чужой JavaScript-код, вам будут попадаться комментарии, которые выглядят иначе:

---

```
/*
Рисуем столько котиков,
сколько захотим!
*/
```

---

Это другая разновидность комментариев; их обычно используют, когда текст примечания не помещается на одной строке. Однако принцип здесь тот же: текст, записанный между /\* и \*/, — это комментарий, и выполнять его компьютер не будет.

## Что мы узнали



В этой главе мы познакомились с языком JavaScript и узнали, что можно делать с его помощью. Кроме того, мы научились запускать JavaScript-код в браузере Google Chrome и ввели несложную программу-пример. Все примеры из этой книги можно (и нужно!) запускать в JavaScript-консоли Google Chrome (если только я не скажу, что этого делать не надо). Просто читать код недостаточно — проверяйте, как он работает! Это единственный способ научиться программировать.

В следующей главе мы приступим к изучению основ языка JavaScript, начиная с трех основных типов данных, с которыми вам предстоит работать: чисел, строк и булевых значений.

# 2

## ТИПЫ ДАННЫХ И ПЕРЕМЕННЫЕ

Программирование — это работа с данными, но что такое данные? Данные — это информация, которая хранится в наших компьютерных программах. Например, ваше имя — это элемент данных, и ваш возраст тоже. Цвет волос, количество братьев и сестер, ваш адрес и пол — все это данные.

В JavaScript есть три основных типа данных: числа, строки и булевые значения. Числа — они и есть числа, тут все понятно. Например, числом можно выразить возраст или рост.

В JavaScript числа записываются так:

---

5;

---

Любые текстовые данные записываются в строки. В JavaScript ваше имя можно выразить строкой (так же как и адрес вашей электронной почты).

Строки выглядят так:

---

"Привет, я строка";

---

Булевые значения могут хранить одну из двух величин — либо это `true` («истина»), либо `false` («ложь»). Например, таким способом можно показать, носите ли вы очки или любите ли вы брокколи.



Пример булева значения:

```
true;
```

С данными разных типов и обращаться следует по-разному. Например, перемножить два числа можно, а перемножить две строки — нет. Зато, имея строку, можно выделить пять ее первых символов. Взяв два булевых значения, можно проверить, являются ли они оба «истиной» (`true`). Вот все эти действия на примере:

```
99 * 123;  
12177  
"Вот длинная строка".slice(0, 3);  
"Вот"  
true && false;  
false
```

Любые данные в JavaScript — не более чем сочетание этих основных типов. Далее мы по очереди рассмотрим каждый тип данных и изучим различные способы работы с ними.



*Наверное, вы заметили, что все эти команды оканчиваются на точку с запятой (;). Этим символом обозначают конец каждой отдельной команды или инструкции языка JavaScript — примерно так же, как точка отмечает конец предложения.*

## Числа и операторы

JavaScript позволяет выполнять основные математические операции, такие как сложение, вычитание, умножение и деление. Для их записи используются символы +, -, \* и /, которые называют **операторами**.

Консоль JavaScript можно использовать как калькулятор. Один из примеров — сложение 3 и 4 — нам уже знаком. Давайте вычислим что-нибудь посложнее: сколько будет 12345 плюс 56789?

```
12345 + 56789;  
69134
```

Посчитать это в уме не так уж просто, а JavaScript мгновенно справился с задачей.

Можно сложить несколько чисел с помощью нескольких знаков «плюс»:

---

```
22 + 33 + 44;  
99
```

---

Также JavaScript умеет вычитать...

---

```
1000 - 17;  
983
```

---

умножать (с помощью символа «звездочка»)...

---

```
123 * 456;  
56088
```

---

и делить (с помощью косой черты — слэша)...

---

```
12345 / 250;  
49.38
```

---

Кроме того, можно объединять эти простые операции, составляя более сложные выражения, вроде такого:

---

```
1234 + 57 * 3 - 31 / 4;  
1397.25
```

---

Есть один нюанс — результат вычислений зависит от порядка, в котором JavaScript выполняет отдельные операции. В математике существует правило, по которому умножение и деление выполняются прежде, чем сложение и вычитание, и JavaScript ему следует.

Порядок, в котором интерпретатор JavaScript выполняет эти операции, показан на рис. 2.1. Сначала он умножает  $57 * 3$ , получая 171 (выделено красным). Затем делит  $31 / 4$ , получая 7.75 (выделено синим). Затем складывает  $1234 + 171$ , получая 1405 (выделено зеленым). И наконец, вычитает 1405 – 7.75, что дает 1397.25 — окончательный результат.

Но как быть, если вы хотите выполнить сложение и вычитание до умножения и деления? Для примера предположим, что у вас есть 1 брат, 3 сестры и 8 карамелек, которые

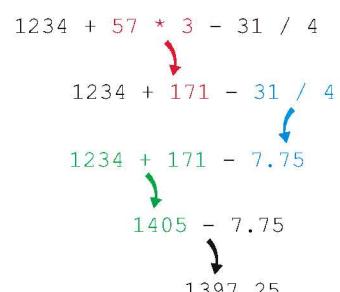


Рис. 2.1. Очередность выполнения операций: умножение, деление, сложение, вычитание

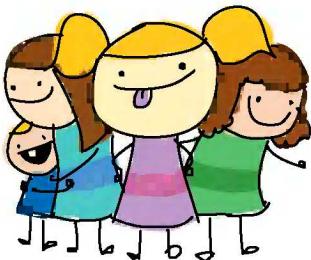
вы решили поровну разделить между ними (свою долю вы уже прикарманили). Нужно разделить 8 на общее количество братьев и сестер.

Попытаемся это сделать:

---

```
8 / 1 + 3;  
11
```

---



Это не может быть верным ответом! Не получится дать каждому родственнику по 11 карамелек, если у вас всего-то 8 конфет! Проблема в том, что JavaScript выполняет деление прежде, чем сложение, то есть он делит 8 на 1 (что равно 8) и затем прибавляет 3, получая в результате 11. Чтобы исправить эту ошибку, заставим JavaScript сначала выполнить сложение, воспользовавшись скобками:

---

```
8 / (1 + 3);  
2
```

---

Так гораздо лучше — вышло по две карамельки каждому из родственников. Скобки вынудили JavaScript сложить 1 и 3 до деления 8 на 4.

### ПОПРОБУЙТЕ!

Предположим, ваша подруга пытается подсчитать с помощью JavaScript, сколько ей нужно купить воздушных шаров. Она устраивает вечеринку и хочет, чтобы каждый из гостей смог надуть по 2 шарика. Сначала было приглашено 15 человек, но потом ваша подруга позвала еще 9.

Она написала такой код:

---

```
15 + 9 * 2;  
33
```

---

Однако ответ, судя по всему, неверен.

Где надо поставить скобки, чтобы JavaScript сначала складывал, а потом умножал, и сколько шариков нужно вашей подруге на самом деле?

## Переменные

Значениям в JavaScript можно давать имена, используя *переменные*. Переменная похожа на ящичек, в который помещается лишь один предмет. Чтобы положить туда что-то еще, прежнее содержимое придется заменить.

Чтобы создать новую переменную, используйте ключевое слово `var`, после которого укажите имя переменной. Ключевое слово — это слово,

обладающее для JavaScript особым значением. В данном случае, когда JavaScript встречает слово var, он понимает, что следом указано имя новой переменной. Например, вот как создать переменную с именем nick:

```
var nick;  
undefined
```

Undefined —  
значение  
не определено

Мы создали новую переменную под названием nick. В ответ консоль выдала undefined — «значение не определено». Однако это не ошибка! JavaScript всегда так делает, если команда не возвращает какого-либо значения. Вы спросите, а что такое «возвращать значение»? Вот пример: когда вы ввели 12345 + 56789;, консоль вернула значение 69134. Однако в JavaScript команда создания переменной никакого значения не возвращает, поэтому интерпретатор печатает undefined.

В этом примере и дальше мы будем давать переменным англоязычные имена, потому что английский — основной язык всей IT-области и программы принято писать только латиницей (кроме комментариев и строковых значений). Использовать русскоязычные имена переменных — это как если при составлении математических уравнений вместо x и y вы использовали бы русские буквы. Можно, но не принято.

Итак, чтобы задать переменной значение, используйте знак «равно»:

```
var age = 12;  
undefined
```

Age — возраст

Задание значения переменной называют *присваиванием* (здесь мы присваиваем значение 12 переменной age). И опять в консоли появляется undefined, поскольку мы только что создали новую переменную. (В дальнейших примерах я буду пропускать это undefined.)

Теперь в интерпретаторе есть переменная age, которой присвоено значение 12. И если ввести в консоли имя age, интерпретатор выдаст значение этой переменной:

```
age;  
12
```

Здорово! При этом значение переменной не высечено в камне (*переменные* потому так и зовутся, что могут менять значения), и, если вам вздумается его обновить, просто используйте знак «равно» еще раз.

```
age = 13;  
13
```

На этот раз я не использовал ключевое слово `var`, поскольку переменная `age` уже существует. Писать `var` нужно только при *создании* переменной, а не при ее *использовании*. И обратите внимание: поскольку мы не создавали новой переменной, команда присваивания вернула значение 13, которое и было напечатано в следующей строке.

Вот чуть более сложный пример — решение задачи про карамельки без помощи скобок:

Number  
of siblings —  
число братьев  
и сестер

Number  
of candies —  
число конфет

---

```
var numberOfSiblings = 1 + 3;  
var numberOfCandies = 8;  
numberOfCandies / numberOfSiblings;
```

---

Сначала мы создали переменную с именем `numberOfSiblings` (количество братьев и сестер) и присвоили ей значение выражения `1 + 3` (которое JavaScript вычислил, получив 4). Потом мы создали переменную `numberOfCandies` (количество карамелек) и присвоили ей значение 8. И наконец, мы ввели: `numberOfCandies / numberOfSiblings`. Поскольку переменная `numberOfCandies` содержит значение 8, а `numberOfSiblings` — 4, JavaScript вычислил, сколько будет `8 / 4`, вернув в результате 2.

## Имена переменных

Вводя имена переменных, будьте внимательны и не допускайте опечаток. Даже если вы перепутаете строчные и заглавные буквы, интерпретатор JavaScript не поймет, чего вы от него хотите! Например, если вы случайно введете имя `numberOfCandies` со строчной буквой `c`, возникнет ошибка:

Reference  
error —  
ошибка  
данных

---

```
numberOfcandies / numberOfSiblings;  
ReferenceError: numberOfcandies is not defined
```

---

Увы, JavaScript следует вашим указаниям буквально. Если вы неправильно ввели имя переменной, JavaScript не поймет, что вы имели в виду, и выдаст сообщение об ошибке.

Еще один нюанс именования переменных в JavaScript — в именах не должно быть пробелов, из-за чего они могут оказаться сложными для чтения. Если бы я назвал переменную `numberofcandies`, без заглавных букв, читать программу стало бы труднее, поскольку неясно, где в этом имени заканчиваются отдельные слова.

Один из обычных способов решения этой проблемы — писать каждое слово с заглавной буквы: `NumberOfCandies`. Такую манеру именования

называют верблюжьей записью, поскольку выпирающие заглавные буквы напоминают верблюжьи горбы.

Имена переменных принято начинать со строчной буквы, поэтому с заглавной буквы обычно пишут все слова имени, кроме самого первого: `numberOfCandies`. В этой книге я также использую эту форму верблюжьей записи; впрочем, вы можете называть свои переменные как вам угодно!

## Создание новых переменных на основе вычислений

Можно создавать новые переменные, выполняя математические действия с переменными, созданными ранее. Давайте с помощью переменных выясним, сколько секунд в году и каков ваш возраст в секундах! Но для начала разберемся, сколько секунд в одном часе.

Сколько секунд в часе

Сначала создадим две новые переменные — `secondsInAMinute` (количество секунд в минуте) и `minutesInAHour` (количество минут в часе) — и присвоим им обеим значение 60 (поскольку, как мы знаем, в минуте 60 секунд, а в часе 60 минут). Теперь создадим переменную `secondsInAHour` (количество секунд в часе), и пусть ее значение равняется `secondsInAMinute` умножить на `minutesInAHour`. И наконец в строке ❶ введем `secondsInAHour`, что означает «покажи мне содержимое переменной `secondsInAHour`», и JavaScript тут же выдаст ответ: 3600.

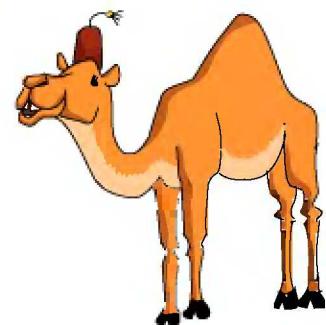
---

```
var secondsInAMinute = 60;
var minutesInAnHour = 60;
var secondsInAnHour = secondsInAMinute * minutesInAnHour;
❶ secondsInAnHour;
3600
```

---

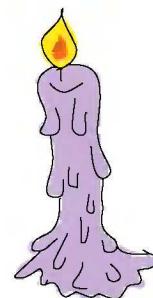
Сколько секунд в сутках

Теперь создадим переменную `hoursInADay` (количество часов в сутках) и присвоим ей значение 24. Затем создадим переменную `secondsInADay` (количество секунд в сутках), и пусть она равняется `secondsInAHour` умножить на `hoursInADay`. Запросив в строке ❶ значение `secondsInADay`, получим 86 400 — именно столько секунд в сутках.



Seconds  
in a minute —  
секунд  
в минуте

Minutes  
in a hour —  
минут в часе



Hours in a day —  
часов в день

---

```
var hoursInADay = 24;
var secondsInADay = secondsInAnHour * hoursInADay;
❶ secondsInADay;
86400
```

---

### Сколько секунд в году

Days in a year —  
дней в году

И наконец, создадим переменные `daysInAYear` (количество дней в году) и `secondsInAYear` (количество секунд в году): `daysInAYear` присвоим значение 365, а `secondsInAYear` пусть равняется `secondsInADay` умножить на `daysInAYear`. Запрашиваем значение `secondsInAYear` и видим, что это число 31 536 000 (более 31 миллиона секунд!).

---

```
var daysInAYear = 365;
var secondsInAYear = secondsInADay * daysInAYear;
secondsInAYear;
31536000
```

---

### Возраст в секундах

Теперь, зная, сколько секунд в году, вы можете запросто узнать свой возраст в секундах (с точностью до последнего дня рождения). К примеру, когда я пишу эти строки, мне 29 лет:

---

```
var age = 29;
age * secondsInAYear;
914544000
```

---

Чтобы вычислить свой возраст в секундах, введите тот же самый код, но замените значение переменной `age` на ваш возраст. Или просто замените эту переменную на число, соответствующее вашему возрасту:

---

```
29 * secondsInAYear;
914544000
```

---

Смотрите-ка, мне исполнилось больше 900 миллионов секунд! А вам?

## Инкремент и декремент

Вам как программисту понадобится увеличивать или уменьшать значения числовых переменных на единицу. Например, у вас в программе может быть переменная для подсчета, сколько раз за день вам сказали

«Дай пять!». И при каждом новом приветствии эту переменную надо будет увеличить на 1.

Увеличение на 1 называют *инкрементом*, а уменьшение на 1 — *декрементом*. Выполняются инкремент и декремент с помощью операторов `++` и `--`.

```
var highFives = 0;  
++highFives;  
1  
++highFives;  
2  
--highFives;  
1
```

High fives —  
дай пять!

После выполнения оператора `++` значение `highFives` (количество приветствий) увеличится на 1, а после выполнения оператора `--` уменьшится на 1. Также эти операторы можно писать после имени переменной — эффект будет прежним, однако после выполнения такой команды JavaScript вернет первоначальное значение переменной, каким оно было до инкремента или декремента.

```
highFives = 0;  
highFives++;  
0  
highFives++;  
1  
highFives;  
2
```

В этом примере мы сначала обнулили значение `highFives`. Команда `highFives++` увеличивает переменную на 1, но число, которое печатает после этого JavaScript, является значением до инкремента. Однако, запрашивая значение `highFives` в самом конце (после двух инкрементов), мы получаем 2.



### **+=(плюс-равно) и -=(минус-равно)**

Чтобы увеличить значение переменной на заданное число, можно написать такой код:

```
var x = 10;  
x = x + 5;  
x;  
15
```

Сначала мы создаем переменную `x` и даем ей значение 10. Затем присваиваем `x` значение `x + 5` — то есть используем старое значение `x`, чтобы получить новое значение. Таким образом, выражение `x = x + 5` по сути означает «увеличить `x` на 5».

В арсенале JavaScript есть более простой способ увеличения или уменьшения переменной на заданную величину: это операторы `+=` и `-=`. Пусть у нас есть переменная `x`, тогда команда `x += 5` означает то же самое, что и `x = x + 5`. Оператор `-=` работает аналогично, то есть `x -= 9` соответствует `x = x - 9` (уменьшить `x` на 9). С помощью этих операторов можно, например, управлять подсчетом очков в игре:

---

**Score** — счет

```
var score = 10;
score += 7;
17
score -= 3;
14
```

---

В этом примере мы сначала присваиваем переменной `score` (счет игры) начальное количество очков (10). Потом, победив монстра, мы увеличиваем счет на 7 очков с помощью оператора `+=` (`score += 7` соответствует `score = score + 7`). Поскольку изначально в `score` было число 10, а  $10 + 7 = 17$ , этой командой мы установили счет в 17 очков.

После победы над монстром мы столкнулись с метеоритом, и счет уменьшился на 3 очка. Опять же, `score -= 3` это то же самое, что и `score = score - 3`. Поскольку перед этим в `score` было 17, `score - 3` равняется 14; это число и будет новым значением `score`.

### ПОПРОБУЙТЕ!

Есть и другие операторы, похожие на `+=` и `-=`. Например, `*=` и `/=`. Как вы думаете, для чего они? Опробуйте их в деле с воздушными шариками:

---

**Balloon** —  
воздушный  
шар

```
var balloons = 100;
balloons *= 2;
???
```

---

Что делает команда `balloons *= 2`? А теперь попробуйте такой код:

---

```
var balloons = 100;
balloons /= 4;
???
```

---

Что делает команда `balloons /= 4`?

## Строки

До сих пор мы имели дело только с числами. Пора познакомиться с еще одним типом данных — со *строками*. В JavaScript (как и в большинстве других языков программирования) строка является набором символов — букв, цифр, знаков пунктуации и пробелов. Чтобы JavaScript знал, где начинается и заканчивается строка, ее берут в кавычки. Вот классический пример с фразой «Привет, мир!»:

```
"Привет, мир!";
"Привет, мир!"
```

Чтобы создать строку, поставьте знак двойной кавычки ("), затем введите какой-нибудь текст и закройте строку еще одной двойной кавычкой. Можно пользоваться и одинарными кавычками ('), однако, чтобы не путаться, все строки в этой книге будут в двойных кавычках.

Строки можно хранить в переменных, так же как числа:

```
var myAwesomeString = "Что-то ОЧЕНЬ крутое!!!";
```

My awesome  
string —  
Моя крутая  
строка

Также ничто не мешает присвоить строковое значение переменной, где раньше хранилось число:

```
var myThing = 5;
myThing = "это строка";
"это строка"
```

My thing —  
моя штука

А что если записать в кавычках число? Страна это будет или число? В JavaScript строка остается строкой, даже если там хранятся цифровые символы. Например:

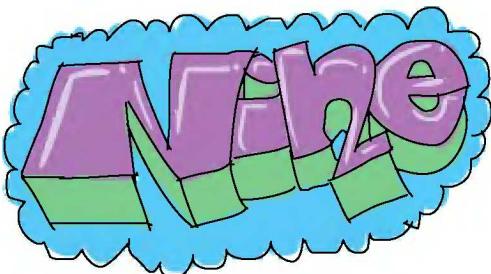
```
var numberNine = 9;
var stringNine = "9";
```

Number nine —  
номер девять

String nine —  
строка девять

В переменной `numberNine` (число девять) хранится число, а в переменной `stringNine` (строка девять) — строка. Чтобы выяснить, в чем их различие, посмотрим, как они реагируют на сложение:

```
numberNine + numberNine;  
18  
stringNine + stringNine;  
"99"
```



Сложив числовые значения 9 и 9, мы получили 18. Однако при использовании оператора + со строками "9" и "9" эти строки просто склеиваются воедино, образуя "99".

## Объединение строк

Как мы только что убедились, оператор + можно использовать и со строками, однако действует он при этом совсем иначе, чем с числами. С помощью оператора + строки можно объединять: результатом будет новая строка, состоящая из первой строки, к концу которой присоединена вторая:

Greeting —  
приветствие

My name —  
мое имя

```
var greeting = "Привет";  
var myName = "Ник";  
greeting + myName;  
"ПриветНик"
```

Здесь мы создали две переменные (`greeting` и `myName`) и присвоили каждой из них строковое значение ("Привет" и "Ник" соответственно). При сложении этих переменных строки объединяются, образуя новую строку — "ПриветНик".

Впрочем, не все тут идеально — между "Привет" и "Ник" должен стоять пробел и запятая. JavaScript не ставит пробелов по собственной инициативе, зато его можно попросить его об этом, добавив пробел к одной из первоначальных строк:

```
❶ var greeting = "Привет, ";  
var myName = "Ник";  
greeting + myName;  
"Привет, Ник"
```

Дополнительный пробел перед закрывающей кавычкой в строке ❶ дает пробел в середине результирующей строки.

Помимо их объединения, со строками можно выполнять множество разных действий. Вот несколько примеров.

## Как узнать длину строки

Чтобы узнать длину строки, достаточно добавить к ее концу `.length`:

**Length** — длина

```
"Суперпупердлиннаястрока".length;  
23
```

Можно добавлять `.length` к концу как самой строки, так и переменной, содержащей строку:

```
var java = "Java";  
java.length;  
4  
var script = "Script";  
script.length;  
6  
var javascript = java + script;  
javascript.length;  
10
```

Здесь мы присвоили строковое значение "Java" переменной `java`, а значение "Script" — переменной `script`. Затем мы добавили `.length` к концу каждой из переменных, узнав таким образом длины отдельных строк, а также длину составленной из них новой строки.

Обратите внимание: я говорил «можно добавлять `.length` к концу как самой строки, так и переменной, содержащей строку». Это касается очень важного свойства переменных: в любом месте программы, где допустимо использовать число или строку, можно также использовать переменную, в которой хранится число или строка.

## Получение отдельного символа строки

Иногда требуется получить из строки одиничный символ. Например, вы можете зашифровать в наборе слов тайное послание, состоящее из вторых символов каждого слова. Тогда, чтобы узнать это послание, нужно получить все вторые символы и объединить их в новую строку.

Чтобы получить символ, стоящий в определенной позиции строки, используйте квадратные скобки — `[]`. Возьмите строку (или переменную, в которой хранится строка) и поставьте сразу после нее квадратные скобки, в которых указана позиция нужного символа. Например, чтобы получить первый символ строковой переменной `myName`, используйте запись `myName[0]`:

```
var myName = "Ник";
myName[0];
"Н"
myName[1];
"и"
myName[2];
"к"
```

Обратите внимание — чтобы получить первый символ, мы указали в скобках позицию 0, а не 1. Дело в том, что JavaScript (как и многие другие языки программирования) ведет отсчет символов с нуля. Таким образом, для получения первого символа строки указывайте позицию 0, второго — 1 и т. д.

Попробуем разгадать наш тайный шифр, где во вторых буквах некоторого набора слов скрыто послание. Вот как это сделать:

**Code word** —  
кодовое  
слово

```
var codeWord1 = "обернись";
var codeWord2 = "неужели";
var codeWord3 = "огурцы";
var codeWord4 = "липкие";
var codeWord5 = "?!";
codeWord1[1] + codeWord2[1] + codeWord3[1] + codeWord4[1] + ←
codeWord5[1];
"беги!"
```

**!** Страницы этой книги недостаточно широки, чтобы напечатать выражение, в котором мы объединяем буквы, одной строкой. Знаками ← помечены места, где код пришлось перенести на следующую строку. Однако вы, вводя этот код в компьютер, можете напечатать его без переносов.

И снова обращаю внимание — второй символ каждой строки мы получаем, указав позицию 1.

**Slice** — часть

## Получение среза строки

Чтобы получить часть, или «срез», строки, используйте `slice`. Например, представьте, что вам нужен отрывок из длинного описания фильма для анонса на вашем сайте. Чтобы воспользоваться `slice`, поставьте в конце строки (или переменной, содержащей строку) точку, а после нее слово `slice` и круглые скобки. В скобках укажите позицию первого символа той части строки, которую вы хотите получить, затем запятую, а затем позицию последнего символа. На рис. 2.2 показано, как использовать `slice`.



Рис. 2.2. Использование `slice` для получения среза строки

Например:

```
var longString = "Эта длинная строка такая длинная";
longString.slice(4, 18);
"длинная строка"
```

Long string —  
длинная  
строка

Первое число в скобках — позиция символа, с которого начинается срез, а второе число — позиция символа, который следует за последним символом среза. На рис. 2.3 показано, каким символам соответствуют эти значения. Начальная (4) и конечная (18) позиции выделены синим цветом.

Э т а   д л и н н а я   с т р о к а   т а к а я   д л и н н а я  
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

Рис. 2.3. В нашем примере `slice` возвращает символы, обведенные серой рамкой

По сути, мы попросили JavaScript: «Вырежи из этой длинной строки часть, которая начинается с символа в позиции 4 и продолжается до позиции 18».

Если указать в скобках после `slice` только одно число, мы получим строку-срез, которая начинается сданной позиции и длится до конца строки:

```
var longString = "Эта длинная строка такая длинная";
longString.slice(4);
"длинная строка такая длинная"
```

## Перевод строки в заглавный или строчный регистр

Если нужно вывести какой-нибудь текст заглавными буквами, воспользуйтесь `toUpperCase`.

To upper  
case —  
в верхний  
регистр

```
"Эй, как дела?".toUpperCase();
"ЭЙ, КАК ДЕЛА?"
```

`.toUpperCase()` возвращает новую строку, все буквы в которой — заглавные.

To lower case —  
в нижний  
регистр

Можно произвести и обратную операцию, используя `toLowerCase`:

---

```
"ЭЙ, КАК ДЕЛА?".toLowerCase();  
"эй, как дела?"
```

---

`.toLowerCase()` делает все символы строчными. Но ведь по правилам предложение должно начинаться с заглавной буквы? Как сделать первый символ строки заглавным, а остальные — строчными?



*Попробуйте сами разобраться, как с помощью только что изученных команд превратить строку "ЭЙ, КАК ДЕЛА?" в "Эй, как дела?". Если ничего не выходит, сверьтесь с разделами, где рассказывается о получении символа строки и использовании slice. Когда закончите, вернитесь к этому месту и сравните свое решение с моим.*

Вот один из вариантов решения:

Silly string —  
буквально  
«глупая  
строка»

Lower string —  
здесь «строка  
в нижнем  
регистре»

First character —  
здесь «первая  
буква»

First character  
upper — здесь  
«первая буква  
в верхнем  
регистре»

Rest of string —  
оставшаяся  
часть строки

---

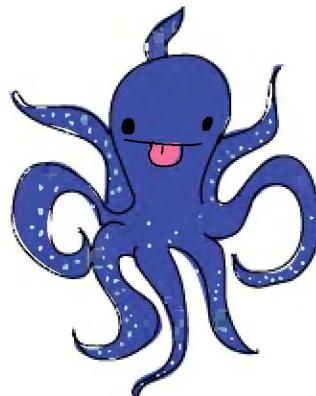
```
❶ var sillyString = "ЭЙ, КАК ДЕЛА?";  
❷ var lowerString = sillyString.toLowerCase();  
❸ var firstCharacter = lowerString[0];  
❹ var firstCharacterUpper = firstCharacter.toUpperCase();  
❺ var restOfString = lowerString.slice(1);  
❻ firstCharacterUpper + restOfString;  
"Эй, как дела?"
```

---

Давайте разберем этот код построчно. В строке ❶ мы создаем новую переменную `sillyString` и кладем в нее строку, которую собираемся изменить. В строке ❷ мы с помощью `.toLowerCase()` получаем версию `sillyString`, где все буквы строчные ("эй, как дела?"), и кладем ее в новую переменную `lowerString`.

В строке ❸ мы с помощью операции `[0]` получаем первый символ `lowerString` (это «э») и сохраняем это значение в переменной `firstCharacter` (напоминаю, позиция 0 соответствует первому символу). Затем в строке ❹ мы переводим `firstCharacter` в верхний регистр и сохраняем в переменной `firstCharacterUpper`.

В строке ❺ мы с помощью `slice` получаем все символы `lowerString`, начиная со второго ("й, как дела?"), и сохраняем их в переменной `restOfString`. И наконец, в строке ❻ мы объединяем



`firstCharacterUpper` («Э») и `restOfString`, что дает нам исковую строку "Эй, как дела?".

Поскольку значения и переменные взаимозаменяемы, можно заменить строки со ② по ⑥ единственной строкой:

---

```
var sillyString = "эй, как дела?";
sillyString[0].toUpperCase() + sillyString.slice(1). ←
toLowerCase();
"Эй, как дела?"
```

---

Однако понять такой код сложнее, так что имеет смысл решать сложные задачи вроде этой пошагово, с помощью переменных — по крайней мере до тех пор, пока вы не привыкнете читать сложный код.

## Булевые значения

Теперь поговорим о булевых значениях. В сущности, есть лишь два варианта таких значений — это либо `true` (истина), либо `false` (ложь). Например, вот простое выражение с булевым значением:

---

```
var javascriptIsCool = true;
javascriptIsCool;
true
```

---

Javascript  
is cool —  
Javascript —  
это круто

Здесь мы создали новую переменную с именем `javascriptIsCool` и присвоили ей булево значение `true`. Следующей строкой мы запросяли содержимое `javascriptIsCool` и, разумеется, получили `true`.

## Логические операции

Подобно тому как числа можно объединять с помощью математических операторов (+, -, \*, / и других), булевые значения можно объединять посредством булевых (логических) операторов. Результатом выражения, составленного из булевых значений и булевых операторов, всегда будет другое булево значение (либо `true`, либо `false`).

Три основных булевых оператора — это `&&`, `||` и `!`. Выглядят они странновато, однако после небольшой практики пользоваться ими будет несложно. Давайте познакомимся с ними поближе.

### `&& (И)`

Оператор `&&` означает «и». Вслух его называют «и», либо «и-и», либо «ампер-санд-амперсанд» (амперсандом называется символ `&`). Используйте оператор `&&` с двумя булевыми значениями, когда нужно узнать, равны ли они *оба* `true`.

Например, перед тем как пойти в школу, вы хотите убедиться, что приняли душ, *а также* взяли рюкзак. Если оба эти условия истинны (`true`), можно идти в школу, но если хоть одно ложно (`false`), вы еще не готовы.

**Had shower** —  
принял душ

**Has backpack** —  
есть рюкзак

```
var hadShower = true;  
var hasBackpack = false;  
hadShower && hasBackpack;  
false
```



Здесь мы устанавливаем переменную `hadShower` («вы приняли душ?») в `true`, а переменную `hasBackpack` («вы взяли рюкзак?») в `false`. Далее, вводя `hadShower && hasBackpack`, мы спрашиваем JavaScript: «равны ли оба этих значения `true`»? Поскольку это не так (рюкзак не в руках), JavaScript возвращает `false` (то есть вы не готовы идти в школу).

Давайте повторим попытку, установив на этот раз обе переменные в `true`:

```
var hadShower = true;  
var hasBackpack = true;  
hadShower && hasBackpack;  
true
```

Теперь JavaScript сообщает нам, что `hadShower && hasBackpack` равняется `true`. Можно идти в школу!

### || (ИЛИ)

Булев оператор `||` означает «или». Так его и следует называть — «или», или даже «или-или», хотя некоторые называют его «пайп», поскольку среди англоязычных программистов символ `|` зовется «пайп» («труба»). Используйте оператор `||` с двумя булевыми значениями для проверки, что *как минимум одно* из них равняется `true`.

Предположим, вы снова готовитесь идти в школу и хотите взять с собой к обеду фрукты, причем вам неважно, будет это яблоко, или апельсин, или и то и другое. С помощью JavaScript можно проверить, есть ли у вас хотя бы один из этих плодов:

**Has apple** —  
есть яблоко

**Has orange** —  
есть апельсин

```
var hasApple = true;  
var hasOrange = false;  
hasApple || hasOrange;  
true
```

Выражение `hasApple || hasOrange` даст `true`, если либо `hasApple` («взяли яблоко?»), либо `hasOrange` («взяли апельсин?»), либо обе эти переменные имеют значение `true`. Однако если *обе* они равны `false`, выражение даст `false` (то есть у вас с собой нет ни одного фрукта).

## ! (НЕ)

Оператор `!` означает «не» — так его и называйте. Используйте этот оператор, чтобы превратить `false` в `true` или, наоборот, `true` в `false`. Это полезно для работы со значениями-противоположностями. Например:

---

```
var isWeekend = true;
var needToShowerToday = !isWeekend;
needToShowerToday;
false
```

---

`isWeekend` — выходной  
`Need to shower today` — нужно принять душ сегодня

В этом примере мы установили переменную `isWeekend` («сейчас выходной?») в `true`. Затем мы дали переменной `needToShowerToday` («сегодня нужно принять душ?») значение `!isWeekend`. Оператор `!` преобразует значение в противоположное — то есть, если `isWeekend` равно `true`, `!isWeekend` даст нам не `true` (то есть `false`). Соответственно, запрашивая значение `needToShowerToday`, мы получаем `false` (сегодня выходной, так что мыться совсем не обязательно).

Поскольку `needToShowerToday` равно `false`, `!needToShowerToday` даст `true`:

---

```
needToShowerToday;
false
!needToShowerToday;
true
```

---

Иными словами, то, что вам не обязательно принимать сегодня душ, — истина (`true`).

## Совмещение логических операторов

Операторы дают больше возможностей, если использовать их совместно. Допустим, вам нужно идти в школу, если сегодня не выходной, и вы приняли душ, и у вас с собой есть яблоко или апельсин. Вот как с помощью JavaScript проверить, выполняются ли все эти условия:

---

```
var isWeekend = false;
var hadShower = true;
var hasApple = false;
```

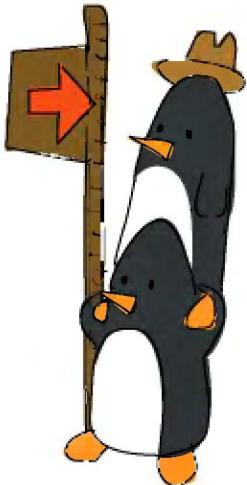
Should go  
to school —  
нужно идти  
в школу

```
var hasOrange = true;  
var shouldGoToSchool = !isWeekend && hadShower && (hasApple || ←  
hasOrange);  
shouldGoToSchool;  
true
```

В данном случае сегодня не выходной, вы приняли душ, у вас нет с собой яблока, зато есть апельсин — значит, нужно идти в школу.

Выражение `hasApple || hasOrange` записано в скобках, поскольку нам важно убедиться, что эта проверка выполнена в первую очередь. Точно так же как JavaScript выполняет умножение прежде сложения, в логических выражениях он выполняет `&&` прежде `||`.

## Сравнение чисел с помощью булевых значений



Булевые значения можно использовать для проверки чисел, если эта проверка подразумевает простой ответ: да или нет. Например, представьте, что вы работаете в парке развлечений, где один из аттракционов имеет ограничение: туда допускаются посетители ростом не менее 150 см (иначе они могут вывалиться из кабинки!). Когда кто-нибудь хочет прокатиться, он сообщает свой рост, и вам нужно понять, больше названное число или меньше.

### Больше

Чтобы узнать, больше ли одно число, чем другое, нужно использовать оператор «больше» (`>`). Например, для проверки, что рост посетителя (155 см) больше, чем ограничение по росту (150 см), мы можем задать переменной `height` (рост посетителя) значение 155, а переменной `heightRestriction` (ограничение по росту) значение 150, а затем использовать оператор `>` для сравнения двух переменных:

Height —  
высота  
Height  
restriction —  
ограничение  
по росту

```
var height = 155;  
var heightRestriction = 150;  
height > heightRestriction;  
true
```

Введя `height > heightRestriction`, мы просим JavaScript показать нам, больше ли первое значение, чем второе, или нет. В данном случае посетитель достаточно высок!

Но что если рост посетителя в точности равен 150 см?

```
var height = 150;  
var heightRestriction = 150;  
height > heightRestriction;  
false
```

Нет, посетитель недостаточно высок! Хотя если ограничение по росту — 150 см, наверное, стоит пускать и тех, чей рост в точности равен 150 см? Это нужно исправить. К счастью, в JavaScript есть еще один оператор, `>=`, что означает «больше или равно».

---

```
var height = 150;
var heightRestriction = 150;
height >= heightRestriction;
true
```

---

Ну вот, теперь лучше — 150 удовлетворяет условию «больше или равно 150».

### Меньше

Оператор, противоположный «больше» (`>`), зовется оператором «меньше» (`<`). Он пригодится, если аттракцион предназначен только для маленьких детей. Например, пусть рост посетителя равен 150 см, но по правилам аттракциона на него допускаются посетители ростом не более 120 см:

---

```
var height = 150;
var heightRestriction = 120;
height < heightRestriction;
false
```

---

Мы хотим убедиться, что рост посетителя *меньше* ограничения, и поэтому используем `<`. Поскольку 150 не меньше 120, ответом будет `false` (человек ростом 150 см слишком высок для этого аттракциона).

И, как вы, наверное, уже догадались, есть оператор `<=`, что означает «меньше или равно».

---

```
var height = 120;
var heightRestriction = 120;
height <= heightRestriction;
true
```

---

Посетителю, рост которого равен 120 см, вход все еще разрешен.



### Равно

Чтобы проверить два числа на точное равенство, используйте тройной знак равенства (`==`) — это оператор «равно». Будьте осторожны, не путайте `==` с одиночным знаком равенства (`=`), поскольку `==` означает «равны ли эти два числа?», а `=` означает «положить значение справа в переменную слева». Иначе говоря, `==` задает вопрос, а `=` присваивает переменной значение.

**Чико, Харпо и Граучо** — псевдонимы троих из братьев Маркса, американских звезд немой комедии.

**My secret number** — мое загаданное число

**Chico guess** — догадка Чико (Харпо, Граучо)

При использовании `=` имя переменной должно стоять слева, а значение, которое вы хотите в эту переменную положить, справа. Однако `==` служит лишь для проверки двух значений на равенство, поэтому неважно, какое значение с какой стороны стоит.

Представьте, что вы загадали своим друзьям Чико, Харпо и Граучо число, а именно число 5. Вы облегчили им задачу, сообщив, что это число от 1 до 9, и ваши друзья начали угадывать. Сначала присвоим переменной `mySecretNumber` значение 5. Первый из играющих, Чико, загадывает ответ 3, который мы кладем в переменную `chicoGuess`. Поглядим, что будет дальше:

```
var mySecretNumber = 5;
var chicoGuess = 3;
mySecretNumber === chicoGuess;
false
var harpoGuess = 7;
mySecretNumber === harpoGuess;
false
var grouchoGuess = 5;
mySecretNumber === grouchoGuess;
true
```

Число, которое вы загадали, находится в переменной `mySecretNumber`. Переменные `chicoGuess`, `harpoGuess` и `grouchoGuess` соответствуют предположениям ваших друзей. Далее с помощью оператора `==` можно проверить, равен ли какой-нибудь ответ вашему числу. Третий друг, Граучо, назвал 5 и победил.

Сравнивая два числа с помощью `==`, вы получаете `true`, только когда оба числа совпадают. Поскольку в `grouchoGuess` находится значение 5, а `mySecretNumber` также равно 5, выражение `mySecretNumber == grouchoGuess` вернет `true`. Другие варианты ответа не совпадают с `mySecretNumber`, поэтому сравнение с ними даст `false`.

Также с помощью `==` можно сравнить две строки или два булевых значения. Если же сравнивать так значения разных типов, ответом всегда будет `false`.

### Двойной знак равенства

Еще немного запутаю вас: в JavaScript есть еще один оператор сравнения (двойное равно, `==`), который означает «практически равно». Используйте его для проверки двух значений на соответствие друг другу, даже если одно из них строка, а другое — число. Все значения принадлежат к тому или иному типу, так что число 5 отличается от строки «5», хоть они и выглядят похоже. Если сравнить их с помощью `==`, JavaScript ответит, что значения не равны. Однако при сравнении через `==` они окажутся равными:

```
var stringNumber = "5";
var actualNumber = 5;
stringNumber === actualNumber;
false
stringNumber == actualNumber;
true
```

String number —  
здесь  
«строка-число»

Actual number —  
число

Возможно, тут вы подумаете: «Похоже, двойное равно удобнее, чем тройное!» Однако будьте очень осторожны: двойное равно можетвести вас в заблуждение. Например, как считаете, 0 равен false? А строка "false" значению false? При сравнении через двойное равно 0 оказывается равным false, а строка "false" не равна false:

```
0 == false;
true
"false" == false;
false
```

Дело в том, что, сравнивая значения через двойное равно, JavaScript первым делом пытается преобразовать их к одному типу. В данном случае булево значение он преобразует в числовое — при этом false становится нулем, а true — единицей. Поэтому, сравнивая `0 == false`, вы получите `true`!

Из-за всех этих странностей лучше пока пользуйтесь только оператором `==`.

### ПОПРОБУЙТЕ!

Вас попросили написать JavaScript-код для автоматической системы управления кинотеатром. Задача состоит в том, чтобы определить, пускать ли зрителя на фильм «с 12 лет и старше» или нет.

Правила таковы: если посетителю 12 лет или больше, он может проходить. Если ему еще не исполнилось 12, но его сопровождает взрослый, пусть тоже проходит. Во всех остальных случаях вход запрещен.



```
var age = 11;
var accompanied = true;
???
```

Accompanied —  
в сопровождении

Допишите этот код, чтобы он определял, можно ли 11-летнему посетителю посмотреть фильм (возраст задается в переменной `age`, а переменная `accompanied` равна `true`, если посетитель пришел со взрослым). Попробуйте поменять эти значения (например, пусть в `age` будет число 12, а в `accompanied` — `true`) и убедитесь, что код по-прежнему находит верное решение.

## undefined и null

И наконец, в JavaScript есть два особых значения, они называются `undefined` и `null`. Оба они означают «пусто», но смысл этого в обоих случаях немного различается.

JavaScript использует значение `undefined`, когда не может найти иного значения. Например, если, создав новую переменную, вы не присвоите ей значение с помощью оператора `=`, ее значением будет `undefined`:

**My variable** —  
моя переменная

```
var myVariable;  
myVariable;  
undefined
```

А значение `null` обычно используется, чтобы явно обозначить — «тут пусто».

**My null variable** —  
моя пустая  
переменная

```
var myNullVariable = null;  
myNullVariable;  
null
```

Пока вы будете нечасто использовать `undefined` и `null`. Вы получите `undefined`, если создадите переменную и не присвоите ей значения, — JavaScript всегда возвращает `undefined`, когда значение не определено. Однако специально `undefined` обычно ничему не присваивают; если вам захочется обозначить, что в переменной «пусто», используйте для этого `null`.

Иначе говоря, `null` нужен, чтобы явно показать отсутствие значения, и порой это бывает полезно. Например, есть переменная, обозначающая ваш любимый овощ. Если вы терпеть не можете все без исключения овощи, имеет смысл дать переменной «любимый овощ» значение `null`.

Этим вы явно покажете любому, кто увидит ваш код, что у вас нет любимого овоща. Однако если в переменной будет `undefined`, кто-нибудь может подумать, что вы просто еще не приписали ей значения.

## Что мы узнали

Теперь вы знаете все базовые типы данных JavaScript — это числа, строки и булевые значения, — а также специальные значения `null` и `undefined`. Числа нужны для всего, что связано с математикой, строки — для работы с текстом, а булевые значения — для разрешения вопросов, на которые можно ответить «да» или «нет». Значения же `null` и `undefined` дают нам способ обозначать то, чего не существует.

В следующих двух главах мы поговорим о массивах и объектах — и то и другое представляет собой способ объединения простых типов данных в более сложные наборы значений.

# 3

## МАССИВЫ

Мы уже изучили числа и строки — типы данных, которые можно хранить и использовать в своих программах. Но одни лишь числа и строки — это как-то скучновато; не столь уж многое можно сделать со строкой как таковой. С помощью массивов JavaScript позволяет создавать и группировать данные более любопытными способами. А по сути своей массив — всего лишь список, где хранятся другие значения.

Например, если вашему другу интересно, какие три вида динозавров вам нравятся больше всего, вы можете создать массив и расположить там по порядку названия этих динозавров:

```
var myTopThreeDinosaurs = ["Тираннозавр", "Велоцираптор", ↵  
    "Стегозавр"];
```

My top three  
dinosaurs —  
три моих  
любимых  
динозавра

Теперь вместо того, чтобы показывать своему другу три отдельные строки, вы можете воспользоваться единственным массивом myTopThreeDinosaurs.

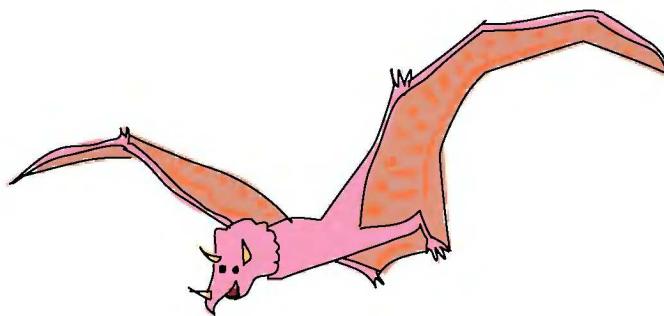
### Зачем нужны массивы?

Вернемся к нашим динозаврам. Положим, вы решили написать программу для учета всех видов динозавров, которые вам известны. Вы можете создать для каждого вида отдельную переменную:

Dinosaur —  
динозавр

```
var dinosaur1 = "Тираннозавр";
var dinosaur2 = "Велоцираптор";
var dinosaur3 = "Стегозавр";
var dinosaur4 = "Трицератопс";
var dinosaur5 = "Брахиозавр";
var dinosaur6 = "Птеранодон";
var dinosaur7 = "Апатозавр";
var dinosaur8 = "Диплодок";
var dinosaur9 = "Компсогнат";
```

Однако пользоваться этим списком не слишком удобно — у вас есть девять переменных там, где можно обойтись лишь одной. А теперь представьте, что динозавров в программе не девять, а 1000! Пришлось бы создать 1000 отдельных переменных, работать с которыми было бы решительно невозможно.



Это похоже на список покупок, составленный так, что каждая покупка указана на отдельном листе бумаги. На одном листке написано «яйца», на другом — «хлеб», на следующем — «апельсины». Большинство людей предпочли бы видеть весь список на одном листе бумаги. Так не проще ли сгруппировать всех динозавров в один список?

Вот для этого и нужны массивы.

### Создание массива

Чтобы создать массив, используйте квадратные скобки []. Фактически для задания пустого массива достаточно лишь пары квадратных скобок:

```
[];  
[]
```

Но кому нужен пустой массив? Давайте-ка заполним его динозаврами!

Чтобы создать массив со значениями, нужно перечислить эти значения внутри квадратных скобок, разделяя их запятыми. Отдельные значения, хранящиеся в массиве, называют *элементами*. В данном примере все элементы будут строковыми (это названия любимых динозавров), поэтому запишем их в кавычках. Сохраним наш массив в переменной с именем `dinosaurs`:

---

```
var dinosaurs = ["Тираннозавр", "Велоцираптор", "Стегозавр", ←  
"Трицератопс", "Брахиозавр", "Птеранодон", "Апатозавр", ←  
"Диплодок", "Компсогнат"];
```

---

Длинный список сложно читать, когда он записан одной строкой, но, к счастью, это не единственный способ форматирования кода при создании массива. Вы можете поставить открывающую квадратную скобку на одной строке, каждый элемент писать с новой строки и последней строкой поставить закрывающую квадратную скобку:

---

```
var dinosaurs = [  
    "Тираннозавр",  
    "Велоцираптор",  
    "Стегозавр",  
    "Трицератопс",  
    "Брахиозавр",  
    "Птеранодон",  
    "Апатозавр",  
    "Диплодок",  
    "Компсогнат"  
];
```

---

Чтобы ввести такой код в консоли, вам придется нажимать одновременно с `ENTER` клавишу `SHIFT` каждый раз, когда нужно перейти к новой строке. Иначе JavaScript попытается выполнить то, что вы уже ввели, даже если команда еще не завершена. Поэтому при работе в консоли проще записывать массивы одной строкой.

Для JavaScript неважно, отформатируете вы код, расположив весь массив на одной строке или на нескольких строках по частям. Сколько бы ни стояло переносов, JavaScript увидит один и тот же массив — в нашем случае состоящий из девяти строк.

## Доступ к элементам массива

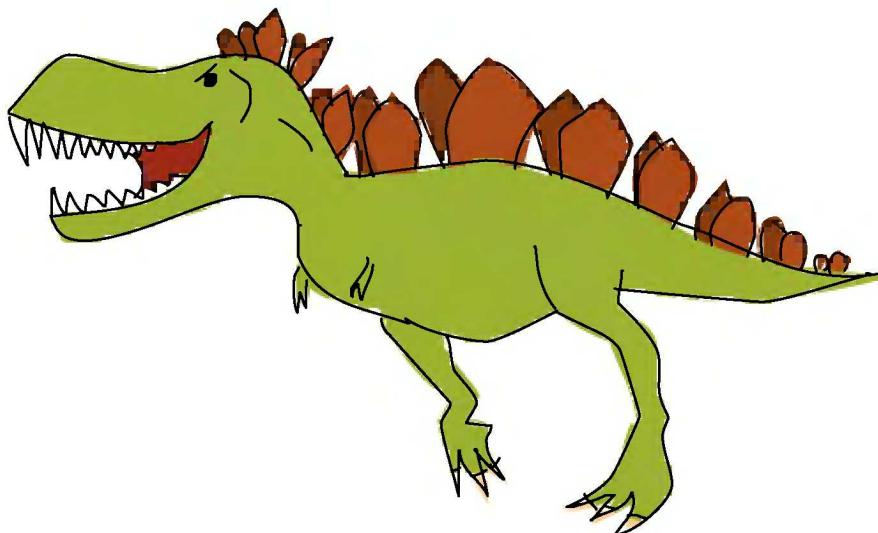
Чтобы получить доступ к элементам массива, используйте квадратные скобки с индексом нужного вам элемента, как в этом примере:

```
dinosaurs[0];  
"Тираннозавр"  
dinosaurs[3];  
"Трицератопс"
```

Индекс — это номер элемента, в котором хранится значение. Аналогично символам в строке, первому элементу массива соответствует индекс 0, второму — 1, третьему — 2 и т. д. Поэтому, запросив индекс 0 в массиве `dinosaurs`, мы получили "Тираннозавр" (это первый элемент), а запросив индекс 3 — "Трицератопс" (четвертый элемент).

Возможность доступа к отдельным элементам массива очень полезна. Например, если вы хотите показать кому-то самого-самого любимого своего динозавра, ни к чему показывать весь массив. Вместо этого просто возьмите первый элемент:

```
dinosaurs[0];  
"Тираннозавр"
```



## Создание и изменение элементов

Используя индекс в квадратных скобках, можно задавать или изменять значения элементов и даже добавлять новые элементы. Например, чтобы заменить содержимое первого элемента массива `dinosaurs` ("Тираннозавр") на "Тираннозавр рекс", можно написать:

```
dinosaurs[0] = "Тираннозавр рекс";
```

После этого массив `dinosaurs` станет таким:

```
["Тираннозавр рекс", "Велоцираптор", "Стегозавр", "Трицератопс", ←  
"Брахиозавр", "Птеранодон", "Апатозавр", "Диплодок", ←  
"Компсогнат"]
```

С помощью индексов также можно добавлять в массив элементы. Например, вот как создать массив `dinosaurs`, задавая каждый элемент через квадратные скобки:

```
var dinosaurs = [];  
dinosaurs[0] = "Тираннозавр";  
dinosaurs[1] = "Велоцираптор";  
dinosaurs[2] = "Стегозавр";  
dinosaurs[3] = "Трицератопс";  
dinosaurs[4] = "Брахиозавр";  
dinosaurs[5] = "Птеранодон";  
dinosaurs[6] = "Апатозавр";  
dinosaurs[7] = "Диплодок";  
dinosaurs[8] = "Компсогнат";  
  
dinosaurs;  
["Тираннозавр", "Велоцираптор", "Стегозавр", "Трицератопс", ←  
"Брахиозавр", "Птеранодон", "Апатозавр", "Диплодок", ←  
"Компсогнат"]
```

Сначала создаем пустой массив: `var dinosaurs = []`. Затем в каждой из следующих строк добавляем по одному элементу командами `dinosaurs[]` с индексом от 0 до 8. Закончив наполнение массива, можно посмотреть его содержимое (набрав `dinosaurs;`) и убедиться, что JavaScript расположил значения по порядку, в соответствии с индексами.

На самом деле в массив можно добавить элемент с любым индексом. Например, чтобы добавить нового (выдуманного) динозавра с индексом 33, введем:

```
dinosaurs[33] = "Филосораптор";  
  
dinosaurs;  
["Тираннозавр", "Велоцираптор", "Стегозавр", "Трицератопс", ←  
"Брахиозавр", "Птеранодон", "Апатозавр", "Диплодок", ←  
"Компсогнат", undefined × 24 "Филосораптор"]
```

Элементы между индексами 8 и 33 получат значение undefined. При печати массива Chrome сообщает количество этих undefined-элементов, а не выводит каждый из них по отдельности.

## Разные типы данных в одном массиве

Не обязательно, чтобы все элементы массива были одного типа. Например, вот массив, в котором хранится число (3), строка ("динозавры"), массив ([*"трицератопс"*, *"стегозавр"*, 3627.5]) и еще одно число (10):

# Dinosaurs and numbers — динозавры и числа

Чтобы обратиться к элементам массива, вложенного в другой массив, нужно использовать вторую пару квадратных скобок. Например, если команда `dinosaursAndNumbers[2]`; вернет весь вложенный массив, то `dinosaursAndNumbers[2][0]`; — лишь первый элемент этого вложенного массива ("тицератопс").

```
dinosaursAndNumbers[2];
["трицератопс", "стегозавр", 3627.5];
dinosaursAndNumbers[2][0];
"трицератопс"
```



Рис. 3.1. Индексы основного массива показаны красным цветом, а индексы вложенного массива — синим

Вводя `dinosaursAndNumbers[2][0]`, мы просим JavaScript обратиться к индексу 2 массива `dinosaursAndNumbers`, где находится массив `["трицератопс", "стегозавр", 3627.5]`, и вернуть значение с индексом 0 из этого вложенного массива — это первый элемент, `"трицератопс"`. На рис. 3.1 показаны индексы для этих массивов.

## Работаем с массивами

Работать с массивами вам помогут *свойства* и *методы*. Свойства хранят различные сведения о массиве, а методы обычно либо изменяют его, либо возвращают новый массив. Давайте разберемся.

### Длина массива

Порой нужно знать, сколько в массиве элементов. Например, если снова и снова добавлять динозавров в массив `dinosaurs`, вы можете забыть, сколько их теперь всего.

Для этого есть свойство `length` (длина), хранящее количество элементов в массиве. Чтобы узнать длину массива, просто добавьте `.length` после его имени. Давайте посмотрим, как это работает. Но сначала создадим новый массив с тремя элементами:

```
var maniacs = ["Якко", "Вакко", "Дот"];
maniacs[0];
"Якко"
maniacs[1];
"Вакко"
maniacs[2];
"Дот"
```

Чтобы узнать длину этого массива, добавим `.length` к `maniacs`:

```
maniacs.length;
3
```

JavaScript сообщает, что в массиве 3 элемента, и мы знаем, что их индексы — 0, 1 и 2. Отсюда следует полезное наблюдение: последний индекс массива всегда на единицу меньше длины этого массива. Это значит, что есть простой способ получить последний элемент массива, какой бы ни была его длина:

```
maniacs[maniacs.length - 1];
"Дот"
```

Якко, Вакко и Дот — герои мультсериала «Озорные анимашки» (англ. *Animaniacs*) о семействе Уорнер, созданного компанией Warner Bros.

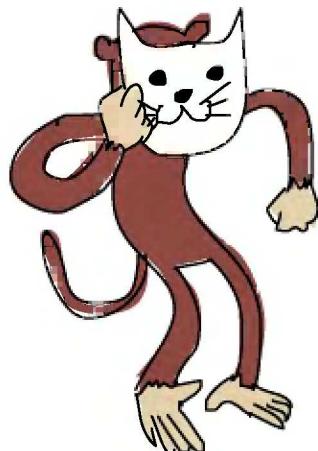
Мы попросили JavaScript вернуть элемент из нашего массива, но вместо числового индекса ввели в квадратных скобках выражение: длина массива минус 1. JavaScript нашел свойство `maniacs.length` со значением 3, вычел 1, получив 2, и наконец вернул элемент с индексом 2 — это и есть последний элемент, "Дот".

## Добавление элементов в массив

**Push** —  
буквально  
«протолкнуть,  
добавить»

**Animals** —  
животные

```
var animals = [];
animals.push("Кот");
①
animals.push("Пес");
②
animals.push("Лама");
③
animals;
["Кот", "Пес", "Лама"]
animals.length;
④
```



Командой `var animals = []`; мы создали пустой массив `animals`, а затем методом `push` добавили туда элемент "Кот". Потом снова использовали `push`, добавив "Пес", а затем "Лама". Запросив теперь содержимое массива `animals`, мы видим, что "Кот", "Пес" и "Лама" стоят там в том же порядке, в каком мы их добавляли.

Запуск метода в программировании называется *вызовом метода*. При вызове метода `push` происходят две вещи. Во-первых, в массив добавляется элемент, указанный в скобках. Во-вторых, метод задает новую длину массива. Именно эти значения длины появляются в консоли после каждого вызова `push`.

Чтобы добавить элемент в начало массива, используйте метод `.unshift(элемент)`:

```
animals;
["Кот", "Пес", "Лама"]
① animals[0];
"Кот"
animals.unshift("Мартышка");
④
animals;
```

```

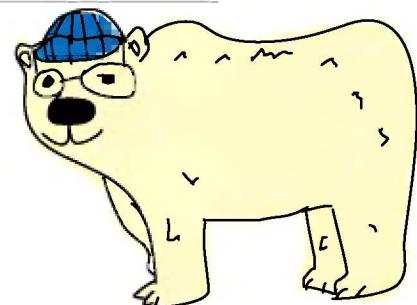
["Мартышка", "Кот", "Пес", "Лама"]
animals.unshift("Белый медведь");
5
animals;
["Белый медведь", "Мартышка", "Кот", "Пес", "Лама"]
animals[0];
"Белый медведь"
❷ animals[2];
"Кот"

```

---

Мы начали с массива, созданного раньше, — ["Кот", "Пес", "Лама"]. Затем добавили в его начало элементы "Мартышка" и "Белый медведь", отчего остальные элементы сдвинулись вперед — при каждом добавлении их индексы увеличивались на 1. В результате элемент "Кот", у которого раньше был индекс 0 ❶, оказался под индексом 2 ❷.

Как и `push`, метод `unshift` при каждом вызове задает новую длину массива.



## Удаление элементов массива

Убрать из массива последний элемент можно, добавив к его имени `.pop()`. Метод `pop` делает сразу два дела: удаляет последний элемент из массива и возвращает этот элемент в виде значения. Для примера начнем с нашего массива `animals` ["Белый медведь", "Мартышка", "Кот", "Пес", "Лама"]. Далее создадим новую переменную `lastAnimal` и сохраним в ней последний элемент, вызвав `animals.pop()`.

**Pop** —  
буквально  
«выдавить»

**Last animal** —  
последнее  
животное

```

animals;
["Белый медведь", "Мартышка", "Кот", "Пес", "Лама"]
❶ var lastAnimal = animals.pop();
lastAnimal;
"Лама"
animals;
["Белый медведь", "Мартышка", "Кот", "Пес"]
❷ animals.pop();
"Пес"
animals;
["Белый медведь", "Мартышка", "Кот"]
❸ animals.unshift(lastAnimal);
4
animals;
["Лама", "Белый медведь", "Мартышка", "Кот"]

```

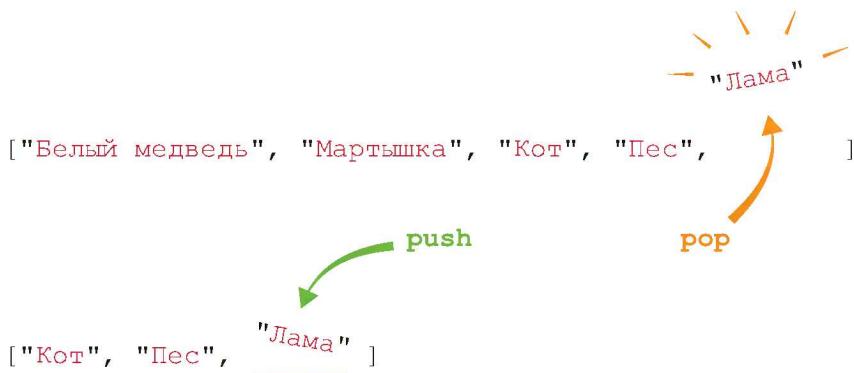
---

При вызове `animals.pop()` в строке ❶ последний элемент массива `animals`, "Лама", был возвращен и сохранен в переменной `lastAnimal`.

Кроме того, элемент "Лама" был удален из массива, в котором после этого осталось четыре элемента. При следующем вызове `animals.pop()` в строке ❷ был удален из массива и возвращен элемент "Пес", а элементов в массиве осталось всего три.

Вызвав `animals.pop()` для элемента "Пес", мы не сохранили это значение в переменной, и оно пропало. С другой стороны, элемент "Лама" был сохранен в переменной `lastAnimal`, чтобы при случае им можно было снова воспользоваться. В строке ❸ мы с помощью `unshift(lastAnimal)` добавили "Лама" обратно, в начало массива. В итоге получился массив ["Лама", "Белый медведь", "Мартышка", "Кот"].

Методы `push` и `pop` хорошо друг друга дополняют, поскольку порой нужно работать только с концом массива. Вы можете добавить элемент в конец вызовом `push`, а потом, когда это понадобится, забрать его оттуда вызовом `pop`. Мы рассмотрим это на примере чуть позже в этой главе.

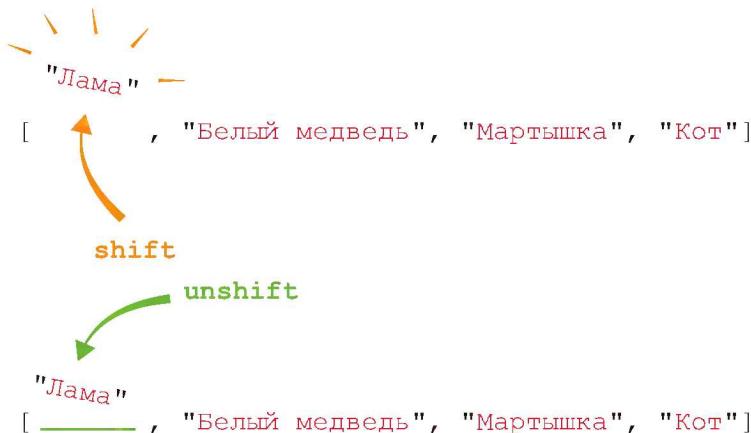


Чтобы удалить из массива первый элемент, вернув его значение, используйте `.shift()`:

```
First animal —  
первое  
животное  
animals;  
["Лама", "Белый медведь", "Мартышка", "Кот"]  
var firstAnimal = animals.shift();  
firstAnimal;  
"Лама"  
animals;  
["Белый медведь", "Мартышка", "Кот"]
```

Метод `animals.shift()` работает аналогично `animals.pop()`, но элемент берется из начала массива. В начале этого примера массив `animals` имел вид ["Лама", "Белый медведь", "Мартышка", "Кот"]. Вызов `.shift()` вернул первый элемент, "Лама", который мы сохранили в переменной `firstAnimal`. Поскольку `.shift()` не только возвращает элемент, но и удаляет его, в массиве `animals` осталось лишь ["Белый медведь", "Мартышка", "Кот"].

Методы `unshift` и `shift` добавляют и удаляют элементы с начала массива — также как `push` и `pop` добавляют и удаляют элементы с конца.



## Объединение массивов

Чтобы «склеить» два массива, создав таким образом новый массив, используйте команду `firstArray.concat(otherArray)`. Метод `concat` создает массив, в котором элементы из `firstArray` будут расположены перед элементами из `otherArray`.

Пускай у нас есть два списка — список пушистых животных и список чешуйчатых животных — и мы хотим их объединить. Если поместить наших пушистых животных в массив `furryAnimals`, а чешуйчатых — в массив `scalyAnimals`, команда `furryAnimals.concat(scalyAnimals)` создаст новый массив, в начале которого будут элементы из первого массива, а в конце — из второго.



**First Array** —  
первый массив  
**Other Array** —  
другой массив

**Furry animals** —  
пушистые  
животные

**Scaly animals** —  
чешуйчатые  
животные

---

```
var furryAnimals = ["Альпака", "Кольцехвостый лемур", "Йети"];
var scalyAnimals = ["Удав", "Годзилла"];
var furryAndScalyAnimals = furryAnimals.concat(scalyAnimals);
furryAndScalyAnimals;
["Альпака", "Кольцехвостый лемур", "Йети", "Удав", "Годзилла"]
furryAnimals;
["Альпака", "Кольцехвостый лемур", "Йети"]
scalyAnimals;
["Удав", "Годзилла"]
```

---

**Furry and scaly  
animals** —  
пушистые  
и чешуйчатые  
животные

Хоть команда `firstArray.concat(otherArray)` и возвращает массив, содержащий все элементы из `firstArray` и `otherArray`, сами эти массивы остаются прежними. Запросив содержимое `furryAnimals` и `scalyAnimals`, мы видим, что массивы не изменились.

### Объединение нескольких массивов

С помощью `concat` можно объединить больше чем два массива. Для этого укажите дополнительные массивы в скобках, разделив их запятыми:

**Feathered Animals** —  
животные с перьями

**All animals** —  
все животные

```
var furryAnimals = ["Альпака", "Кольцехвостый лемур", "Йети"];
var scalyAnimals = ["Удав", "Годзилла"];
var featheredAnimals = ["Ара", "Додо"];
var allAnimals = furryAnimals.concat(scalyAnimals, ←
featheredAnimals);
allAnimals;
["Альпака", "Кольцехвостый лемур", "Йети", "Удав", "Годзилла",
"Ара", "Додо"]
```

Мы видим, что пернатые животные из массива `featheredAnimals` оказались в самом конце нового массива, поскольку `featheredAnimals` был указан последним в скобках метода `concat`.

Метод `concat` удобен, когда нужно объединить несколько массивов в один. Скажем, пусть у вас есть список любимых книг и у вашего друга тоже есть свой список, и вы решили выяснить, можно ли купить все эти книги в магазине по соседству. Тогда будет проще, если у вас будет не два списка, а один. Просто объедините ваш список со списком вашего друга методом `concat`, и пожалуйста — у вас один список на двоих!

**Color** — цвет

### Поиск индекса элемента в массиве

Чтобы выяснить, какой у определенного элемента индекс в массиве, используйте `.indexOf("элемент")`. Создадим массив `colors` с названиями цветов, а затем получим индексы элементов «синий» и «зеленый» с помощью команд `colors.indexOf("синий")` и `colors.indexOf("зеленый")`. Поскольку «синий» располагается по индексу 2, `colors.indexOf("синий")` вернет 2. А «зеленый» находится по индексу 1, так что `colors.indexOf("зеленый")` вернет 1.

```
var colors = ["красный", "зеленый", "синий"];
colors.indexOf("синий");
2
colors.indexOf("зеленый");
1
```

Метод `indexOf` похож на квадратные скобки, только здесь все наоборот: команда `colors[2]` вернет "синий", а `colors.indexOf("синий")` вернет 2.

```
colors[2];
"синий"
colors.indexOf("синий");
2
```

Хотя элемент "синий" стоит третьим по порядку, его индекс равен 2, ведь мы всегда считаем с 0. Разумеется, то же относится к "зеленым" с индексом 1.

Если элемента, индекс которого вы запрашиваете, в массиве нет, JavaScript вернет значение `-1`.

```
colors.indexOf("фиолетовый");
-1
```

Таким образом JavaScript сообщает: «элемент не найден», так или иначе возвращая из метода число.

Если элемент встречается в массиве больше чем один раз, `indexOf` вернет индекс того элемента, который находится ближе к началу массива.

```
var insects = ["Пчела", "Муравей", "Пчела", "Пчела", "Муравей"];
insects.indexOf("Пчела");
0
```

Insects —  
насекомые

## Превращаем массив в строку

Воспользовавшись методом `.join()`, можно соединить все элементы массива в одну большую строку.

Join —  
соединить

```
var boringAnimals = ["Мартышка", "Кот", "Рыба", "Ящерица"];
boringAnimals.join();
"Мартышка, Кот, Рыба, Ящерица"
```

Boring  
animals —  
скучные  
животные

Метод `join` возвращает строку, в которой через запятую перечислены все элементы массива `boringAnimals`. Но что если мы не хотим использовать в качестве разделителя запятую?

Нам поможет метод `.join ("разделитель")`, который делает все то же самое, но вместо запятых ставит между элементами выбранный



разделитель. Давайте попробуем три разных разделителя: дефис с пробелами по сторонам, звездочку \* и союз «и» с пробелами по сторонам. Обратите внимание: разделитель нужно записывать в кавычках — ведь это строка.

```
var boringAnimals = ["Мартышка", "Кот", "Рыба", "Ящерица"];
boringAnimals.join(" - ");
"Мартышка - Кот - Рыба - Ящерица"
boringAnimals.join("*")
"Мартышка*Кот*Рыба*Ящерица"
boringAnimals.join(" и ")
"Мартышка и Кот и Рыба и Ящерица"
```

**Среднее имя** — имя, обычно расположеноное между личным именем и фамилией. Используется как элемент полного имени, в основном в Европе и западных странах.

Этот вариант `join` удобен, когда у вас есть массив, из которого нужно сделать строку. Предположим, у вас много средних имен и вы решили хранить их все в массиве вместе со своим личным именем и фамилией. И вдруг кому-то понадобилось ваше полное имя в виде строки. Тогда метод `join` с разделителем-пробелом преобразует все имена в искомую строку:

```
var myNames = ["Николас", "Эндрю", "Максвелл", "Морган"];
myNames.join(" ");
"Николас Эндрю Максвелл Морган"
```

Не будь метода `join`, пришлось бы соединять элементы вручную, что очень утомительно:

```
myNames[0] + " " + myNames[1] + " " + myNames[2] + " " + myNames[3];
"Николас Эндрю Максвелл Морган"
```

Кроме того, этот код сработает, только если у вас ровно два средних имени. Если же их три или одно, программу придется менять. А с `join` ничего менять не надо — этот метод соединит в строку все элементы массива независимо от его длины.

Если же в массиве хранятся нестроковые значения, JavaScript преобразует их в строки перед тем, как соединить:

```
var ages = [11, 14, 79];
ages.join(" ");
"11 14 79"
```

**Ages** — возрасты

## Что полезного можно сделать с массивами

Теперь вы умеете разными способами создавать массивы и знаете немало действий с ними. Но как все это может вам пригодиться в жизни? В этом разделе мы разберем несколько коротких программ, посвященных практическому использованию массивов.

### Поиск дороги домой

Представьте, что ваша подруга побывала у вас в гостях, а теперь хочет показать вам свой дом. Но вот незадача — вы никогда не бывали у нее раньше, а путь назад вам предстоит проделать в одиночку.

К счастью, вам в голову приходит хитрый способ решения этой проблемы: по дороге к дому подруги вы будете записывать возможные ориентиры (телефонную будку, вывеску магазина или аптеки, школу и т. д.). А по дороге назад, двигаясь по списку с конца, вычеркивать каждый встреченный ориентир — так вы всегда будете знать, куда идти дальше.

#### Построение массива с помощью push

Давайте напишем код для выполнения этих действий. Начнем с создания массива — пустого, поскольку, пока вы еще не отправились в гости, неизвестно, какие ориентиры вам повстречаются. Затем, по дороге к дому вашей подруги, мы будем добавлять описание каждого ориентира в массив с помощью `push`. И наконец, когда придет время идти домой, будем методом `pop` изымать каждый пройденный ориентир из массива.

---

```
var landmarks = [];
landmarks.push("Мой дом");
landmarks.push("Дорожка к дому");
landmarks.push("Мигающий фонарь");
landmarks.push("Протекающий гидрант");
landmarks.push("Пожарная станция");
landmarks.push("Приют для кошек");
landmarks.push("Моя бывшая школа");
landmarks.push("Дом подруги");
```

---



Здесь мы создали пустой массив `landmarks` и методом `push` сохранили в нем все ориентиры, замеченные по дороге к дому подруги.

**Landmarks** —  
заметные  
объекты

#### Двигемся в обратном порядке с помощью `pop`

Вы добрались до дома подруги, и можно изучить массив ориентиров. Разумеется, первым стоит "Мой дом", потом "Дорожка к дому", и т. д. до конца массива, где находится элемент "Дом подруги". Теперь, когда наступит время идти домой, вам останется лишь изымать из массива по одному элементу, и всегда будет понятно, куда идти дальше.

```
landmarks.pop();
"Дом подруги"
landmarks.pop();
"Моя бывшая школа"
landmarks.pop();
"Приют для кошек"
landmarks.pop();
"Пожарная станция"
landmarks.pop();
"Протекающий гидрант"
landmarks.pop();
"Мигающий фонарь"
landmarks.pop();
"Дорожка к дому"
landmarks.pop();
"Мой дом"
```



Вот вы и дома!

Заметили, что первый ориентир, который вы поместили в массив методом `push`, оказался также последним, который вы извлекли методом `pop`? А последний добавленный ориентир оказался первым извлеченным? Может показаться, что лучше бы первый добавленный элемент и извлекался всегда первым, однако извлекать элементы в обратном порядке в некоторых случаях удобно.

Такой подход нередко используется в больших программах — именно поэтому `push` и `pop` в JavaScript всегда под рукой.



Среди программистов такой способ работы с элементами называется «стек». Представьте, что стек — это стопка блинов. Всякий раз, когда готов новы́й блин, его кладут сверху стопки (как метод `push`), и, когда вы берете блин, чтобы его съесть, вы тоже берете его сверху (как метод `pop`). Снятие элементов со стека похоже на путешествие назад во времени: последним изымается элемент, который был в стеке первым. То же происходит с блинами: последний блин, который вы съедите, — это первый, который был приготовлен. На сленге программистов этот способ называется «последним вошел, первым вышел», английская аббревиатура LIFO (*last in, first out*). Есть и альтернативный подход — «первым вошел, первым вышел», аббревиатура FIFO (*first in, first out*). Его также называют очередью, поскольку таким же образом устроены очереди — первый человек, вставший в очередь, будет первым, кого обслугуют.

## Случайный выбор

Используя массивы, можно написать программу, которая выдает случайные варианты из заданного списка (наподобие «шара судьбы»). Однако сначала нужно разобраться, откуда нам брать случайные числа.

### Использование Math.random()

Случайные числа можно генерировать с помощью специального метода `Math.random()`, который при каждом вызове возвращает случайное число от 0 до 1:

**Math random** —  
случайное  
число

---

```
Math.random();  
0.8945409457664937  
Math.random();  
0.3697543195448816  
Math.random();  
0.48314980138093233
```

---

Важно помнить, что `Math.random()` всегда возвращает число меньше 1, то есть никогда не возвращает собственно 1.

Если вам нужно число побольше, просто умножьте полученное из метода `Math.random()` значение на подходящий коэффициент. Например, если нужно случайное число от 0 до 10, умножьте `Math.random()` на 10:

---

```
Math.random() * 10;  
7.648027329705656  
Math.random() * 10;  
9.7565904534421861  
Math.random() * 10;  
0.21483442978933454
```

---

### Округление с помощью Math.floor()

И все же эти случайные значения нельзя использовать как индексы в массиве, поскольку индексы должны быть целыми числами, а не десятичными дробями. Чтобы исправить этот недостаток, нужен метод `Math.floor()`, округляющий число до ближайшего снизу целого значения (по сути, он просто отбрасывает все знаки после запятой).

**Math floor** —  
целая часть  
числа

---

```
Math.floor(3.7463463);  
3  
Math.floor(9.9999);  
9  
Math.floor(0.793423451963426);  
0
```

---

Давайте используем оба метода, чтобы получить случайный индекс. Нужно лишь умножить `Math.random()` на длину массива и затем округлить полученное число методом `Math.floor()`. Например, если в массиве четыре элемента, это можно сделать так:

```
Math.floor(Math.random() * 4);  
// может выпасть 0, 1, 2 или 3
```

При каждом запуске этот код будет возвращать случайное число от 0 до 3 (включая 0 и 3). Поскольку `Math.random()` всегда возвращает значение меньше 1, `Math.random() * 4` никогда не вернет 4 или большее число.

Используя это случайное число как индекс, можно получить случайный элемент массива:

**Random words** — случайные слова

```
var randomWords = ["Взрыв", "Пещера", "Принцесса", "Карандаш"];  
var randomIndex = Math.floor(Math.random() * 4);  
randomWords[randomIndex];  
"Пещера"
```

**Random index** —  
случайный  
индекс

С помощью `Math.floor(Math.random() * 4);` мы получили случайное число от 0 до 3. Сохранив это число в переменной `randomIndex`, мы использовали его как индекс для получения строки из массива `randomWords`.

В сущности, можно сделать этот код короче, избавившись от переменной `randomIndex`:

```
randomWords[Math.floor(Math.random() * 4)];  
"Принцесса"
```

### Программа случайного выбора вариантов

Теперь давайте создадим массив с фразами, чтобы случайным образом выбирать их с помощью написанного ранее кода. Это и будет наш компьютерный «шар судьбы»! В комментариях указаны примеры вопросов, которые можно задать нашей программе.

**Phrases** —  
фразы

```
var phrases = [  
    "Звучит неплохо",  
    "Да, это определенно надо сделать",  
    "Не думай, что это хорошая идея",  
    "Может, не сегодня?",  
    "Компьютер говорит нет"  
];  
// Мне выпить еще молочного коктейля?
```

```
phrases[Math.floor(Math.random() * 5)];  
"Не думаю, что это хорошая идея"  
// Мне пора делать домашнюю работу?  
phrases[Math.floor(Math.random() * 5)];  
"Может, не сегодня?"
```

Мы создали массив `phrases`, в котором хранятся различные советы. Теперь, придумав вопрос, можно запросить случайный элемент из массива `phrases`, и полученный совет поможет принять решение!

Обратите внимание: поскольку в массиве с советами пять элементов, мы умножаем `Math.random()` на 5. Таким образом, мы всегда получим одно из пяти значений индекса: 0, 1, 2, 3 или 4.

## Генератор случайных дразнилок

Можно усовершенствовать код выбора вариантов, создав программу, которая при каждом запуске генерирует случайную дразнилку!

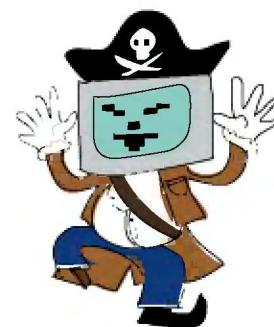
```
var randomBodyParts = ["глаз", "нос", "череп"];  
var randomAdjectives = ["вонючая", "унылая", "дурацкая"];  
var randomWords = ["муха", "выпра", "дубина", "мартьшка", "крыса"];  
// Выбор случайной части тела из массива randomBodyParts:  
① var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];  
// Выбор случайного прилагательного из массива randomAdjectives:  
② var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];  
// Выбор случайного слова из массива randomWords:  
③ var randomWord = randomWords[Math.floor(Math.random() * 5)];  
// Соединяем случайные строки в предложение:  
var randomInsult = "У тебя " + randomBodyPart + " словно " + ←  
randomAdjective + " " + randomWord + "!!!";  
randomInsult;  
"У тебя нос словно дурацкая выпра!!!"
```

У нас есть три массива со словами, и в строках **①**, **②** и **③** мы с помощью трех индексов берем из каждого массива по случайному слову. Затем мы склеиваем их, помещая результат в переменную `randomInsult`, — это и есть готовая дразнилка. В строках **①** и **②** мы используем множитель 3, поскольку и в `randomAdjective`, и в `randomBodyPart` по три элемента. Аналогично в строке **③** мы умножаем на 5, ведь в `randomWords` пять элементов. Обратите внимание, что мы добавили между `randomAdjective` и `randomWord`

**Random body part** —  
случайная часть тела

**Random adjective** —  
случайное прилагательное

**Random insult** —  
случайная дразнилка



строку с единственным пробелом. Запустите этот код несколько раз — при каждом запуске получится новая случайная дразнилка!

### ПОПРОБУЙТЕ!

Если хотите сделать все действительно по-умному, замените строку ❸ на вот такую:

```
var randomWord = randomWords[Math.floor(Math.random() * ↵
randomWords.length)];
```

В этой программе всегда нужно умножать `Math.random()` на длину массива; использование `randomWords.length` как множителя означает, что нам не понадобится менять этот код, если длина массива изменится.

Вот еще один способ составления случайной дразнилки:

```
var randomInsult = ["У тебя", randomBodyPart, "словно", ↵
randomAdjective, randomWord + "!!!"].join(" ");
"У тебя череп словно дурацкая дубина!!!"
```

В этом варианте каждое слово дразнилки — это отдельный элемент массива, и мы соединяем все эти элементы методом `join` с разделителем-пробелом. Лишь в одном случае пробел не нужен — между `randomWord` и «!!!». В этом случае мы используем оператор `+`, чтобы соединить строки без пробела.

### Что мы узнали

Как мы теперь знаем, массивы JavaScript предназначены для хранения списка значений. Мы научились создавать массивы и работать с ними и освоили много способов доступа к их элементам.

Массивы JavaScript — один из способов хранения множества значений в одном месте. В следующей главе мы познакомимся с объектами — другим способом объединения значений в единую сущность. Для доступа к элементам объектов используются *строковые ключи*, а не индексы.

## УПРАЖНЕНИЯ

Чтобы укрепить знания, полученные в этой главе, выполните эти упражнения.

### #1. Новые дразнилки

Сделайте генератор случайных дразнилок со своим набором слов.

### #2. Изощренные дразнилки

Усовершенствуйте генератор дразнилок, чтобы он создавал дразнилки такого типа: «У тебя [часть тела] еще более [прилагательное], чем [часть тела животного] у [животное]».

Подсказка: нужно будет создать еще один массив.

### #3. Оператор + или join?

Сделайте две версии своего генератора дразнилок: одна пусть использует для составления дразнилки оператор +, а другая создает массив со словами и соединяет их через пробел с помощью join. Какой вариант вам больше нравится и почему?

### #4. Соединение чисел

Как с помощью метода join превратить массив [3, 2, 1] в строку "3 больше, чем 2 больше, чем 1"?

# 4

## ОБЪЕКТЫ

Объекты JavaScript очень похожи на массивы, но для доступа к элементам объектов используются строки, а не числа. Эти строки называют *ключами*, или *свойствами*, а элементы, которые им соответствуют, — *значениями*. Вместе эти фрагменты информации образуют пары «ключ-значение». Причем если массивы используются главным образом как списки, хранящие множество элементов, то объекты часто применяют как одиночные сущности с множеством характеристик, или *атрибутов*. Например, в третьей главе мы создали несколько массивов, хранящих названия разных животных. Но что если нужно хранить набор различных сведений об одном конкретном животном?

### Создание объектов

Для хранения всевозможной информации об одном животном подойдет JavaScript-объект. Вот пример объекта, где хранятся сведения о трехногой кошке по имени Гармония.

Cat — кошка  
Legs — ноги  
Color — цвет, окрас

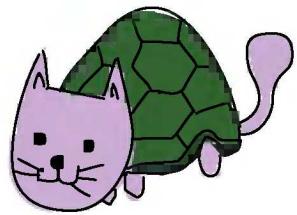
---

```
var cat = {  
    "legs": 3,  
    "name": "Гармония",  
    "color": "Черепаховый"  
};
```

---

Мы создали переменную под названием `cat` и присвоили ей объект с тремя парами «ключ-значение» (лапы, имя, окрас). При создании объекта используются фигурные скобки `{}` вместо квадратных, к которым

мы привыкли, создавая массивы. Внутри фигурных скобок можно вводить пары «ключ-значение», а вместе скобки и пары значений называются **литералом объекта**. Литерал объекта — это быстрый способ создания объекта вместе с его содержимым.



! Мы уже встречались с литералами массивов (например, `["a", "б", "в"]`), числовыми литералами (например, `37`), строковыми литералами (например, `"лось"`) и булевыми литералами (`true` и `false`). Литерал — это когда значение записывается сразу и целиком, а не составляется постепенно, шаг за шагом.

Например, чтобы создать массив с числами от 1 до 3, можно ввести литерал массива `[1, 2, 3]`. Альтернативный способ — создать пустой массив и методом `push` добавить в него значения 1, 2 и 3. Не всегда заранее известно, что за данные будут храниться в массиве или объекте, поэтому создавать массивы и объекты с помощью одних лишь литералов не получится.



Рис. 4.1. Общий синтаксис создания объекта

На рис. 4.1 показан базовый синтаксис создания нового объекта.

При создании объекта ключ записывается перед двоеточием (`:`), а значение — после. Это двоеточие напоминает знак «равно», поскольку значения, стоящие слева, присваиваются именам (ключам), стоящим справа, что похоже на создание переменных со значениями. Все пары «ключ-значение» должны быть разделены запятыми — в нашем примере эти запятые стоят в конце строк. И обратите внимание, что после завершающей пары «ключ-значение» (`"color": "Черепаховый"`) запятую ставить не нужно — следом за этой парой ставится закрывающая фигурная скобка.

## Ключи без кавычек

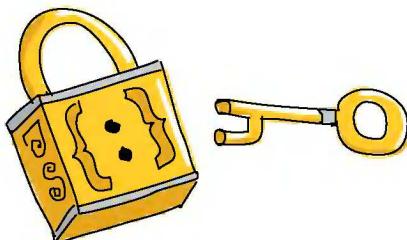
Создавая первый объект, мы писали имена ключей в кавычках, однако это не обязательно. Следующая запись тоже является допустимым литералом объекта:

```
var cat = {  
    legs: 3,  
    name: "Гармония",  
    color: "Черепаховый"  
};
```

JavaScript знает, что ключи всегда строковые, поэтому можно обходиться без кавычек. В этом случае имена ключей должны соответствовать тем же правилам, что и имена переменных: например, в них не должно быть пробелов. Но если ключ указан в кавычках, пробелы в его имени допустимы:

```
var cat = {  
    legs: 3,  
    "full name": "Гармония Филомена Уси-Пусечка Морган",  
    color: "Черепаховый"  
};
```

Full name —  
полное имя



Помните, что, хотя ключ всегда является строковым (в кавычках он записан или без), значение, соответствующее этому ключу, может быть любого типа — даже переменной, в которой хранятся данные.

Кроме того, весь объект можно записать одной строкой, хотя читать такую программу будет, пожалуй, не слишком удобно:

```
var cat = { legs: 3, name: "Гармония", color: "Черепаховый" };
```

## Доступ к значениям внутри объектов

Хранящиеся в объектах значения можно получить с помощью квадратных скобок — так же, как элементы массива. Единственное различие в том, что вместо индекса (число) используется ключ (строка).

```
cat["name"];  
"Гармония"
```

Точно так же, как необязательны кавычки при записи литерала объекта, их можно опускать и при доступе к значениям по ключу. Однако в этом случае код будет немного другим:

---

```
cat.name;  
"Гармония"
```

---

Такую запись называют *точечной нотацией*. Вместо того чтобы писать имя ключа в кавычках внутри квадратных скобок, мы просто ставим точку, после которой пишем имя ключа, без кавычек. И, аналогично ключам без кавычек при записи литерала, такой прием сработает, только если ключ не содержит специальных символов — например, пробелов.

Теперь предположим, что вы хотите узнать, какие вообще ключи есть у данного объекта. Для этого в JavaScript есть удобное средство — команда `Object.keys()`:

---

```
var dog = { name: "Оладушек", age: 6, color: "белый", ↪  
bark: "Гав тяф тяф!" };  
var cat = { name: "Гармония", age: 8, color: "черепаховый" };  
Object.keys(dog);  
["name", "age", "color", "bark"]  
Object.keys(cat);  
["name", "age", "color"]
```

---

`Object.keys(anyObject)` возвращает массив, содержащий все ключи объекта `anyObject`.

## Добавление элементов объекта

Пустой объект похож на пустой массив, только вместо квадратных скобок при его создании используются фигурные:

---

```
var object = {};
```

---

Добавлять элементы объекта можно так же, как элементы массива, — но используя строки вместо чисел:

---

```
var cat = {};  
cat["legs"] = 3;  
cat["name"] = "Гармония";  
cat["color"] = "Черепаховый";  
cat;  
{ color: "Черепаховый", legs: 3, name: "Гармония" }
```

---

Мы начали с пустого объекта под названием `cat`, а затем поочередно добавили к нему три пары «ключ-значение». Потом мы ввели `cat`,

**Object key** —  
ключ объекта  
**Dog** — пес  
**Bark** — лай

и браузер отобразил содержимое объекта. Тут надо отметить, что разные браузеры могут показывать объекты по-разному. Например, Chrome (на момент написания этих строк) выводит объект `cat` в консоли следующим образом:

```
Object {legs: 3, name: "Гармония", color: "Черепаховый"}
```

Chrome перечисляет ключи в таком порядке — (`legs`, `name`, `color`), но другие браузеры могут выводить их в другой очередности. Дело в том, что JavaScript хранит ключи объектов, не упорядочивая их.

В массивах элементы расположены строго один за другим: индекс 0 перед индексом 1, индекс 3 после индекса 2; однако в случае объектов неясно, как расположить элементы друг относительно друга. Должен ли ключ `color` стоять перед `legs` или после? «Правильного» ответа на этот вопрос нет, поэтому объекты хранят свои ключи без конкретной очередности, в результате чего разные браузеры показывают ключи в разном порядке. Так что никогда не полагайтесь в своих программах на тот или иной порядок ключей.

## Добавление ключей через точку

Новые ключи также можно добавлять через точечную нотацию. Давайте перепишем этим способом предыдущий пример, то есть создадим пустой объект и заполним его данными:

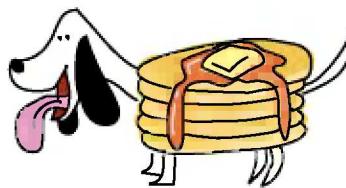
```
var cat = {};
cat.legs = 3;
cat.name = "Гармония";
cat.color = "Черепаховый";
```

Если обратиться к несуществующему свойству объекта, JavaScript вернет специальное значение `undefined`, сообщая таким образом: «здесь ничего нет». Например:

```
var dog = {
  name: "Оладушек",
  legs: 4,
  isAwesome: true
};
dog.isBrown;
undefined
```

**Is awesome** —  
классный  
**Is brown** —  
коричневый

Здесь мы определили три свойства объекта `dog`: `name`, `legs` и `isAwesome`. Свойства `isBrown` среди них нет, поэтому `dog.isBrown` возвращает `undefined`.



## Массивы объектов

До этого момента мы рассматривали только массивы и объекты, в которых содержатся данные простых типов, такие как числа и строки. Однако ничто не мешает сделать элементом массива или объекта другой массив или объект. Например, так может выглядеть массив с объектами, описывающими динозавров:

```
var dinosaurs = [  
  { name: "Тираннозавр рекс", period: "Верхнемеловой" },  
  { name: "Стегозавр", period: "Верхнеюрский" },  
  { name: "Платеозавр", period: "Триасовый" }  
];
```

Period —  
период

Получить сведения о первом динозавре можно уже известным нам способом — указав индекс в квадратных скобках:

```
dinosaurs[0];  
{ name: "Тираннозавр рекс", period: "Верхнемеловой" }
```

А если нужно только название первого динозавра, достаточно указать ключ объекта в еще одних квадратных скобках, следом за индексом:

```
dinosaurs[0]["name"];  
"Тираннозавр рекс"
```

Другой вариант — воспользоваться точечной нотацией:

```
dinosaurs[1].period;  
"Верхнеюрский"
```



Точечную нотацию можно использовать только с объектами, для массивов она не подходит.

Lucky  
numbers —  
счастливые  
числа

```
var anna = { name: "Анна", age: 11, luckyNumbers: [2, 4, 8, 16] };  
var dave = { name: "Дэйв", age: 5, luckyNumbers: [3, 9, 40] };  
var kate = { name: "Кейт", age: 9, luckyNumbers: [1, 2, 3] };
```

Мы создали три объекта, сохранив их в переменных `anna`, `dave` и `kate`. У каждого из этих объектов есть по три свойства: `name`, `age` и `luckyNumbers`. Каждому ключу `name` соответствует строковое значение, ключу `age` — числовое, а ключу `luckyNumbers` — массив, содержащий несколько чисел.

Теперь создадим массив друзей:

Friends —  
друзья

```
var friends = [anna, dave, kate];
```

Итак, в переменной `friends` находится массив с тремя элементами: `anna`, `dave` и `kate` (каждый из них является объектом). Мы можем получить любой из объектов по его индексу в массиве:

Array —  
массив

```
friends[1];  
{ name: "Дэйв", age: 5, luckyNumbers: Array[3] }
```

Здесь мы извлекли из массива второй объект, `dave` (по индексу 1). Вместо массива `luckyNumbers` Chrome напечатал `Array[3]`, что означает «этот массив с тремя элементами» (можно изучить содержимое этого массива с помощью Chrome, см. раздел «Исследование объектов в консоли» на с. 77.) Также мы можем получить значение, хранящееся в объекте, указав индекс объекта в квадратных скобках, поставив точку и написав соответствующий ключ:

```
friends[2].name  
"Кейт"
```

Этот код запрашивает элемент по индексу 2 (что соответствует переменной `kate`), а затем — свойство этого объекта, хранящееся по ключу `"name"` (это `"Кейт"`). Можно даже получить значение из массива, находящегося в объекте, который, в свою очередь, находится в массиве `friends`:

```
friends[0].luckyNumbers[1];  
4
```

Использованные в этом примере индексы показаны на рис. 4.2. `friends[0]` — это элемент по индексу 0 из массива `friends`, то есть объект `anna.friends[0].luckyNumbers` — это массив `[2, 4, 8, 16]` из объекта `anna`. И наконец, `friends[0].luckyNumbers[1]` — это значение по индексу 1 из массива `luckyNumbers` — то есть число 4.

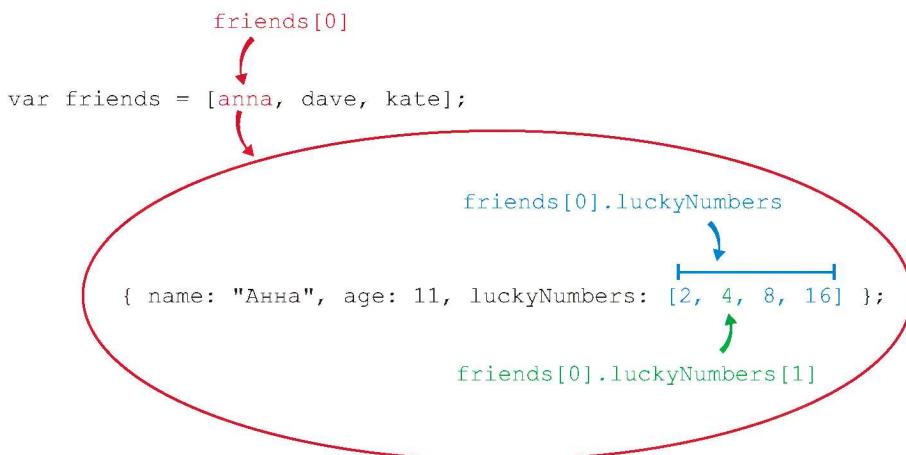


Рис. 4.2. Доступ ко вложенным значениям

## Исследование объектов в консоли

Chrome позволяет изучать содержимое объектов, показанных в консоли. Например, если вы введете

```
friends[1];
```

Chrome отобразит то, что показано на рис. 4.3.

```
friends[1];  
▶ Object {name: "Дэйв", age: 5, luckyNumbers: Array[3]}
```

Рис. 4.3. Отображение объекта в интерпретаторе Chrome

Треугольник слева обозначает, что объект можно раскрыть. Для этого кликните мышкой по объекту, и увидите то, что показано на рис. 4.4.

```
friends[1];
▼ Object {name: "Дэйв", age: 5, luckyNumbers: Array[3]} ⓘ
  age: 5
  ► luckyNumbers: Array[3]
    name: "Дэйв"
  ► __proto__: Object
```

Рис. 4.4. Раскрытие объекта

Массив luckyNumbers также можно раскрыть, кликнув по нему (рис. 4.5).

```
friends[1];
▼ Object {name: "Дэйв", age: 5, luckyNumbers: Array[3]} ⓘ
  age: 5
  ▼ luckyNumbers: Array[3]
    0: 3
    1: 9
    2: 40
    length: 3
    ► __proto__: Array[0]
    name: "Дэйв"
  ► __proto__: Object
```

Length —  
длина

Рис. 4.5. Раскрытие вложенного в объект массива

Не беспокойтесь о свойствах с названием `__proto__`, они относятся к прототипу объекта. Мы поговорим о прототипах позже, в главе 12. Обратите внимание, что помимо элементов массива интерпретатор показывает его свойство `length`.

Также вы можете просмотреть массив `friends` целиком и раскрыть каждый его элемент, как показано на рис. 4.6.

```
friends
[▼ Object ⓘ , ▼ Object ⓘ , ▼ Object ⓘ ]
  age: 11           age: 5           age: 9
  ► luckyNumbers: Array[4]   ► luckyNumbers: Array[3]   ► luckyNumbers: Array[3]
    name: "Анна"       name: "Дэйв"       name: "Кейт"
  ► __proto__: Object     ► __proto__: Object     ► __proto__: Object
```

Рис. 4.6. Все три объекта из массива `friends`, как отображает их интерпретатор Chrome

## Что полезного можно сделать с объектами

Теперь, когда вам известны разные способы создания объектов и добавления к ним свойств, давайте применим эти знания на практике, введя несколько простых программ.

### Учет долгов

Предположим, вы решили открыть банк. Вы одолжили друзьям денег и теперь думаете, как вести учет того, кто и сколько вам должен.

Можно использовать объект как способ связать строку с числом. Строкой в нашем случае будет имя друга, а числом — сумма, которую вам должны:

```
❶ var owedMoney = {};  
❷ owedMoney["Джимми"] = 5;  
❸ owedMoney["Анна"] = 7;  
❹ owedMoney["Джимми"];  
❺  
❻ owedMoney["Элис"];  
undefined
```

Owed money —  
одолженная  
сумма денег

В строке ❶ мы создали пустой массив owedMoney. В строке ❷ мы присвоили ключу "Джимми" значение 5, а в строке ❸ присвоили значение 7 ключу "Анна". В строке ❹, запросив значение, связанное с ключом "Джимми", мы получили 5. Затем в строке ❺, пытаясь узнать значение, связанное с ключом "Элис", мы получили ответ undefined, поскольку такой ключ не задан.

Теперь представим, что Джимми занял у вас еще немного денег (скажем, 3 доллара). Пора обновить данные в нашем объекте, добавив 3 к долгу Джимми — используем для этого оператор «плюс равно» (+=), речь о котором шла во второй главе.

```
owedMoney["Джимми"] += 3;  
owedMoney["Джимми"];  
8
```



Это примерно то же самое, что и `owedMoney["Джимми"] = owedMoney["Джимми"] + 3`. Также можно посмотреть на объект целиком, чтобы выяснить, сколько денег задолжал каждый из друзей:

owedMoney:  
{ Джимми: 8, Анна: 7 }



## Хранение информации о фильмах

Предположим, у вас большая коллекция кино на DVD и Blu-ray. Правда было бы здорово хранить информацию об этих фильмах на компьютере, чтобы в случае чего быстро найти сведения о том или ином фильме?

Для этого можно создать объект, каждый ключ в котором — это название фильма, а каждое значение — другой объект, в котором содержится информация об этом фильме. Да, хранящиеся в объекте значения тоже могут быть объектами!

- Movies** — фильмы
- Release date** — дата выхода
- Duration** — продолжительность
- Actors** — в ролях
- Format** — формат

```
var movies = {  
    "В поисках Немо": {  
        releaseDate: 2003,  
        duration: 100,  
        actors: ["Альберт Брукс", "Эллен Дедженерес", "Александр Гуилд"],  
        format: "DVD"  
    },  
    "Звездные войны: Эпизод VI – Возвращение джедая": {  
        releaseDate: 1983,  
        duration: 134,  
        actors: ["Марк Хэмилл", "Харрисон Форд", "Кэрри Фишер"],  
        format: "DVD"  
    },  
    "Гарри Поттер и Кубок огня": {  
        releaseDate: 2005,  
        duration: 157,  
        actors: ["Дэниел Рэдклифф", "Эмма Уотсон", "Руперт Гринт"],  
        format: "Blu-ray"  
    }  
};
```

Наверное, вы заметили, что названия фильмов (ключи внешнего объекта) я поставил в кавычки, но ключи внутренних объектов записал без кавычек. Дело в том, что в названиях нужны пробелы — иначе пришлось бы писать нечто вроде ЗвездныеВойныЭпизодVIВозвращениеДжедая, а это уж совсем нелепо. Для ключей вложенных объектов кавычки необязательны, поэтому я их и не ставил. Код выглядит аккуратнее, когда в нем нет излишних знаков пунктуации.

Теперь, если вы захотите что-то узнать о фильме, это легко сделать:

---

```
var findingNemo = movies["В поисках Немо"];
findingNemo.duration;
100
findingNemo.format;
"DVD"
```

---

Мы сохранили сведения о фильме «В поисках Немо» в переменной `findingNemo`. Теперь достаточно обратиться к свойствам этого объекта (таким как `duration` и `format`), чтобы получить интересующую нас информацию.

Кроме того, в коллекцию легко добавить новые фильмы:

---

```
var cars = {
  releaseDate: 2006,
  duration: 117,
  actors: ["Оуэн Уилсон", "Бонни Хант", "Пол Ньюман"],
  format: "Blu-ray"
};
movies["Тачки"] = cars;
```

---

Здесь мы создали новый объект со сведениями о мультфильме «Тачки» (`Cars`), а затем добавили его в объект `movies` с ключом "Тачки".

Коллекция растет, и вам может понадобиться простой способ просмотреть названия всех своих фильмов. Для этого подойдет `Object.keys`:

---

```
Object.keys(movies);
["В поисках Немо", "Звездные войны: Эпизод VI – Возвращение
джедая", "Гарри Поттер и Кубок огня", "Тачки"]
```

---

## Что мы узнали

Теперь мы знаем, как устроены объекты JavaScript. Они во многом похожи на массивы и тоже нужны для хранения множества элементов данных в одном месте. Но есть важное отличие — для доступа к элементам объекта используются строки, тогда как элементы массива расположены по числовым индексам. Поэтому массивы отсортированы по порядку, а объекты нет.

В дальнейших главах, когда мы больше узнаем о возможностях JavaScript, мы научимся использовать объекты для многих других задач. В следующей главе речь пойдет об HTML — языке разметки веб-страниц.

## УПРАЖНЕНИЯ

Попрактикуйтесь в использовании объектов, выполнив эти упражнения.

### #1. Подсчет очков

Представьте, что вы играете в какую-нибудь игру со своими друзьями и вам нужно вести счет. Создайте для этого объект и назовите его `scores`. Пусть ключами будут имена ваших друзей, а значениями — набранные ими очки (0 или больше). Счет игроков надо будет увеличивать по мере того, как они зарабатывают новые очки. Как вы будете менять счет игрока, хранящийся в объекте `scores`?

### #2. Вглубь объектов и массивов

Пускай у вас есть такой объект:

---

```
My crazy  
object —  
мой нелепый  
объект
```

---

```
Some array —  
какой-то  
массив
```

---

```
Purpose — цель
```

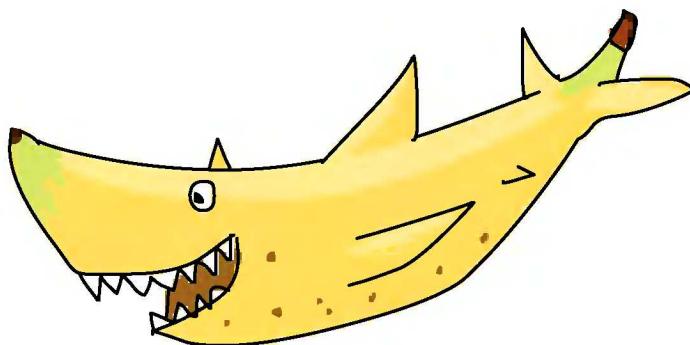
---

```
Random  
animal —  
случайное  
животное
```

```
var myCrazyObject = {  
  "name": "Нелепый объект",  
  "some array": [7, 9, { purpose: "путаница", number: 123 }, 3.3],  
  "random animal": "Банановая акула"  
};
```

---

Как одной строкой JavaScript-кода извлечь из этого объекта число 123? Проверьте свое решение, запустив его в консоли.



# 5

## ОСНОВЫ HTML

Встроенная в браузер JavaScript-консоль, которой мы до сих пор пользовались, хороша, когда нужно протестировать небольшой фрагмент кода, но для создания более масштабных программ понадобится чуть более гибкое и универсальное средство — вроде веб-страницы со встроенным JavaScript-кодом. В этой главе мы как раз и научимся создавать несложные странички на языке HTML.

Гипертекстовый язык разметки *HTML* предназначен специально для создания веб-страниц. Слово *гипертекстовый* означает, что фрагменты текста связаны между собой *гиперссылками* — то есть ссылками в документе на другие объекты. А язык *разметки* — это способ встраивать в текст дополнительную информацию. Разметка указывает программам (таким как браузер), как отображать текст и что с ним делать.

В этой главе я покажу, как создавать HTML-документы в *текстовом редакторе* — программе, предназначенной для работы с простым текстом без форматирования, в отличие от текстовых процессоров вроде Microsoft Word. Документы текстовых процессоров содержат *форматированный текст* (с различными типами и размерами шрифтов, цветами и т. п.), и устроены эти программы так, чтобы форматирование было легко менять. Кроме того, многие текстовые процессоры позволяют вставлять в текст картинки и другие графические элементы.

Простой же текст является только текстом — без цветов, стилей, размеров и т. д. Вставить в такой текст картинку не выйдет, разве только составить ее из символов — скажем, как этого котика справа.

/ \\_ / \  
=( °W° )=  
 ) ( //  
( \_ \_ ) //

## Текстовые редакторы

Мы будем создавать HTML-документы в кросс-платформенном (совместимом с Windows, Mac OS и Linux) редакторе Sublime Text. Скачать Sublime Text можно бесплатно, однако спустя некоторое время вас попросят приобрести лицензию. На случай, если вам такой вариант не по нраву, я отобрал несколько полностью бесплатных альтернатив. Хотя в этой главе я буду ориентироваться на Sublime Text, работа с другими редакторами будет не сильно отличаться — благодаря относительной простоте текстовых редакторов как таковых.

- Gedit — кросс-платформенный текстовый редактор, часть проекта GNOME (<https://wiki.gnome.org/Apps/Gedit/>).
- Для Microsoft Windows хорошей альтернативой будет Notepad++ (<http://notepad-plus-plus.org/>).
- В Mac OS вы можете воспользоваться TextWrangler (<http://www.barebones.com/products/textwrangler/>).

Чтобы установить Sublime Text, зайдите на сайт <http://www.sublimetext.com/>. Инструкции по установке редактора отличаются для каждой из операционных систем, но написаны просто и понятно. В случае каких-либо проблем загляните в раздел Support («Поддержка») на сайте приложения.

## ПОДСВЕТКА СИНТАКСИСА

Sublime Text будет отображать ваши программы в цвете — это называется подсветкой синтаксиса. Смысл в том, что программы легче читать, когда разные конструкции языка выделены разными цветами. Например, строки могут отображаться зеленым цветом, а ключевые слова вроде `var` — оранжевым.

Sublime Text позволяет выбрать одну из множества схем подсветки. В этой книге используется схема IDLE — вы можете включить ее, войдя в меню Preferences → Color Scheme и выбрав там IDLE, чтобы у вас в редакторе программы выглядели так же, как примеры кода в этой главе и далее.

## Наш первый HTML-документ

Установив Sublime Text, запустите его и создайте новый файл, выбрав File → New File. Затем выберите File → Save, чтобы сохранить новый, пустой файл; назовите его `page.html` и сохраните на рабочий стол.

Настало время писать HTML-код. Введите в файл `page.html` следующий текст:

```
<meta charset="UTF-8">
<h1>Привет, мир!</h1>
<p>Моя первая веб-страничка.</p>
```

Сохраните обновленный файл *page.html*, выбрав File → Save. Теперь посмотрим, на что это будет похоже в веб-браузере. Откройте Chrome и, удерживая CTRL, нажмите O (в Mac OS вместо CTRL используйте клавишу COMMAND). В появившемся окне выберите файл *page.html*, находящийся на рабочем столе. То, что вы должны после этого увидеть, изображено на рис. 5.1.



Рис. 5.1. Ваша первая HTML-страница в Chrome

Вы только что создали свой первый HTML-документ! Вы просматриваете его через браузер, однако находится он не в интернете — Chrome открыл его с вашего компьютера и, считав разметку, определил, как нужно отображать текст.

## Теги и элементы

HTML-документы состоят из элементов. Каждый элемент начинается с *открывающего тега* и оканчивается *закрывающим тегом*. Например, в нашем первом документе пока всего два элемента: *h1* и *p* (а также элемент *meta*, но его мы отдельно здесь рассматривать не будем. Он нужен, чтобы в браузере отображался русский текст). Элемент *h1* начинается с открывающего тега *<h1>* и заканчивается закрывающим тегом *</h1>*, а элемент *p* начинается с открывающего тега *<p>* и заканчивается закрывающим тегом *</p>*. Все, что находится между открывающим и закрывающим тегами, называют *содержимым элемента*.

Открывающие теги представляют собой название элемента в угловых скобках: *< и >*. Закрывающие теги выглядят так же, но перед именем элемента в них ставится наклонная черта *(/)*.

## Элементы заголовков

У каждого элемента есть особое назначение и способ применения. Например, элемент `h1` означает «это заголовок верхнего уровня». Содержимое, которое вы введете между открывающим и закрывающим тегами `<h1>`, браузер отобразит на отдельной строке крупным жирным шрифтом.

Всего в HTML шесть уровней заголовков: `h1`, `h2`, `h3`, `h4`, `h5` и `h6`. Выглядят они так:

```
<meta charset="UTF-8">
<h1>Заголовок первого уровня</h1>
<h2>Заголовок второго уровня</h2>
<h3>Заголовок третьего уровня</h3>
<h4>Заголовок четвертого уровня</h4>
<h5>Заголовок пятого уровня</h5>
<h6>Заголовок шестого уровня</h6>
```

На рис. 5.2 показано, как эти заголовки выглядят в браузере.

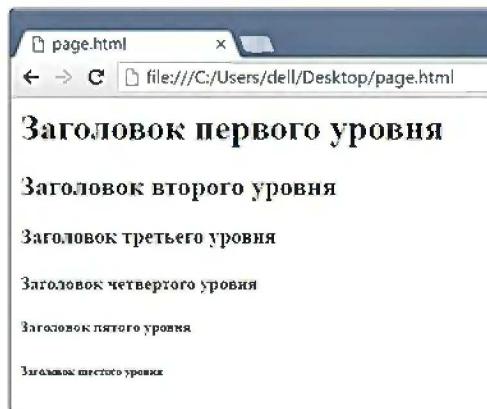


Рис. 5.2. Элементы заголовков разного уровня

## Элемент `p`

Элемент `p` нужен для разделения текста на параграфы. Любой фрагмент текста, который вы поместите между тегами `<p>`, будет отображен как отдельный параграф, с отступами сверху и снизу. Давайте посмотрим, что происходит, если элементов `<p>` несколько. Для этого добавьте новую строку в документ `page.html` (прежние строки показаны серым цветом).

---

```
<meta charset="UTF-8">
<h1>Привет, мир!</h1>
<p>Моя первая веб-страничка.</p>
<p>Добавим-ка еще параграф.</p>
```

---

На рис. 5.3 показана страничка с нашим новым параграфом.

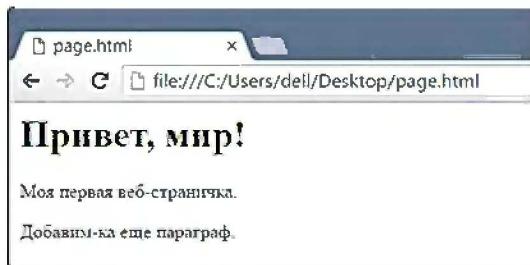


Рис. 5.3. Та же страничка с еще одним параграфом

Обратите внимание, что каждый параграф отображен с новой строки, а между параграфами сделан отступ. Все это благодаря тегу `<p>`.

## Пробелы в HTML и блочные элементы

А как наша страничка будет выглядеть без тегов? Давайте посмотрим:

---

```
<meta charset="UTF-8">
Привет, мир!
Моя первая веб-страничка.
Добавим-ка еще параграф.
```

---

На рис. 5.4 показана страничка без тегов.

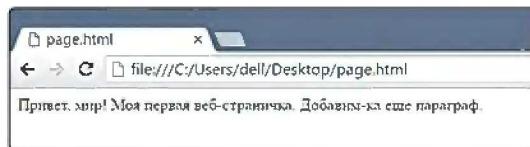


Рис. 5.4. Та же страничка без HTML-тегов

Мало того что пропало форматирование, теперь весь текст отображается в одну строку! Дело в том, что в HTML все *пробельные символы* преобразуются в единственный пробел. Пробельные символы — это любые символы, которые отображаются в браузере как пробелы или



отступы, — например, это пробел, символ табуляции и символ перевода строки (тот самый, который вы вводите, нажимая ENTER или RETURN). Поэтому все пустые строки, которые вы вставите между фрагментами текста в HTML-документе, сожмутся до одного пробела.

Элементы `p` и `h1` — *блочные*; это значит, что их содержимое отображается отдельными блоками текста с новой строки и любое содержимое, идущее после такого блока, тоже начнется с новой строки.

## Строчные элементы

А теперь добавим к нашему документу еще два элемента, `em` и `strong`:

На рис. 5.5 показано, как выглядит страница с новыми тегами.

```
<meta charset="UTF-8">
<h1>Привет, мир!</h1>
<p>Моя <em>первая</em> <strong>веб-страничка</strong>.</p>
<p>Добавим-ка еще <strong><em>параграф</em></strong>.</p>
```

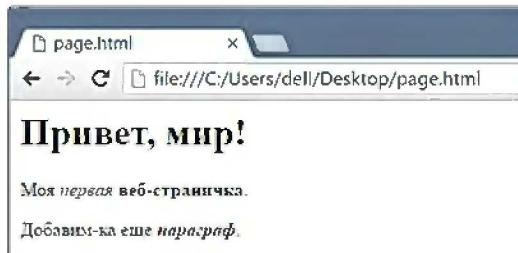


Рис. 5.5. Элементы `em` и `strong`

Элемент `em` отображает свое содержимое курсивом, а элемент `strong` — жирным шрифтом. И `em`, и `strong` относятся к строчным элементам, поскольку они, в отличие от блочных элементов, не выводят свое содержимое отдельной строкой.

Чтобы отобразить текст одновременно жирным шрифтом и курсивом, поместите его внутрь обоих тегов. Обратите внимание, что в последнем примере теги стояли в такой последовательности: `<strong><em>параграф</em></strong>`. Очень важно правильным образом *вкладывать* элементы друг в друга: если один элемент находится внутри другого элемента, то его открывающий тег и его

закрывающий тег также должны находиться внутри этого элемента. Например, такой вариант недопустим:

```
<strong><em>параграф</strong></em>
```

Закрывающий тег `</strong>` расположен здесь перед закрывающим тегом `</em>`. Как правило, браузеры никак не сообщают о подобных ошибках, однако неправильно вложенные теги приведут к неверному отображению страниц.

## Полноценный HTML-документ

До сих пор мы имели дело лишь с фрагментами HTML, тогда как полноценный HTML-документ должен включать некоторые дополнительные элементы. Давайте посмотрим на законченный HTML-документ и разберемся, зачем нужна каждая его часть. Добавьте в файл `page.html` следующие элементы:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Моя первая настоящая HTML-страничка</title>
  </head>

  <body>
    <h1>Привет, мир!</h1>
    <p>Моя <em>первая</em> <strong>веб-страничка</strong>.</p>
    <p><strong><em>параграф</em></strong>.</p>
  </body>
</html>
```

**Head** — здесь «шапка документа»

**Title** — название

**Body** — тело документа

**!** *Sublime Text автоматически ставит отступы при вводе некоторых строк кода, как показано в этом примере. По тегам (таким как `<html>`, `<h1>` и т. д.) он определяет, внутри каких элементов находится каждая строка, и делает отступы в соответствии с этим. Перед тегами `<head>` и `<body>` Sublime Text, в отличие от некоторых других редакторов, отступов не ставит.*

На рис. 5.6 показан законченный HTML-документ.

Давайте по очереди рассмотрим элементы из файла `page.html`. Тег `<!DOCTYPE html>` — всего лишь объявление, он сообщает: «это HTML-документ». Далее следует открывающий тег `<html>` (закрывающий тег `</html>` находится в самом конце кода). Каждый

HTML-документ должен содержать элемент `html` верхнего уровня вложенности.

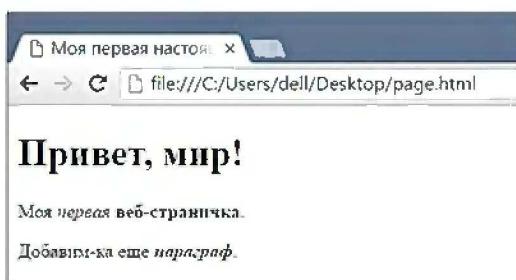


Рис. 5.6. Законченный HTML-документ

Внутри элемента `html` находятся элементы `head` и `body`. Элемент `head` содержит определенную информацию об HTML-документе, например элемент `title`, устанавливающий название документа, — обратите внимание, что текст на закладке браузера на рис. 5.6 («Моя первая настоящая HTML-страничка») соответствует содержимому `title`. Элемент `title` находится внутри элемента `head`, который, в свою очередь, находится внутри элемента `html`.

Внутри элемента `body` находится содержимое, которое отображается в браузере. В данном случае мы просто скопировали эти данные из предыдущего примера.

## Иерархия HTML

HTML-элементы подчинены строгой иерархии, которую можно себе представить в виде перевернутого дерева. На рис. 5.7 в виде дерева показан наш документ.

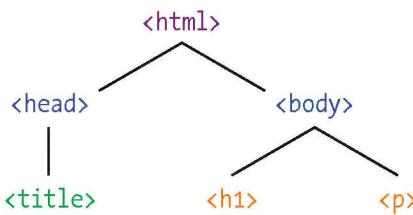
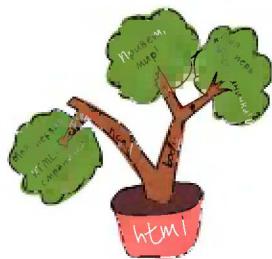


Рис. 5.7. Элементы страницы с рис. 5.6, показанные в виде дерева

Сверху находится элемент `html`. Он содержит элементы `head` и `body`. В свою очередь, `head` содержит элемент `title`, а `body` — элементы `h1` и `p`. Браузер интерпретирует наш HTML согласно этой

иерархии. О том, как менять структуру документа, мы узнаем позже, в девятой главе.

На рис. 5.8 показан другой способ изображения иерархии HTML — в виде вложенных прямоугольников.

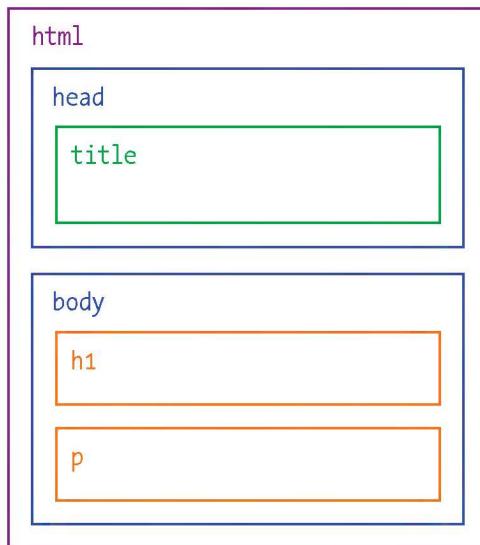


Рис. 5.8. Иерархия HTML в виде вложенных прямоугольников

## Добавим в HTML ссылки

Ранее в этой главе мы узнали, что HTML — гипертекстовый язык. Это значит, что HTML-документы могут содержать *гиперссылки* (или просто *ссылки*), ведущие на другие веб-страницы. Такие ссылки можно создавать с помощью элемента  (от английского *anchor* — «якорь»).

Измените свой HTML-документ, чтобы он соответствовал следующему примеру: удалите второй элемент *p*, а также теги *em* и *strong* и добавьте выделенный цветом код, чтобы создать ссылку на интернет-адрес <http://comicsia.ru/collections/xkcd>:

```
!DOCTYPE html
<html>
  <head>
    <meta charset="UTF-8">
    <title>Моя первая настоящая HTML-страничка</title>
  </head>

  <body>
    <h1>Привет, мир!</h1>
```

```
<p>Моя первая веб-страничка.</p>
<p><a href="http://comicsia.ru/collections/xkcd">Нажмите сюда</a>, чтобы почитать отличные комиксы.</p>
</body>
</html>
```

---

Сохраните файл и откройте страничку в браузере — она должна выглядеть как на рис. 5.9.

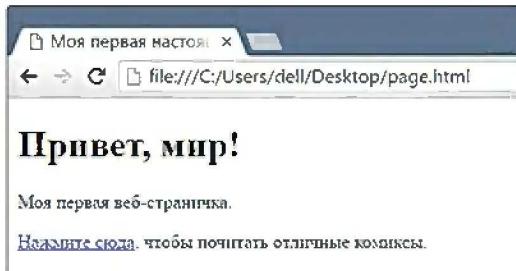


Рис. 5.9. Веб-страница со ссылкой на <http://comicsia.ru/collections/xkcd>

Если кликнуть по этой ссылке, браузер должен перейти по адресу <http://comicsia.ru/collections/xkcd>. Насладившись комиксами (выберите там тег «программисты» и почитайте смешные истории из жизни разработчиков), кликните на кнопку «назад», чтобы вернуться к нашей страничке.



## Атрибуты ссылок

Давайте разберемся, как мы создали эту HTML-ссылку. Чтобы браузер знал, куда перейти по клику, мы добавили элементу `a` так называемый *атрибут*. Атрибуты HTML-документов напоминают пары «ключ-значение» в объектах JavaScript: у каждого атрибута есть имя и значение. Посмотрите еще раз на созданную нами ссылку:

---

```
<a href="http://comicsia.ru/collections/xkcd">Нажмите сюда</a>
```

---

**Href** —  
от hypertext  
reference —  
гипертекстова  
яя ссылка

В данном случае у атрибута есть имя `href` и значение "<http://comicsia.ru/collections/xkcd>" — то есть веб-адрес.

На рис. 5.10 показаны все составные части этой ссылки.

Ссылка отправит вас по любому адресу, который указан в качестве значения атрибута `href`.



Рис. 5.10. Базовый синтаксис для создания гиперссылки

## Атрибут title

Также к ссылкам можно добавлять атрибут `title` — он задает текст, который появляется при наведении курсора на ссылку. Например, давайте изменим открывающий тег `<a>`, чтобы он выглядел так:

---

```
<a href="http://comicsia.ru/collections/xkcd" title="xkcd: Комиксы для гиков!">Нажмите сюда</a>
```

---

Теперь перезагрузите страничку. При наведении мышки на ссылку должна появиться надпись: «`xkcd: Комиксы для гиков!`», как на рис. 5.11.

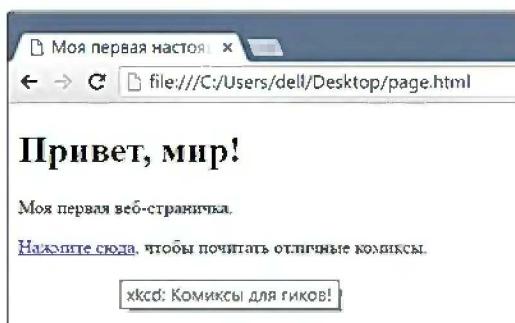


Рис. 5.11. Веб-страничка, содержащая ссылку на адрес `http://comicsia.ru/collections/xkcd/` с атрибутом `title`

## ПОПРОБУЙТЕ!

Создайте новый файл под названием `links.html`. Пусть его HTML-структура будет такой же, как у странички `page.html`, однако название и заголовок поменяйте на другие, а также добавьте три элемента `p` («параграф»). В каждом параграфе создайте ссылку на один из своих любимых сайтов. Убедитесь, что для всех элементов `a` заданы атрибуты `href` и `title`.

**Links** — ссылки

## Что мы узнали

В этой главе мы познакомились с основами HTML — языка для создания веб-страниц. Также мы создали простой HTML-документ со ссылкой на другую страницу.

В следующей главе мы разберемся, как встраивать в нашу страничку JavaScript-код. Это облегчит создание более объемных программ по мере изучения новых возможностей JavaScript.

Эта книга посвящена JavaScript, а не HTML, поэтому я рассмотрел лишь самые азы создания HTML-документов. Вот некоторые ресурсы, где можно узнать о HTML больше:

На английском языке:

- Курс HTML и CSS от Codecademy: <http://www.codecademy.com/tracks/web/>
- Mozilla Webmaker: <https://webmaker.org/>

На русском языке:

- Введение в HTML от Mozilla Developer Network: <https://developer.mozilla.org/ru/docs/Web/Guide/HTML/Introduction>
- <https://htmlacademy.ru/>

# 6

## УСЛОВИЯ И ЦИКЛЫ

Условные конструкции и циклы — одни из самых важных понятий в JavaScript. Условная конструкция представляет собой команду: «если что-то истинно (`true`), сделай это, иначе сделай то». Пример: выполнив домашнее задание, вы можете съесть мороженое, но, если домашнее задание не готово, мороженое вам не светит. А цикл — это инструкция: «до тех пор, пока что-то истинно (`true`), продолжай делать это». Пример: пока вы испытываете жажду, продолжайте пить воду.

Условные конструкции и циклы — понятия, лежащие в основе любой мало-мальски серьезной программы. Их называют *управляющими конструкциями*, поскольку они позволяют решать, какие части кода и когда выполнять, а также насколько часто это нужно делать, исходя из заданных вами условий.

Для начала давайте разберемся, как встраивать JavaScript в HTML-файл. Это позволит писать программы более сложные, чем те, с которыми мы имели дело до сих пор.

### Внедрение JavaScript-кода в HTML

Вот HTML-файл, который мы создали в пятой главе, с некоторыми дополнениями — они показаны в цвете, тогда как прежний текст набран серым (чтобы упростить этот пример, я убрал из него ссылку на <http://comicsia.ru/collections/xkcd/>).

**Script** —  
скрипт,  
сценарий

**Message** —  
сообщение

---

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Моя первая настоящая HTML-страничка</title>
</head>
<body>
<h1>Привет, мир!</h1>
<p>Моя первая веб-страничка.</p>
<script>
var message = "Привет, мир!";
console.log(message);
</script>
</body>
</html>
```

---

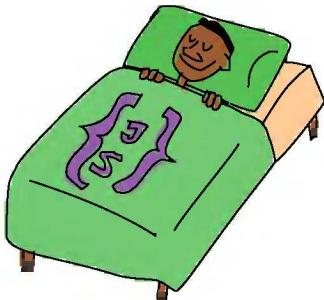
Как видите, мы добавили новый элемент под названием `script`. Этот элемент особенный: содержимое большинства элементов HTML отображается на страничке, однако то, что находится внутри тегов `script`, считается JavaScript-кодом и передается для выполнения интерпретатору JavaScript.

Теперь рассмотрим код внутри элемента `script`:

---

```
var message = "Привет, мир!";
❶ console.log(message);
```

---



Выполнение JavaScript, встроенного в HTML, заметно отличается от запуска кода в консоли. Введенный в консоли код выполнится при первом же нажатии ENTER, после чего вы увидите результат сработавшей команды. Однако код, встроенный в веб-страницу, выполняется сразу и целиком, от верхних строк к нижним, причем в консоль ничего автоматически не выводится — если мы не попросим браузер об этом отдельно. Для вывода в консоль можно воспользоваться командой `console.log` — это поможет следить за ходом выполнения программы. Метод `console.log` принимает любое значение и печатает (*логирует*) это значение в консоли. Например, загрузив в браузер наш последний пример, вы увидите в консоли вот что (разумеется, если она открыта):

---

```
Привет, мир!
```

---

Вызов `console.log(message)` в строке ❶ привел к появлению в консоли строки "Привет, мир!".

Теперь, когда вы знаете, как с удобством писать длинные JavaScript-программы, можно перейти к изучению условных конструкций.

## Условные конструкции

В JavaScript есть два вида условных конструкций — это `if` и `if...else`. Оператор `if` выполняет фрагмент кода, если какое-то условие истинно (`true`). Например: *если* вы хорошо себя вели, то получите конфетку. А оператор `if...else` выполняет один фрагмент кода, если условие дает `true`, и другой фрагмент в противном случае. Например: *если* вы хорошо себя вели, получите конфетку, *иначе* вас не отпустят гулять.

**If** — если  
**If... else** —  
если... иначе

### Конструкция `if`

Самая простая из управляющих конструкций JavaScript — это `if`. Она используется, чтобы запускать код, если некое условие истинно (`true`). Вернитесь к нашему HTML-файлу и замените содержимое элемента `script` следующими строками:

---

```
❶ var name = "Николай";
❷ console.log("Привет, " + name);
❸ if (name.length > 6) {
❹   console.log("Ну и длиннющее же у вас имя!");
}
```

---

Сначала в строке ❶ мы создали переменную `name` и присвоили ей значение — строку "Николай". Затем, в строке ❷, мы с помощью `console.log` напечатали строку "Привет, Николай".

В строке ❸ мы использовали конструкцию `if`, чтобы проверить: длина `name` больше, чем шесть символов? Если это так, в строке ❹ мы посредством `console.log` выводим: "Ну и длиннющее же у вас имя!".

Как показано на рис. 6.1, конструкция `if` состоит из двух частей: условия и тела. Условие должно давать булево значение. А тело — одна или несколько строк JavaScript-кода, которые будут выполнены, если условие истинно (`true`).

Конструкция `if` проверяет, истинно ли условие



```
if (condition) {
  console.log("Делаем что-то");
}
```

Код, который выполняется, если условие дает `true`,  
называется *телом if*

Рис. 6.1. Общая структура конструкции `if`

После загрузки нашей HTML-странички со встроенным JavaScript-кодом в консоли должно появиться:

---

```
Привет, Николай
Ну и длиннее же у вас имя!
```

---

В имени *Николай* 7 букв, поэтому `name.length` вернет значение 7, и условие `name.length > 6` даст `true`. В результате будет выполнено тело оператора `if`, и в консоли появится несколько фамильярное сообщение. Чтобы избежать выполнения `if`, поменяйте имя *Николай* на *Ник* (оставив остальной код без изменений):

---

```
var name = "Ник";
```

---

Теперь сохраните файл и перегрузите страничку. На этот раз условие `name.length > 6` даст `false`, поскольку `name.length` равно 3. В итоге тело оператора `if` выполнено не будет, а в консоли появится лишь:

---

```
Привет, Ник
```

---

Тело оператора `if` выполняется, только когда условие дает `true`. Если же условие дает `false`, интерпретатор игнорирует конструкцию `if` и переходит к следующей за ней строке.

## Конструкция `if... else`

Как я уже говорил, оператор `if` запускает код своего тела, только если условие дает `true`. Но если вы хотите, чтобы по условию `false` тоже что-то происходило, вам нужна конструкция `if... else`.

Давайте дополним предыдущий пример:

---

```
var name = "Николай";
console.log("Привет, " + name);
if (name.length > 6) {
  console.log("Ну и длиннее же у вас имя!");
} else {
  console.log("Имя у вас не из длинных.");
}
```

---

Этот код делает практически то же, что и раньше, однако, если имя (`name`) не длиннее 6 символов, он выводит другое, альтернативное сообщение.

Как видно по рис. 6.2, конструкция `if... else` похожа на конструкцию `if`, однако у нее целых два тела, между которыми расположено ключевое слово `else`. Первое тело будет выполнено, если условие дает `true`, иначе выполняется код второго тела.

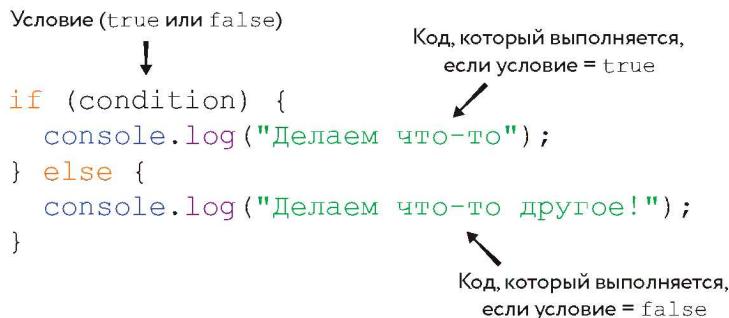


Рис. 6.2. Общая структура конструкции `if... else`

## Цепочка из конструкций `if... else`

Зачастую нужно проверить несколько условий и сделать что-то, если одно из них дает `true`. Пример: вы пришли в китайский ресторан и выбираете, что бы такое съесть. Больше всего вы любите курицу с лимоном (lemon chicken), и, если она есть в меню, вы ее закажете. Если же ее нет, вы закажете говядину в соусе из черных бобов (beef with black bean). Однако если и это блюдо отсутствует, вы остановитесь на свинине в кисло-сладком соусе (sweet and sour pork). Наконец, в маловероятном случае, когда нет ни одного из этих блюд, вы закажете рис с яйцом, поскольку знаете, что его подают во всех китайских ресторанах.

---

```

var lemonChicken = false;
var beefWithBlackBean = true;
var sweetAndSourPork = true;

if (lemonChicken) {
    console.log("Отлично! Я буду курицу с лимоном!");
} else if (beefWithBlackBean) {
    console.log("Заказываю говядину.");
} else if (sweetAndSourPork) {
    console.log("Ладно, закажу свинину.");
} else {
    console.log("Что ж, остается рис с яйцом.");
}
    
```

---

Чтобы создать цепочку `if... else`, начните с обычного оператора `if` и после закрывающей фигурной скобки его тела введите ключевые слова `else if`, а следом — еще одно условие и еще одно тело. После можно добавить еще `else if`, и так до тех пор, пока у вас не закончатся условия (которых может быть сколько угодно). Завершающая секция `else` будет выполнена, если ни одно из условий не дает `true`. На рис. 6.3 показана классическая цепочка конструкций `if... else`.

```
if (condition1) {  
    console.log("Сделай это, если условие 1 истинно");  
} else if (condition2) {  
    console.log("Сделай это, если условие 2 истинно");  
} else if (condition3) {  
    console.log("Сделай это, если условие 3 истинно");  
} else {  
    console.log("Иначе сделай это");  
}
```

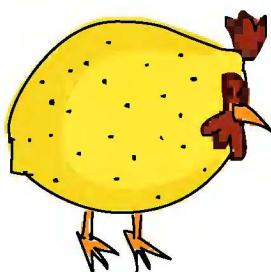
Для каждого условия указан код,  
который выполняется, если условие  
дает `true`

Код, который нужно выполнить,  
когда все условия дают `false`

Рис. 6.3. Цепочка конструкций `if... else`

Можно прочитать этот код так:

- Если первое условие дает `true`, выполнить код из первого тела.
- Иначе, если второе условие дает `true`, выполнить второе тело.
- Иначе, если третье условие дает `true`, выполнить третье тело.
- Иначе выполнить тело `else`.



Имея цепочку `if... else` с завершающей секцией `else`, можно не сомневаться, что одно (и только одно) из тел будет выполнено. Как только выяснится, что одно из условий дает `true`, будет запущен код из соответствующего тела, а последующие условия проверяться уже не будут. Если запустить код из предыдущего примера, мы увидим в консоли «Заказываю говядину», поскольку `beefWithBlackBean` — первое из условий в цепочке `if... else`, которое равно `true`. Если же ни одно из условий не даст `true`, будет выполнено тело `else`.

Также обратите внимание: указывать завершающее `else` не обязательно. Однако если вы этого не сделаете, то в случае, когда ни одно из условий не дает `true`, ничего из цепочки `if... else` выполнено не будет.

---

```
var lemonChicken = false;
var beefWithBlackBean = false;
var sweetAndSourPork = false;
if (lemonChicken) {
  console.log("Отлично! Я буду курицу с лимоном!");
} else if (beefWithBlackBean) {
  console.log("Заказываю говядину.");
} else if (sweetAndSourPork) {
  console.log("Ладно, закажу свинину.");
}
```

---

В этом примере мы не стали указывать завершающую секцию `else`. Поскольку ни одного из ваших любимых блюд нет, в консоли не появится никаких сообщений (и, по всей видимости, вы останетесь без обеда).

### ПОПРОБУЙТЕ!

Напишите программу с переменной `name`. Если в этой переменной находится ваше имя, напечатайте: «Привет мне!» — иначе напечатайте: «Привет, незнакомец!» (Подсказка: используйте `==` для сравнения переменной `name` с вашим именем.)

Теперь дополните программу, чтобы она здоровалась с вашим папой, если в `name` его имя, и с вашей мамой, если в `name` ее имя. Если же там что-то иное, по-прежнему печатайте «Привет, незнакомец!».

## Циклы

Как мы теперь знаем, условные конструкции позволяют запускать фрагмент кода, если условие дает `true`. Циклы, с другой стороны, позволяют выполнять фрагмент кода многократно — до тех пор, пока некое условие дает `true`. Примеры: до тех пор, пока в тарелке есть пища, следует продолжать есть; до тех пор, пока на лице грязь, следует продолжать умываться.



### Цикл `while`

Самый простой из циклов — цикл `while`. Этот цикл снова и снова выполняет код своего тела, до тех пор, пока заданное условие не перестанет давать `true`. Используя цикл `while`, мы имеем в виду следующее: «Продолжай делать это, пока условие дает `true`. Но если оно даст `false`, остановись».

**While** — до тех пор, пока

Как видно на рис. 6.4, цикл `while` начинается с ключевого слова `while`, после которого в скобках стоит условие, а за ним идет тело, заключенное в фигурные скобки.

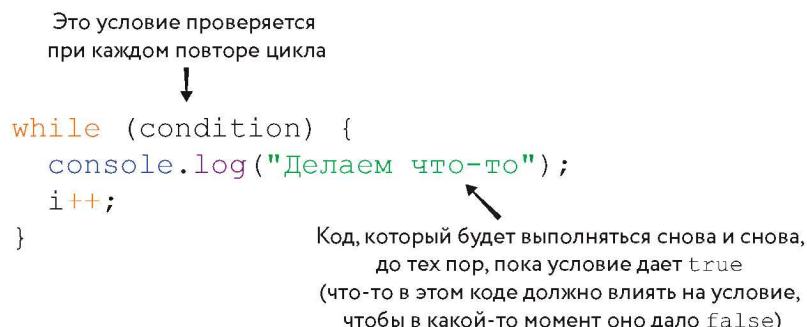


Рис. 6.4. Общая структура цикла `while`

Аналогично конструкции `if`, тело цикла `while` выполняется, если заданное условие дает `true`. Но, в отличие от `if`, после того как тело цикла выполнено, условие будет проверено снова, и, если оно все еще дает `true`, тело цикла начнет выполняться опять. И так будет продолжаться, пока условие не даст `false`.

### Считаем овец с помощью цикла `while`

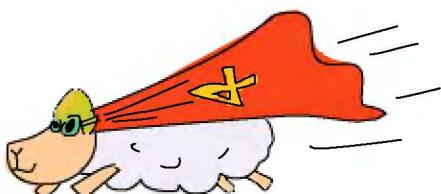
Предположим, у вас проблемы со сном и вы решили посчитать овец. Но раз уж вы программист, почему бы не написать программу, которая будет считать овец за вас?

Sheep  
counted —  
посчитанные  
овцы

---

```
var sheepCounted = 0;  
❶ while (sheepCounted < 10) {  
❷     console.log("Посчитано овец: " + sheepCounted + "!");  
     sheepCounted++;  
}  
console.log("Хррррррррррр-псссс");
```

---



Мы создали переменную `sheepCounted` и задали ей значение 0. Дойдя до цикла `while` в строке ❶, мы проверяем, правда ли, что `sheepCounted` меньше 10. Поскольку 0 меньше 10, выполняется код в фигурных скобках (тело цикла, которое начинается со строки ❷) и выражение "Посчитано овец: " + `sheepCounted` + "!" выводится в консоль как «Посчитано овец: 0!». Далее команда `sheepCounted++` увеличивает значение

`sheepCounted` на 1, мы возвращаемся к началу цикла, и все повторяется снова:

---

```
Посчитано овец: 0!
Посчитано овец: 1!
Посчитано овец: 2!
Посчитано овец: 3!
Посчитано овец: 4!
Посчитано овец: 5!
Посчитано овец: 6!
Посчитано овец: 7!
Посчитано овец: 8!
Посчитано овец: 9!
Хрррррррррр-посс
```

---

Тело цикла повторяется, пока `sheepCounted` не примет значение 10, после чего условие становится ложным (`false`), ведь 10 не меньше 10. И тогда программа переходит к строке, идущей после цикла, — в данном случае на консоль выводится "Хрррррррррр-псссс".

#### Бесконечный цикл

Имея дело с циклами, помните: если условие никогда не даст `false`, цикл будет повторяться бесконечно (по крайней мере до тех пор, пока вы не закроете страницу в браузере). Например, не будь в теле цикла строчки `sheepCounted++`, в `sheepCounted` всегда был бы 0 и программа печатала бы:

---

```
Посчитано овец: 0!
Посчитано овец: 0!
Посчитано овец: 0!
Посчитано овец: 0!
...
...
```

---

Поскольку повторения цикла ничем не ограничены, программа будет печатать эту строку снова и снова, без конца. Это называется **бесконечным циклом**.

## Цикл `for`

Оператор `for` упрощает создание циклов, устроенных следующим образом: сначала создается переменная, а затем тело цикла выполняется снова и снова до тех пор, пока условие дает `true`, причем в конце каждого повтора значение переменной обновляется. Программируя цикл `for`, мы создаем переменную, задаем условие, указываем, как

**For** — для

должна меняться переменная после каждого повтора, — и лишь затем переходим к написанию тела цикла. Например, вот как можно считать овец с помощью `for`:

```
for (var sheepCounted = 0; sheepCounted < 10; sheepCounted++) {  
    console.log("Посчитано овец: " + sheepCounted + "!");  
}  
console.log("Хппппппппп-пссс");
```

Согласно рис. 6.5, в составе цикла `for` есть три выражения, разделенные точками с запятой: это настройка, проверка условия и приращение.



Рис. 6.5. Общая структура цикла `for`

*Настройка* (`var sheepCounted = 0`) выполняется до запуска цикла. Как правило, здесь создают переменную для отслеживания количества повторов. В нашем случае это переменная `sheepCounted` с начальным значением 0.

*Условие* (`sheepCounted < 10`) проверяется перед каждым повтором тела цикла. Если условие дает `true`, тело выполняется, иначе цикл заканчивает работу. В нашем случае цикл остановится, когда значение `sheepCounted` достигнет 10.

*Приращение* (`sheepCounted++`) выполняется после каждого повтора тела цикла. Как правило, здесь изменяют значение переменной цикла. В этом примере мы после каждого повтора увеличиваем `sheepCounted` на 1.

Циклы `for` удобны, когда нужно сделать что-то определенное количество раз. Например, эта программа три раза выведет слово «Привет!».

Times to  
say hello —  
сколько  
раз сказать  
«привет»

```
var timesToSayHello = 3;  
for (var i = 0; i < timesToSayHello; i++) {  
    console.log("Привет!");  
}
```

Вот что появится в консоли:

---

```
Привет!  
Привет!  
Привет!
```

---

Вообразите, что вы интерпретатор JavaScript, который выполняет этот код. Сначала вы создадите переменную `timesToSayHello`, задав ей значение 3. Дойдя до цикла `for`, вы выполните настройку, то есть создадите переменную `i` и присвойте ей значение 0. Далее вы проверите условие. Поскольку в переменной `i` сейчас 0, а в `timesToSayHello` — 3, условие даст `true` и вы запустите тело цикла, где печатается строка "Привет!". А затем выполните приращение, то есть увеличите `i` на 1.

Теперь снова проверьте условие. Оно по-прежнему даст `true`, и вы опять перейдете к телу цикла, а затем к приращению. И так будет происходить до тех пор, пока `i` не примет значение 3. После этого условие даст `false` (3 не меньше, чем 3) — таким образом, вы завершите цикл.

#### Цикл `for`, массивы и строки

Очень часто цикл `for` используют для перебора всех элементов массива или всех символов строки. Например, вот цикл, который печатает названия всех животных, которые есть в зоопарке:

---

```
var animals = ["лев", "фламинго", "белый медведь", "удав"];
for (var i = 0; i < animals.length; i++) {
    console.log("В этом зоопарке есть " + animals[i] + ".");
}
```

---

В этом цикле `i` сначала равняется 0, а затем возрастает до значения `animals.length - 1`, то есть 3. Числа 0, 1, 2 и 3 — индексы элементов в массиве `animals`. Это значит, что при каждом повторе цикла `i` принимает значение очередного индекса, а `animals[i]` соответствует очередному животному из массива `animals`. Когда в `i` число 0, `animals[i]` даст нам строку "лев". Когда в `i` число 1, `animals[i]` даст "фламинго" и т. д.

Запустив эту программу, мы увидим:

---

```
В этом зоопарке есть лев.  
В этом зоопарке есть фламинго.  
В этом зоопарке есть белый медведь.  
В этом зоопарке есть удав.
```

---



Как мы уже знаем из второй главы, к отдельным символам строки можно обращаться тем же способом, что и к элементам массива, — с помощью квадратных скобок. В следующем примере цикл `for` используется для вывода символов имени:

---

```
var name = "Ник";
for (var i = 0; i < name.length; i++) {
    console.log("В моем имени есть буква " + name[i] + ".");
}
```

---

Вот что выдаст эта программа:

---

```
В моем имени есть буква Н.
В моем имени есть буква и.
В моем имени есть буква к.
```

---

#### Другие варианты применения `for`

Как вы, может быть, догадываетесь, не обязательно сначала задавать переменной цикла значение 0, а затем каждый раз увеличивать ее на 1. Например, вот как можно напечатать все степени двойки, не превышающие числа 10 000:

---

```
for (var x = 2; x < 10000; x = x * 2) {
    console.log(x);
}
```

---

Здесь мы присваиваем `x` значение 2 и увеличиваем его командой `x = x * 2`, то есть, удваиваем значение `x` при каждом повторе цикла. В результате `x` очень быстро возрастает:

---

```
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
```

---

Вуаля! Этот несложный цикл печатает все степени двойки меньше 10 000.

### ПОПРОБУЙТЕ!

Напишите цикл `for`, который печатает степени тройки, не превышающие 10 000 (программа должна выводить 3, 9, 27 и т. д.)

Перепишите это задание, вместо `for` использовав цикл `while`.  
(Подсказка: установите начальное значение перед входом в цикл.)

### Что мы узнали

Мы разобрались с условными конструкциями и циклами. Условные конструкции нужны, чтобы выполнять некие действия, если условие дает `true`. А с помощью циклов можно многократно выполнять фрагмент кода — до тех пор, пока условие дает `true`. Условными конструкциями можно пользоваться, чтобы запускать код в подходящие для этого моменты, а циклами — чтобы программа выполнялась так долго, как требуется. И это открывает перед нами целый мир новых возможностей в программировании.

В следующей главе мы воспользуемся мощью условных конструкций и циклов при создании нашей первой игры!



## УПРАЖНЕНИЯ

Выполните эти упражнения, чтобы попрактиковаться в работе с условными конструкциями и циклами.

## #1. Прекрасные животные

Напишите цикл `for`, который изменяет массив животных, делая их прекрасными! Например, если есть следующий массив:

```
var animals = ["Кот", "Рыба", "Лемур", "Комодский варан"];
```

то ваш цикл должен сделать его таким:

["Кот - прекрасное животное", "Рыба - прекрасное животное", "Лемур - прекрасное животное", "Комодский варан - прекрасное животное"]

Подсказка: вам понадобится *переприсвоить* значения для каждого индекса, то есть присвоить новые значения уже существующим элементам. Например, сделать первое животное прекрасным можно так:

```
animals[0] = animals[0] + " - прекрасное животное";
```

## #2. Генератор случайных строк

Напишите генератор случайных строк. Для этого вам понадобится строка со всеми буквами алфавита:

## Alphabet — алфавит

```
var alphabet = "абвгдеёжзийклмнопрстуфхцшъъюя";
```

Чтобы выбирать из этой строки случайную букву, можно использовать примерно такой же код, как для генератора случайных дразнилок из третьей главы: `Math.floor(Math.random() * alphabet.length)`. Так вы получите случайный индекс в строке. Затем, воспользовавшись квадратными скобками, можно получить символ по этому индексу.

Начните создавать случайную строку с пустой строки (`var randomString = ""`). Затем добавьте цикл `while` и при каждом его повторе добавляйте в строку новый случайный символ — до тех пор, пока длина строки `randomString` не превысит шесть символов (или любой другой длины на ваш выбор).

Добавлять символ в конец строки можно с помощью оператора `+=`. После того как цикл закончит работу, выведите получившуюся строку в консоль, чтобы полюбоваться на свое творение!

### #3. h4ck3r sp34k

Переведите англоязычный текст на «хакерский язык» (`h4ck3r sp34k`)! Многим в интернете нравится заменять некоторые буквы похожими на них числами — например, число «4» похоже на букву «A», «3» похоже на «E», «1» — на «I», а «0» — на «O». Хотя цифры напоминают скорее заглавные буквы, мы будем заменять ими буквы строчные. Чтобы перевести обычный текст на «хакерский язык», понадобится строка с исходным текстом и новая пустая строка для результата:

---

```
var input = "javascript is awesome";
var output = "";
```

---

Теперь воспользуйтесь циклом `for`, чтобы перебрать все символы исходной строки. Встретив букву «a», добавьте к результатирующей строке «4». Встретив «e», добавьте «3», встретив «i», добавьте «1», а встретив «o» — «0». В противном случае просто добавляйте к результату исходный символ. И снова оператор `+=` отлично подойдет для добавления символа в конец строки.

После завершения цикла выведите результатирующую строку в консоль. Если программа работает верно, вы должны увидеть следующее: `"j4v4scr1pt ls 4w3s0m3"`.

`h4ck3r sp34k` —  
`hacker speak` —  
хакерский  
язык

**Input** —  
входное  
значение

**Output** —  
выходное  
значение

`JavaScript`  
`is awesome` —  
JavaScript  
очень  
классный

# 7

## ПИШЕМ ИГРУ «ВИСЕЛИЦА»

В этой главе мы разработаем игру «Виселица» и разберемся, как с помощью диалоговых окон сделать ее интерактивной, запрашивая у игрока данные.

«Виселица» — игра на угадывание слов. Один игрок выбирает слово, а второй пытается его отгадать. Например, если первый игрок загадал слово КАПУСТА, он изобразит семь «пустых мест», по одному на каждую букву слова:

-----

Второй игрок старается отгадать это слово, называя буквы. Каждый раз, когда он угадывает букву, первый игрок заполняет пустоты, вписывая ее везде, где она встречается. Например, если второй игрок назвал букву «А», первый должен вписать все «А» для слова КАПУСТА, вот так:

\_ A \_ \_ \_ A

Если второй игрок назовет букву, которой нет в слове, у него отнимается очко, а первый игрок рисует руку, ногу или другую часть тела человечка. Если первый игрок закончит рисовать человечка раньше, чем второй угадает все буквы, второй игрок проиграл.

В нашем варианте «Виселицы» JavaScript будет выбирать слово, а игрок-человек — отгадывать буквы. И рисовать человечка наша программа не будет, поскольку мы пока не знаем, как это делается (рисованием на JavaScript мы займемся в главе 13).

## Взаимодействие с игроком

Для этой игры нам нужно, чтобы игрок (человек) мог каким-то образом вводить в программу свои ответы. Один из способов это сделать — открывать диалоговое окно (в JavaScript оно называется *prompt*), в котором игрок может что-нибудь напечатать.

**Prompt** —  
здесь «запрос»

### Создаем диалоговое окно

Сначала создадим новый HTML-документ. Выбрав в меню File → Save As, сохраните файл *page.html* из пятой главы под новым именем — *prompt.html*. Чтобы создать диалоговое окно, введите следующий код между тегов <script>, а затем откройте файл *prompt.html* в браузере:

```
var name = prompt("Как вас зовут?");
console.log("Привет, " + name);
```

Здесь мы создали новую переменную *name* и присвоили ей значение, которое вернул вызов *prompt*("Как вас зовут?"). При вызове *prompt* открывается маленькое диалоговое окно (часто его называют просто *диалог*), показанное на рис. 7.1.

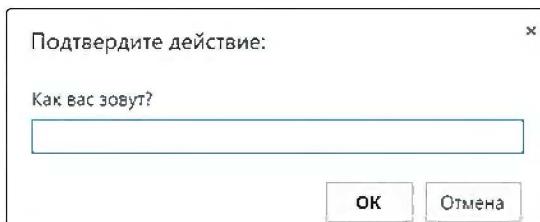


Рис. 7.1. Диалоговое окно *prompt*

Вызов *prompt*("Как вас зовут?") создает окно с запросом «Как вас зовут?» и строкой для ввода текста. В нижней части этого диалога есть две кнопки — «ОК» и «Отмена».

Если вы введете какой-нибудь текст и нажмете «ОК», этот текст станет значением, которое вернет в программу *prompt*. Например, если я введу свое имя и нажму «ОК», JavaScript напечатает в консоли:

```
Привет, Ник
```

Поскольку я ввел *Ник* и нажал на «ОК», строка «Ник» попала в переменную *name*, а вызов *console.log* напечатал: «Привет, " + "Ник"», то есть «Привет, Ник».



Когда вы во второй раз создадите какое-либо диалоговое окно Chrome, в окне появится кнопка-флажок с подписью «Предотвратить создание дополнительных диалоговых окон на этой странице». Таким образом Chrome защищает пользователей от веб-страниц, открывающих множество назойливых рекламных окошек. Выполняя примеры из этой главы, просто отключите этот флажок.

### А ЧТО ЕСЛИ ВЫ НАЖМЕТЕ «ОТМЕНА»?

Если вы нажмете кнопку «Отмена», `prompt` вернет значение `null`. Как нам известно из второй главы, `null` используется для обозначения чего-либо, что намеренно оставлено пустым.

После нажатия «Отмена» в консоли должно появиться:

```
Привет, null
```

В данном случае `console.log` печатает `null` как строку. Вообще-то `null` строкой не является, но, поскольку в консоль можно выводить только строки и вы попросили JavaScript напечатать "Привет, " + `null`, JavaScript преобразовал `null` в строку "`null`", чтобы напечатать это значение. Ситуация, когда JavaScript автоматически преобразует значение к другому типу, называется **неявным приведением типа**.

Неявное приведение типа — пример того, как JavaScript старается быть умным. Способа объединить строку и `null` не существует, и JavaScript делает лучшее, на что он способен. В данном случае он знает, что для успешного выполнения операции нужны две строки. Строковая версия значения `null` — это "`null`", и в результате мы видим в консоли "Привет, `null`".



**Confirm** — подтвердить

**Likes cats** — нравятся кошки

### Используем `confirm`, чтобы получить ответ «да» или «нет»

Функция `confirm` позволяет задать пользователю вопрос, на который он может ответить «да» или «нет» (что соответствует булеву значению). В следующем примере мы используем `confirm`, чтобы спросить у пользователя, нравятся ли ему кошки (см. рис. 7.2).

Если получен утвердительный ответ, переменная `likesCats` принимает значение `true` и мы печатаем: «Ты классная кошка!» Если же кошки пользователю не нравятся, `likesCats` принимает значение `false`, и мы отвечаем: «Что ж, не проблема. Все равно ты молодец!»

```
var likesCats = confirm("Тебе нравятся кошки?");  
if (likesCats) {  
    console.log("Ты классная кошка!");  
} else {  
    console.log("Что ж, не проблема. Все равно ты молодец!");  
}
```

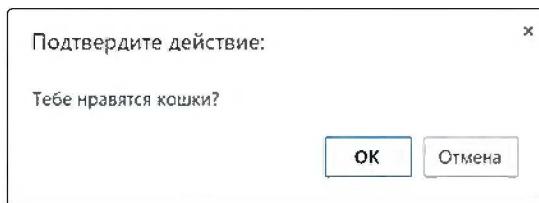


Рис. 7.2. Диалог confirm

Ответ на вопрос, заданный с помощью `confirm`, возвращается в программу как булево значение. Если в окне, показанном на рис. 7.2, пользователь нажмет «OK», `confirm` вернет `true`. Если же пользователь нажмет «Отмена», `confirm` вернет `false`.

## Используем alert для выдачи информации

Если требуется просто показать что-то пользователю, можно воспользоваться диалогом `alert`, который отображает сообщение с кнопкой «OK». Например, если вы считаете, что JavaScript — это здорово, вы можете использовать `alert` так:

```
alert("JavaScript это здорово!");
```

**Alert** — предупреждение

На рис. 7.3 показано, как выглядит этот диалог.

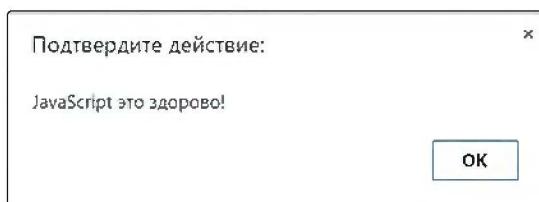


Рис. 7.3. Диалог alert

Диалог `alert` просто отображает сообщение до тех пор, пока пользователь не нажмет «OK».



## Чем alert лучше console.log?

Зачем нужен диалог `alert`, когда есть `console.log`? Во-первых, если необходимо просто сообщить о чем-то игроку, `alert` позволяет сделать именно это — не требуя, чтобы игрок открыл консоль и прочитал сообщение. Во-вторых, вызов `alert` (а также `prompt` и `confirm`) приостанавливает работу интерпретатора JavaScript до нажатия кнопки «OK» (или «Отмена», в случае `prompt`) и — значит, у пользователя будет достаточно времени, чтобы прочитать сообщение. А при использовании `console.log` текст отображается в консоли, а интерпретатор тут же переходит к следующей строке программы.

### Проектирование игры

Прежде чем перейти к созданию игры «Виселица», давайте подумаем о ее структуре. Нам нужно, чтобы программа умела выполнять следующие действия:

1. Случайным образом выбирать слово.
2. Запрашивать у игрока вариант ответа (букву).
3. Завершать игру по желанию игрока.
4. Проверять, является ли введенный ответ буквой.
5. Вести учет угаданных букв.
6. Показывать игроку, сколько букв он угадал и сколько еще предстоит угадать.
7. Завершать игру, если слово отгадано.

Все эти действия, кроме первого и последнего (выбор слова и завершение игры), нужно выполнять многократно, причем заранее неизвестно, сколько раз (это зависит от ответов игрока). И, как мы теперь знаем, если требуется повторять какие-то действия, значит, в программе нужен цикл.

Однако в нашем списке действий ничего не говорится о том, что и когда должно происходить. Чтобы выяснить этот вопрос и лучше представить себе структуру будущей программы, мы можем воспользоваться псевдокодом.

## Используем псевдокод для проектирования игры

Псевдокод — удобный инструмент, который программисты часто используют при проектировании программ. Слово «псевдокод» означает «ненастоящий код». Хотя в псевдокоде есть циклы и условия, в целом программа описывается обычным человеческим языком. Чтобы разобраться, что это значит, давайте посмотрим на описание нашей игры в псевдокоде:

---

```
Выбрать случайное слово
Пока слово не угадано {
    Показать игроку текущее состояние игры
    Запросить у игрока вариант ответа
    Если игрок хочет выйти из игры {
        Выйти из игры
    }
    Иначе Если вариант ответа — не одиночная буква {
        Сообщить игроку, что он должен ввести букву
    }
    Иначе {
        Если такая буква есть в слове {
            Обновить состояние игры, подставив новую букву
        }
    }
}
Поздравить игрока с победой — слово угадано
```

---

Как видите, это не программный код, который может выполнить компьютер. Однако такая запись дает нам представление о структуре программы прежде, чем мы перейдем к написанию кода и выяснению мелких деталей, например, как именно выбирать случайное слово.

## Отображение состояния игры

Одна из строк нашего псевдокода гласит: «Показать игроку текущее состояние игры». Для игры «Виселица» это означает подставить в слово угаданные игроком буквы, а также показать, какие буквы осталось угадать. Как мы будем это делать? В сущности, можно хранить состояние игры тем же способом, что и в обычной «Виселице»: в виде последовательности «пустых мест», которые мы будем заполнять по мере того, как игрок угадывает буквы.

Мы сделаем это с помощью массива «пустых мест» — по одному элементу для каждой буквы в слове. Назовем этот массив «итоговым массивом» и будем по ходу игры заполнять его угаданными буквами. А каждое из «пустых мест» представим в виде строки со знаком подчеркивания: \_.

Сначала наш итоговый массив будет просто набором «пустых мест», количество которых равно количеству букв в загаданном слове. Например, если загадано слово «рыба», массив будет выглядеть так:

---

```
[ " ", " ", " ", " ", " "]
```

---

Если игрок угадает букву «ы», мы заменим второй элемент на «ы»:

---

```
[ " ", "ы", " ", " ", " "]
```

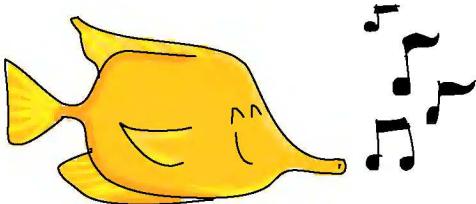
---

А когда игрок угадает все буквы, массив примет вид:

---

```
[ "р", "ы", "б", "а" ]
```

---



Также нам понадобится переменная для хранения количества букв, которые осталось угадать. Для каждого вхождения верно угаданной буквы эта переменная будет уменьшаться на 1, и, когда она примет значение 0, мы поймем, что игрок победил.

## Проектируем игровой цикл

Основная часть игры будет располагаться внутри цикла `while` (в нашем псевдокоде этот цикл начинается со строки «Пока слово не угадано»). В цикле мы будем отображать текущее состояние игры (то есть слово, поначалу представленное одними знаками подчеркивания), запрашивать у игрока вариант ответа (и проверять, действительно ли тот ввел одиночную букву), а также обновлять итоговый массив, подставляя введенную букву, если она действительно есть в слове.

Практически все компьютерные игры организованы в виде того или иного цикла, нередко структурно похожего на цикл нашей «Виселицы». В целом игровой цикл выполняет следующие задачи:

1. Принимает ввод от игрока.
2. Обновляет состояние игры.
3. Показывает игроку текущее состояние игры.

Такого рода цикл применяется даже в играх, где непрерывно что-то меняется, — просто он выполняется очень быстро. В случае нашей

«Виселицы» программа запрашивает у игрока вариант ответа, обновляет итоговый массив (если ответ верный) и отображает новое состояние итогового массива.

Если игрок угадает все буквы в слове, мы должны показать ему законченное слово, а также вывести сообщение, поздравляющее с победой.

## Программируем игру

Теперь, когда у нас есть представление о структуре игры, можно переходить к написанию кода. Сначала мы рассмотрим его по частям. После этого вы увидите весь код целиком, чтобы с удобством ввести его и поиграть.

### Выбираем случайное слово

Первым делом нам нужно выбрать случайное слово. Вот как это делается:

```
❶ var words = [  
    "программа",  
    "макака",  
    "прекрасный",  
    "оладушек"  
];  
  
❷ var word = words[Math.floor(Math.random() * words.length)];
```

Words — слова

Наша игра начинается со строки ❶, где мы создаем массив со словами (*программа*, *макака*, *прекрасный* и *оладушек*), из которого затем будем выбирать слово для отгадывания (все слова должны быть записаны строчными буквами). Сохраним этот массив в переменной *words*. В строке ❷ мы используем *Math.random* и *Math.floor*, чтобы выбрать из массива случайное слово — так же как в третьей главе выбирали слова для генератора дразнилок.

### Создаем итоговый массив

Далее создадим пустой массив под названием *answerArray* (итоговый массив) и заполним его символами подчеркивания (\_), количество которых соответствует количеству букв в загаданном слове.

```
var answerArray = [];  
❶ for (var i = 0; i < word.length; i++) {  
    answerArray[i] = "_";  
}  
  
var remainingLetters = word.length;
```

Answer array —  
массив  
с ответом

Remaining  
letters —  
оставшиеся  
буквы

В строке ❶ в начале цикла `for` создается переменная цикла `i`, которая сначала равна 0, а затем возрастает до `word.length` (не включая, однако, само значение `word.length`). При каждом повторе цикла мы добавляем в массив новый элемент — `answerArray[i]`. Когда цикл завершится, длина `answerArray` будет такой же, как длина слова. Например, если было выбрано слово «макака» (в котором шесть букв), `answerArray` примет вид `["-", "-", "-", "-", "-", "-"]` (шесть знаков подчеркивания).

Наконец, создадим переменную `remainingLetters`, приравняв ее к длине загаданного слова. Эта переменная понадобится, чтобы отслеживать количество букв, которые осталось угадать. Каждый раз, когда игрок угадает букву, мы будем **декрементировать** (то есть уменьшать) значение этой переменной: на 1 для каждого вхождения буквы в слово.

## Программируем игровой цикл

Основа игрового цикла выглядит так:

```
while (remainingLetters > 0) {  
    // Основной код  
    // Показываем состояние игры  
    // Запрашиваем вариант ответа  
    // Обновляем answerArray и remainingLetters для каждого  
    // вхождения угаданной буквы  
}
```



Мы используем цикл `while`, который будет повторяться до тех пор, пока условие `remainingLetters > 0` дает `true`. В теле цикла надо будет обновлять `remainingLetters` для каждого правильного ответа игрока; когда игрок угадает все буквы, `remainingLetters` примет значение 0, и цикл завершится.

Далее мы рассмотрим код, составляющий тело игрового цикла.

Отображение состояния игры

Первым делом в теле игрового цикла нужно показать игроку текущее состояние игры:

```
alert(answerArray.join(" "));
```

Мы делаем это, объединяя элементы `answerArray` в строку с пробелом в качестве разделителя, а затем с помощью `alert` показываем эту строку. Например, пусть загадано слово «макака» и игрок угадал

буквы «м» и «а». Тогда итоговый массив примет вид: `["м", "а", " ", "а", " ", "а"]` и `answerArray.join("")` вернет строку `"ма _ а _ а"`. Диалог `alert` в этом случае будет выглядеть как на рис. 7.4.

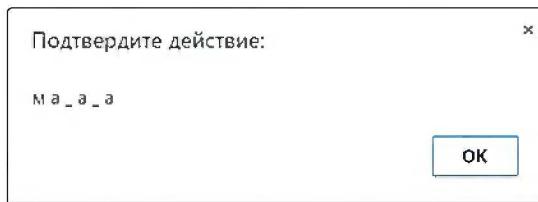


Рис. 7.4. Отображение состояния игры с помощью `alert`

## Обработка введенного ответа

Теперь нужно запросить у игрока ответ и убедиться, что он ввел одиночную букву.

```
❶ var guess = prompt("Угадайте букву или нажмите Отмена для выхода из игры.");
❷ if (guess === null) {
    break;
❸ } else if (guess.length !== 1) {
    alert("Пожалуйста, введите только одну букву.");
} else {
    // Обновляем состояние игры
}
```

**Guess** — предположение

**Break** — здесь «отмена»

В строке ❶ `prompt` запрашивает у игрока ответ и сохраняет его в переменной `guess`. Далее возможен один из четырех вариантов развития событий.

Первый вариант — если игрок нажмет кнопку «Отмена», `guess` примет значение `null`. Этот вариант мы проверяем в строке ❷ командой `if (guess === null)`. Если это условие даст `true`, мы с помощью `break` выйдем из цикла.

**!** Ключевое слово `break` можно использовать для немедленного выхода из любого цикла, независимо от того, где именно внутри цикла это происходит, или от того, выполняется ли на этот момент условие `while`.

Второй и третий варианты — игрок не ввел ничего либо ввел несколько букв. Если он просто нажал «ОК», ничего не вводя, в `guess` окажется пустая строка (`""`), а `guess.length` вернет 0. Если же игрок ввел больше одной буквы, `guess.length` вернет число больше 1.

В строке ❸ мы с помощью `else if (guess.length !== 1)` обрабатываем эти варианты, то есть проверяем, что `guess` содержит в точности

одну букву. В противном случае мы отображаем диалог `alert`, гласящий: «Пожалуйста, введите только одну букву».

Четвертый вариант — игрок, как и положено, ввел одну букву. Тогда мы должны обновить состояние игры — это происходит в строке ❸, в секции `else`. Об этом пойдет речь ниже.

### Обновление состояния игры

Если игрок ввел корректный ответ, мы должны обновить `answerArray` согласно этому ответу. Для этого добавим в тело `else` такой код:

---

```
❶ for (var j = 0; j < word.length; j++) {  
❷   if (word[j] === guess) {  
       answerArray[j] = guess;  
❸       remainingLetters--;  
     }  
}
```

---

В строке ❶ мы задали цикл `for` с новой переменной `j`, которая будет менять значение от 0 до `word.length`, не включая само значение `word.length`. (Мы назвали переменную `j`, поскольку имя `i` уже использовано в предыдущем цикле `for`.) В этом цикле мы проверяем каждую букву переменной `word`. Например, пусть в `word` находится строка "оладушек". Тогда при первом повторе цикла, когда `j` равно 0, `word[j]` вернет "о". При следующем повторе `word[j]` вернет "л", затем "а", "д", "у", "ш", "е" и, наконец, "к".

В строке ❷ мы с помощью `if (word[j] === guess)` проверяем, совпадает ли текущая буква (`word[j]`) с ответом игрока. Если это так, мы обновляем итоговый массив, добавляя туда букву командой `answerArray[j] = guess`. Для каждой буквы, совпадающей с ответом, мы обновляем соответствующую позицию итогового массива. Этот код работает, поскольку переменную цикла `j` можно использовать одновременно в качестве индекса в строке `word` и индекса в массиве `answerArray`, как показано на рис. 7.5.

Индекс ( <code>j</code> )	0	1	2	3	4	5	6	7
<code>word</code>	"о"	л	а	д	у	ш	е	к"
<code>answerArray</code>	["_"]	["_"]	["_"]	["_"]	["_"]	["_"]	["_"]	["_"]

Рис. 7.5. Один и тот же индекс можно использовать для `word` и `answerArray`

Например, представим, что мы только начали игру и дошли до цикла `for` в строке ❶. Пусть загадано слово «программа», в `guess` находится буква "п", а `answerArray` имеет вид:

```
[ " _ ", " _ ", " _ ", " _ ", " _ ", " _ ", " _ ", " _ ", " _ "]
```

При первом повторе `for` в строке ❶ `j` равно 0, поэтому `word[j]` вернет "п". Наш ответ (`guess`) — это "р", поэтому мы пропускаем команду `if` в строке ❷ (ведь условие "п" === "р" дает `false`). При следующем повторе `j` равно 1, и `word[j]` вернет "р". Это значение совпадает с `guess`, и срабатывает оператор `if`. Команда `answerArray[j] = guess` присваивает элементу с индексом 1 (второй элемент) массива `answerArray` значение `guess`, и теперь `answerArray` имеет вид:

```
[ " _ ", "р", " _ ", " _ ", " _ ", " _ ", " _ ", " _ ", " _ "]
```

При следующих двух повторах цикла `word[j]` вернет "о", а затем "г", что не совпадает со значением `guess`. Однако когда `j` достигнет 4, `word[j]` снова вернет "р". И снова мы обновим `answerArray`, на этот раз присвоив значение `guess` элементу с индексом 4 (пятый элемент). Теперь `answerArray` выглядит так:

```
[ " _ ", "р", " _ ", " _ ", "р", " _ ", " _ ", " _ ", " _ "]
```

Оставшиеся буквы не совпадают с "р", так что при дальнейших повторах ничего не произойдет. Так или иначе, после завершения цикла в `answerArray` будут внесены все совпадения `guess` с соответствующими позициями `word`.

Помимо обновления `answerArray` для каждого совпадения с `guess` требуется уменьшать `remainingLetters` на 1. Мы делаем это в строке ❸ командой `remainingLetters--`. Каждый раз, когда `guess` совпадает с буквой из `word`, `remainingLetters` уменьшается на 1, и, когда игрок угадает все буквы, `remainingLetters` примет значение 0.

## Конец игры

Как мы знаем, игровой цикл `while` выполняется при условии `remainingLetters > 0`, поэтому его тело будет повторяться до тех пор, пока еще остаются неотгаданные буквы. Когда же `remainingLetters` уменьшится до 0, цикл завершится. После цикла нам остается лишь закончить игру — это позволяет сделать такой код:

```
alert(answerArray.join(" "));  
alert("Отлично! Было загадано слово " + word);
```



Hangman —  
здесь «Виселица»

В первой строке мы последний раз отображаем итоговый массив. Во второй строке, опять же с помощью `alert`, мы поздравляем игрока с победой.

## Код игры

Итак, мы разобрали по частям весь код игры, осталось лишь соединить все вместе. Ниже он приведен целиком, от начала до конца. Я добавил в него комментарии, поясняющие, что происходит в том или ином месте программы. Обязательно вручную введите код в компьютер — это поможет вам поскорее набить руку в JavaScript. Создайте новый файл под названием `hangman.html` и введите в него следующее:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Виселица!</title>
</head>
<body>
  <h1>Виселица!</h1>

  <script>
    // Создаем массив со словами
    var words = [
      "программа",
      "макака",
      "прекрасный",
      "оладушек"
    ];

    // Выбираем случайное слово
    var word = words[Math.floor(Math.random() * words.length)];

    // Создаем итоговый массив
    var answerArray = [];
    for (var i = 0; i < word.length; i++) {
      answerArray[i] = " ";
    }

    var remainingLetters = word.length;

    // Игровой цикл
    while (remainingLetters > 0) {
      // Показываем состояние игры
      alert(answerArray.join(" "));
    }
  </script>
</body>
</html>
```

```

// Запрашиваем вариант ответа
var guess = prompt("Угадайте букву, или нажмите Отмена для
выхода из игры.");
if (guess === null) {
    // Выходим из игрового цикла
    break;
} else if (guess.length !== 1) {
    alert("Пожалуйста, введите одиночную букву.");
} else {
    // Обновляем состояние игры
    for (var j = 0; j < word.length; j++) {
        if (word[j] === guess) {
            answerArray[j] = guess;
            remainingLetters--;
        }
    }
}

// Конец игрового цикла
}

// Отображаем ответ и поздравляем игрока
alert(answerArray.join(" "));
alert("Отлично! Было загадано слово " + word);
</script>
</body>
</html>

```

---

Если игра не запускается, проверьте, все ли вы ввели правильно. Обнаружить ошибки вам поможет JavaScript-консоль. Например, если вы сделали опечатку в имени переменной, в консоли появится сообщение с указанием, в какой строке ошибка, — примерно такое, как на рис. 7.6.

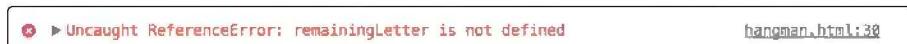
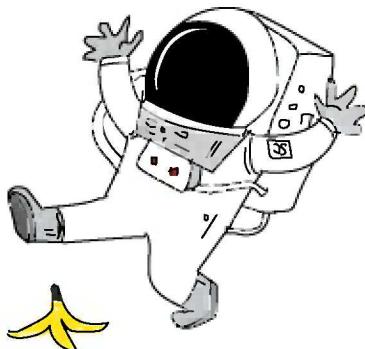


Рис. 7.6. Ошибка JavaScript в консоли Chrome

Кликнув по ссылке `hangman.html:30`, вы увидите строку программы, где произошла ошибка. В данном случае, изучив код, можно понять, что мы напечатали `remainingLetter` вместо `remainingLetters` в условии цикла `while`.

Поиграйте в свою «Виселицу» некоторое время. Работает ли программа так, как вы этого ожидаете? Можете ли вы представить, как выполняется во время игры JavaScript-код?

## Что мы узнали



Не так-то много страниц, а вы уже создали свою первую игру на JavaScript! Как видите, циклы и условные конструкции совершенно необходимы при написании игр и других интерактивных программ; без этих управляющих конструкций programme остается лишь завершиться, едва начав работу.

В восьмой главе мы познакомимся с функциями, что позволит нам запускать один и тот же фрагмент кода из разных мест программы.

## УПРАЖНЕНИЯ

Вот несколько упражнений по усовершенствованию игры «виселица», которую мы написали в этой главе.

### #1. Больше слов

Добавьте новые слова в массив `words`. Не забывайте, что все буквы должны быть строчными.

### #2. Заглавные буквы

Если игрок введет заглавную букву, она не совпадет с такой же строчной буквой в загаданном слове. Эту вероятность можно учесть, преобразовав введенную букву в строчный регистр. (Подсказка: для этого подойдет метод `toLowerCase`.)

### #3. Ограничение по ходам

Сейчас наша «Виселица» позволяет угадывать буквы сколько угодно раз, до победного конца. Добавьте в программу переменную для учета попыток и завершайте игру, если игрок израсходовал все попытки. (Подсказка: проверяйте значение этой переменной в том же цикле `while`, где проверяется условие `remainingLetters > 0`, а с помощью операции `&&` можно убедиться, что сразу два логических условия дают `true` — см. вторую главу.)

### #4. Исправьте ошибку

В игре есть ошибка: если вы будете снова и снова вводить одну и ту же букву, которая есть в загаданном слове, `remainingLetters` будет снова и снова уменьшаться. Постарайтесь это исправить! (Подсказка: можно добавить проверку еще одного условия — что в соответствующем элементе `answerArray` все еще стоит знак подчеркивания. Если там другой символ, значит, эта буква уже угадана.)

# 8

## ФУНКЦИИ

Функции — это механизм для многократного использования частей кода. Они позволяют запускать один и тот же код из разных мест программы без необходимости его копировать. Кроме того, если вы «спрячете» сложные фрагменты кода внутри функций, вам будет легче сосредоточиться на проектировании программы — так вы будете налаживать взаимодействие между функциями, а не барахтаться в мелких деталях, из которых состоит код этих фрагментов. Организация кода в виде небольших, легко контролируемых частей позволяет видеть общую картину и думать о строении программы на более высоком уровне.

Функции очень удобны, когда нужно многократно выполнять в программе некие расчеты или другие действия. Мы уже пользовались готовыми функциями, такими как `Math.random`, `Math.floor`, `alert`, `prompt` и `confirm`. А в этой главе мы научимся создавать свои функции.

### Базовое устройство функции

На рис. 8.1 показано строение функции. Код внутри фигурных скобок называется *телом функции* — аналогично циклам, где код в фигурных скобках зовется *телем цикла*.

```
function () {  
    console.log("Делаем что-то");  
}
```

Тело функции записывается  
в фигурных скобках

Рис. 8.1. Синтаксис создания функции

## Создаем простую функцию

Давайте создадим функцию, которая печатает фразу «Привет, мир!». Введите в консоли браузера следующий код. Чтобы перейти к новой строке без выполнения уже введенных команд, используйте SHIFT-ENTER.

```
var ourFirstFunction = function () {  
    console.log("Привет, мир!");  
};
```

Our first  
function —  
наша первая  
функция

Этот код создает новую функцию, сохраняя ее в переменной `ourFirstFunction`.

## Вызов функции

Чтобы запустить код функции (то есть ее тело), нужно эту функцию вызывать. Для этого укажите ее имя, а следом — открывающую и закрывающую скобки, вот так:

```
ourFirstFunction();  
Привет, мир!
```

При вызове `ourFirstFunction` выполняется ее тело, то есть команда `console.log("Привет, мир!");`, и текст, который мы таким образом выводим, появляется в консоли на следующей строке: "Привет, мир!"

Однако, вызвав эту функцию из браузера, можно заметить в консоли еще одну строчку — с маленькой, указывающей влево стрелкой, как на рис. 8.2. Это значение, которое возвращает функция.

```
> ourFirstFunction();  
Привет, мир!  
<- undefined
```



Рис. 8.2. Вызов функции, возвращаемое значение которой не определено

*Возвращаемое значение* — это значение, которое функция выдает наружу, чтобы потом его можно было использовать где угодно в программе. В данном случае это `undefined`, поскольку мы не указывали возвращаемое значение в теле функции, мы лишь дали команду вывести

текст в консоль. Функция всегда будет возвращать `undefined`, если в теле функции нет указания вернуть что-нибудь другое. (В разделе «Возврат значения из функции» на с. 131 мы выясним, как это можно сделать.)

! В консоли *Chrome* и примерах кода в этой книге возвращаемые значения помечены разными цветами в зависимости от их типа данных, тогда как текст, напечатанный через `console.log`, всегда показан черным цветом.

## Передача аргументов в функцию

Наша функция `ourFirstFunction` выводит одну и ту же строку при каждом вызове, однако хотелось бы, чтобы поведением функции можно было управлять. Чтобы функция могла изменять поведение в зависимости от значений, нам понадобятся *аргументы*. Список аргументов указывается в скобках после имени функции — как при ее создании, так и при вызове.

Say hello to —  
скажи привет  
[кому-то]

Функция `sayHelloTo` использует аргумент `(name)`, чтобы поздороваться с человеком, имя которого передано в аргументе:

```
var sayHelloTo = function (name) {  
    console.log("Привет, " + name + "!");  
};
```

Здесь мы создали функцию и сохранили ее в переменной `sayHelloTo`. При вызове функция печатает строку "Привет, " + name + "!", заменив `name` на значение, переданное в качестве аргумента.

На рис. 8.3 показан синтаксис создания функции с одним аргументом.



Рис. 8.3. Синтаксис создания функции с одним аргументом

Вызывая функцию, которая принимает аргумент, введите значение, которое вы хотите использовать в качестве этого аргумента в скобках после имени функции. Например, чтобы поздороваться с Ником, можно ввести:

```
sayHelloTo("Ник");
```

Привет, Ник!

А с Анной поздороваться можно так:

```
sayHelloTo("Анна");
```

Привет, Анна!

Каждый раз при вызове функции переданный нами аргумент name подставляется в строку, которую печатает функция. Поэтому, когда мы передаем значение "Ник", в консоли появляется "Привет, Ник!", а когда пишем "Анна", мы видим в консоли "Привет, Анна!".

## Печатаем котиков!

Кроме того, переданный в функцию аргумент может указывать, сколько раз требуется что-то сделать. Например, функция drawCats выводит в консоль смайлы — кошачьи мордочки (вот такие: =^.=). Задавая аргумент howManyTimes, мы сообщаем ей, сколько таких смайлов нужно напечатать:

```
var drawCats = function (howManyTimes) {
  for (var i = 0; i < howManyTimes; i++) {
    console.log(i + " =^.^=");
  }
};
```

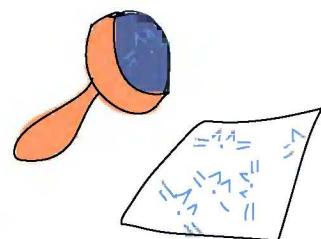
**Draw cats** —  
рисовать  
котов

**How many times** —  
сколько раз

Тело функции представляет собой цикл for, который повторяется столько раз, сколько указано в аргументе howManyTimes (поскольку переменная i сначала равна 0, а затем возрастает до значения howManyTimes - 1). На каждом повторе цикла функция выводит в консоль строку i + « =^.^= ».

Вот что мы увидим, вызвав эту функцию со значением 5 в качестве аргумента howManyTimes:

```
drawCats(5);
0 =^.^=
1 =^.^=
2 =^.^=
3 =^.^=
4 =^.^=
```



Попробуйте задать `howManyTimes` значение 100, чтобы напечатать 100 кошачьих мордочек!

## Передача в функцию нескольких аргументов

В функцию можно передать больше одного значения, задав несколько аргументов. Для этого перечислите аргументы в скобках после имени функции, разделив их запятыми. На рис. 8.4 показан синтаксис создания функции, принимающей два аргумента.

Имена аргументов пишутся  
через запятую

↓

```
function (argument1, argument2) {  
    console.log("Первый аргумент: " + argument1);  
    console.log("Второй аргумент: " + argument2);  
}
```

↑

В теле функции можно использовать  
оба аргумента

Рис. 8.4. Синтаксис создания функции с двумя аргументами

**Print multiple times** —  
напечатать  
многоократно

**What to draw** — что  
рисовать

Функция `printMultipleTimes` похожа на `drawCats`, однако она принимает еще один аргумент с именем `whatToDelete`.

---

```
var printMultipleTimes = function (howManyTimes, whatToDelete) {  
    for (var i = 0; i < howManyTimes; i++) {  
        console.log(i + " " + whatToDelete);  
    }  
};
```

---



Функция `printMultipleTimes` печатает строку, переданную в аргументе `whatToDelete` столько раз, сколько указано в аргументе `howManyTimes`. Второй аргумент сообщает функции, что печатать, а первый — сколько раз это нужно печатать.

Вызывая функцию с несколькими аргументами, перечислите нужные вам значения через запятую в скобках после имени функции. Например, чтобы напечатать кошачьи мордочки с помощью функции `printMultipleTimes`, вызывайте ее так:

---

```
printMultipleTimes(5, "^.^=");  
0 =^.^=  
1 =^.^=
```

---

```
2 =^ .^=
3 =^ .^=
4 =^ .^=
```

---

А чтобы четыре раза напечатать смайлик «^\_^», вызывайте `printMultipleTimes` так:

---

```
printMultipleTimes(4, "^_^");
0 ^ ^
1 ^ ^
2 ^ ^
3 ^ ^
```

---

Здесь при вызове `printMultipleTimes` мы указали значение 4 для аргумента `howManyTimes` и строку «^\_^» для аргумента `whatToDelete`. В результате цикл выполнил четыре повтора (переменная `i` менялась от 0 до 3), каждый раз печатая `i + " " + "^_^"`.

Чтобы дважды напечатать (>\_<), введите:

---

```
printMultipleTimes(2, "(>_<)");
0 (>_<)
1 (>_<)
```

---

На этот раз мы передали число 2 для аргумента `howManyTimes` и строку «(>\_<)» для `whatToDelete`.

## Возврат значения из функции

До сих пор все наши функции выводили текст в консоль с помощью `console.log`. Это простой и удобный способ отображения данных, однако мы не сможем потом взять это значение из консоли и использовать его в коде. Вот если бы наша функция выдавала значение так, чтобы его потом можно было использовать в других частях программы...

Как я уже говорил, функции могут возвращать значение. Вызвав функцию, которая возвращает значение, мы можем затем использовать это значение в своей программе (сохранив его в переменной, передав в другую функцию или объединив с другими данными). Например, следующий код прибавляет 5 к значению, которое возвращает вызов `Math.floor(1.2345)`:

---

```
5 + Math.floor(1.2345);
6
```

---



**Double** — здесь «удвоить»

`Math.floor` — функция, которая берет переданное ей число, округляет его вниз до ближайшего целого значения и возвращает результат. Глядя на вызов функции `Math.floor(1.2345)`, представьте, что вместо него в коде стоит значение, возвращаемое этой функцией, — в данном случае это число 1.

Теперь давайте создадим функцию, которая возвращает значение. Вот функция `double`, которая принимает аргумент `number` и возвращает произведение `number * 2`. Иными словами, значение, которое возвращает эта функция, вдвое больше переданного ей аргумента.

**Return** —  
здесь «выход  
из функции»,  
возвращение

```
var double = function (number) {  
 ❶   return number * 2;  
};
```

Чтобы вернуть из функции значение, используйте оператор `return`, после которого укажите само это значение. В строке ❶ мы воспользовались `return`, вернув из функции `double` число `number * 2`.

Теперь можно вызывать нашу функцию `double` и удваивать числа:

```
double(3);  
6
```

Возвращаемое значение (6) показано здесь второй строкой. Хотя функции и способны принимать несколько аргументов, вернуть они могут лишь одно значение. А если вы не укажете в теле функции, что именно надо возвращать, она вернет `undefined`.

## Вызов функции в качестве значения

Когда функция вызывается из кода программы, значение, возвращаемое этой функцией, подставляется туда, где происходит вызов. Давайте воспользуемся функцией `double`, чтобы удвоить пару чисел и затем сложить результаты:

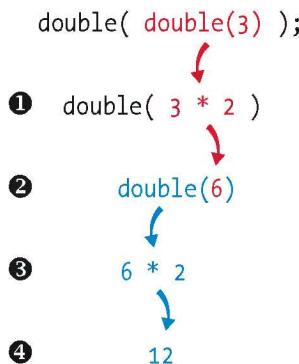
```
double(5) + double(6);  
22
```

Здесь мы дважды вызвали функцию `double` и сложили значения, которые вернули эти два вызова. То же самое было бы, если бы вместо вызова `double(5)` стояло число 10, а вместо `double(6)` — число 12.

Также вызов функции можно указать в качестве аргумента другой функции, при вызове которой в аргумент попадет значение, возвращенное первой функцией. В следующем примере мы вызываем функцию double, передавая ей в качестве аргумента вызов double с аргументом 3. Вызов double(3) даст 6, так что double(double(3)) упрощается до double(6), что, в свою очередь, упрощается до 12.

```
double(double(3));  
12
```

Вот как JavaScript вычисляет это выражение:



Тело функции double возвращает `number * 2`, поэтому в точке ① мы заменяем `double(3)` на `3 * 2`. В точке ② мы заменяем `3 * 2` на `6`. Затем в точке ③ мы делаем то же, заменяя `double(6)` на `6 * 2`. И наконец, в точке ④ мы можем заменить `6 * 2` числом 12.

## Упрощаем код с помощью функций

В третьей главе мы использовали методы `Math.random` и `Math.floor`, чтобы выбирать случайные слова из массивов и генерировать дразнилки. В этом разделе мы перепишем генератор дразнилок, упростив его с помощью функций.

## Функция для выбора случайного слова

Вот как мы выбирали случайное слово из массива в третьей главе:

```
randomWords[Math.floor(Math.random() * randomWords.length)];
```

Pick random word —  
выбрать  
случайное  
слово

Если поместить этот код в функцию, можно многократно вызывать его для получения случайного слова из массива — вместо того чтобы вводить тот же код снова и снова. Например, давайте определим такую функцию `pickRandomWord`:

---

```
var pickRandomWord = function (words) {
    return words[Math.floor(Math.random() * words.length)];
};
```

---

Все, что мы сделали, — поместили прежний код в функцию. Теперь можно создать массив `randomWords`.

---

```
var randomWords = ["Планета", "Червяк", "Цветок", "Компьютер"];
```

---

С помощью функции `pickRandomWord` мы можем получить случайное слово из этого массива, вот так:

---

```
pickRandomWord(randomWords);
"Цветок"
```

---

При этом нашу функцию можно использовать с любым массивом. Например, получить случайное имя из массива имен:

---

```
pickRandomWord(["Чарли", "Радж", "Николь", "Кейт", "Сэнди"]);
"Радж"
```

---

## Генератор случайных дразнилок

Теперь давайте перепишем генератор дразнилок, используя нашу функцию для выбора случайных слов. Для начала вспомним, как выглядел код из третьей главы:

---

```
var randomBodyParts = ["глаз", "нос", "череп"];
var randomAdjectives = ["вонючая", "унывая", "дурацкая"];
var randomWords = ["муха", "выдра", "дубина", "мартышка", "крыса"];
// Выбор случайной части тела из массива randomBodyParts:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Выбор случайного прилагательного из массива randomAdjectives:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];
// Выбор случайного слова из массива randomWords:
var randomWord = randomWords[Math.floor(Math.random() * 5)];
```

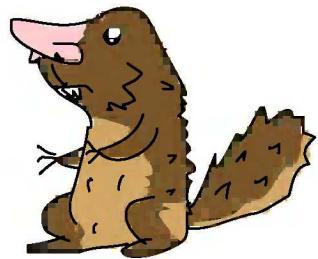
---

```
// Соединяем случайные строки в предложение:  
var randomInsult = "у тебя " + randomBodyPart + " словно " + ↫  
randomAdjective + " " + randomWord + "!!!";  
randomInsult;  
"у тебя нос словно дурацкая выдра!!!"
```

Обратите внимание — конструкция `words[Math.floor(Math.random() * length)]` повторяется здесь несколько раз. Воспользовавшись функцией `pickRandomWord`, можно переписать программу таким образом:

```
var randomBodyParts = ["глаз", "нос", "череп"];  
var randomAdjectives = ["вонючая", "унылая", "дурацкая"];  
var randomWords = ["муха", "выдра", "дубина", "мартышка", "крыса"];  
// Соединяем случайные строки в предложение:  
var randomString = "У тебя " + pickRandomWord(randomBodyParts) + ↫  
" словно " + pickRandomWord(randomAdjectives) + ↫  
" " + pickRandomWord(randomWords) + "!!!";  
randomString;  
"у тебя нос словно дурацкая выдра!!!"
```

Этот код отличается от прежнего двумя моментами. Во-первых, мы использовали функцию `pickRandomWord` для выбора случайного слова из массива вместо того, чтобы каждый раз писать `words[Math.floor(Math.random() * length)]`. А во-вторых, вместо того чтобы сохранять каждое случайное слово в переменной перед тем, как добавлять его к итоговой строке, мы сразу объединяем возвращаемые из функции значения, формируя таким образом строку. Вызов функции можно рассматривать как значение, которое эта функция возвращает, поэтому все, что мы тут делаем, — это объединяем строки. Как видите, новую версию программы гораздо легче читать. Да и писать ее тоже было легче, поскольку часть повторяющегося кода мы вынесли в функцию.



## Делаем генератор дразнилок функцией

Можно еще усовершенствовать наш генератор случайных дразнилок, сделав его функцией, которая возвращает дразнилки:

```
var generateRandomInsult = function () {  
    var randomBodyParts = ["глаз", "нос", "череп"];  
    var randomAdjectives = ["вонючая", "унылая", "дурацкая"];  
    var randomWords = ["муха", "выдра", "дубина", "мартышка", "крыса"];
```

Generate random insult —  
сгенерировать  
случайную  
дразнилку

```
// Соединяем случайные строки в предложение:  
var randomString = "У тебя " + pickRandomWord(randomBodyParts) + ↵  
    " словно " + pickRandomWord(randomAdjectives) + ↵  
    " " + pickRandomWord(randomWords) + "!!!";  
❶ return randomString;  
};  
generateRandomInsult();  
"У тебя череп словно унылая дубина!!!"  
generateRandomInsult();  
"У тебя нос словно дурацкая мартышка!!!"  
generateRandomInsult();  
"У тебя глаз словно воюющая муха!!!"
```

Наша новая функция `generateRandomInsult` представляет собой все тот же код, помещенный в тело функции без аргументов. Мы добавили лишь одну строку, помеченную ❶, где мы возвращаем сгенерированную строку `randomString`. Трижды вызвав функцию `generateRandomInsult`, мы каждый раз получали новую дразнилку.

Теперь весь код находится в функции, и это означает, что для генерации дразнилки мы можем просто вызывать эту функцию, а не копировать в консоль один и тот же код каждый раз, когда понадобится кого-нибудь подразнить.

### Ранний выход из функции по `return`



Как только JavaScript, выполняя код функции, встречает оператор `return`, он завершает функцию, даже если после `return` еще остался какой-нибудь код.

Оператор `return` часто используют, чтобы выйти из функции в самом начале, если какие-нибудь из переданных аргументов имеют некорректные значения — то есть если с такими аргументами функция не сможет правильно работать. Например, следующая функция возвращает строку с информацией о пятой букве вашего имени. Если в имени, переданном в аргументе `name`, меньше пяти

букв, будет выполнен `return`, чтобы сразу же выйти из функции. При этом оператор `return` в конце функции (тот, что возвращает сообщение о пятой букве) так и не будет выполнен.

---

```
var fifthLetter = function (name) {  
❶  if (name.length < 5) {  
❷    return;  
}  
  
  return "Пятая буква вашего имени: " + name[4] + ".";  
};
```

---

В строке ❶ мы проверяем длину переданного имени — уж не короче ли оно пяти символов? Если это так, в строке ❷ мы выполняем `return`, чтобы незамедлительно выйти из функции.

Давайте попробуем эту функцию в деле.

---

```
fifthLetter("Николай");  
"Пятая буква вашего имени: л."
```

---

Fifth letter —  
пятая буква

В имени *Николай* больше пяти букв, так что функция `fifthLetter` благополучно завершается, вернув пятую букву имени *Николай*, то есть *л*. Попробуем вызвать ее еще раз с именем покороче:

---

```
fifthLetter("Ник");  
undefined
```

---

Когда мы вызвали `fifthLetter` для имени *Ник*, функция распознала, что имя недостаточно длинное, и сразу завершилась, выполнив оператор `return` в строке ❷. Поскольку никакого значения после этого `return` не указано, функция вернула `undefined`.

### Многократное использование `return` вместо конструкции `if... else`

Можно многократно использовать `return` внутри разных конструкций `if`, чтобы возвращать из функции разные значения в зависимости от входных данных. Предположим, вы пишете игру, в которой игроки награждаются медалями согласно набранным очкам. Счету меньше трех очков соответствует бронзовая медаль, счету от трех до шести — серебряная, а счету от семи и выше — золотая.



**Medal**  
for score —  
медаль за очки  
**Score** — счет,  
очко (очки)

---

```
var medalForScore = function (score) {
    if (score < 3) {
        ❶      return "Бронзовая";
    }
    ❷    if (score < 7) {
        return "Серебряная";
    }
    ❸    return "Золотая";
};
```

---

В строке ❶ мы возвращаем значение "Бронзовая" и выходим из функции, если счет меньше трех очков. Если мы достигли строки ❷, значит, счет как минимум равен трем очкам, поскольку, будь он меньше трех, мы бы уже вышли из функции (выполнив `return` в первом операторе `if`). И наконец, если мы достигли строки ❸, значит, на счету как минимум семь очков, проверять больше нечего и можно спокойно вернуть значение "Золотая".

Хотя мы проверяем здесь несколько условий, необходимости использовать цепочку конструкций `if... else` нет. Мы используем `if... else`, когда хотим убедиться, что будет выбран лишь один из вариантов. Однако если в каждом варианте выполняется `return`, это также гарантирует однозначный выбор (поскольку выйти из функции можно лишь один раз).

## СОКРАЩЕННАЯ ЗАПИСЬ ПРИ СОЗДАНИИ ФУНКЦИЙ

Есть длинный и короткий способы записи функций. Я использую длинную запись, поскольку она наглядно демонстрирует, что функция хранится в переменной. Тем не менее вам стоит знать и о короткой записи, поскольку ее используют многие JavaScript-разработчики. Возможно, и вы сами, достаточно поработав с функциями, предпочтете короткую запись.

Вот пример длинной записи:

---

```
var double = function (number) {
    return number * 2;
};
```

---

Короткая запись той же функции выглядит так:

---

```
function double(number) {
    return number * 2;
}
```

---

Как видите, при длинной записи мы явно создаем переменную и сохраняем в ней функцию, так что имя `double` записывается прежде ключевого слова `function`. Напротив, при короткой записи сначала идет ключевое слово `function`, а затем название функции. В этом случае JavaScript создает переменную `double` неявным образом.

На техническом сленге длинная запись называется функциональным выражением, а короткая — объявлением функции.

## Что мы узнали

Функции позволяют повторно использовать фрагменты кода. Они могут работать по-разному в зависимости от переданных аргументов и могут возвращать значение в то место кода, откуда они были вызваны. Также функции дают возможность называть фрагменты кода понятными именами, чтобы, глядя на название, мы могли сразу понять, что функция делает.

В следующей главе мы узнаем, как писать JavaScript-код для работы с HTML-элементами.

## УПРАЖНЕНИЯ

Выполните эти упражнения, чтобы попрактиковаться в использовании функций.

### #1. Математические расчеты и функции

Add —  
прибавить  
Multiply —  
умножить

Создайте две функции, `add` и `multiply`; пусть каждая принимает по два аргумента. Функция `add` должна складывать аргументы и возвращать результат, а функция `multiply` — перемножать аргументы.

С помощью только этих двух функций вычислите следующее несложное выражение:

---

36325 \* 9824 + 777

---

### #2. Совпадают ли массивы?

Are arrays  
same —  
одинаковы ли  
массивы

Напишите функцию `areArraysSame`, которая принимает два массива с числами в качестве аргументов. Она должна возвращать `true`, если эти массивы одинаковые (то есть содержат одни и те же числа в одном и том же порядке), или `false`, если массивы различаются. Убедитесь, что ваша функция работает правильно, запустив такой код:

---

```
areArraysSame([1, 2, 3], [4, 5, 6]);  
false  
areArraysSame([1, 2, 3], [1, 2, 3]);  
true  
areArraysSame([1, 2, 3], [1, 2, 3, 4]);  
false
```

---

Подсказка 1: вам понадобится перебрать все значения из первого массива в цикле `for` и убедиться, что они совпадают со значениями из второго массива. Вы можете вернуть `false` прямо из тела `for`, если обнаружите несовпадающие значения.

Подсказка 2: вы можете сразу выйти из функции, пропустив цикл `for`, если у массивов разная длина.

### #3. «Виселица» и функции

Давайте вернемся к игре «Виселица» из седьмой главы и перепишем ее с помощью функций.

Я уже переписал основной код игры. Некоторые его части заменены вызовами функций. Вам осталось лишь запрограммировать эти функции!

```
// Создайте здесь свои функции

// word: загаданное слово
var word = pickWord();
// answerArray: итоговый массив
var answerArray = setupAnswerArray(word);
// remainingLetters: сколько букв осталось угадать
var remainingLetters = word.length;
while (remainingLetters > 0) {
    showPlayerProgress(answerArray);
    // guess: ответ игрока
    var guess = getGuess();
    if (guess === null) {
        break;
    } else if (guess.length !== 1) {
        alert("Пожалуйста, введите одиночную букву.");
    } else {
        // correctGuesses: количество открытых букв
        var correctGuesses = updateGameState(guess, word,
answerArray);
        remainingLetters -= correctGuesses;
    }
}
showAnswerAndCongratulatePlayer(answerArray);
```

Show player progress —  
показать прогресс игрока

Этот вариант с использованием функций почти так же прозрачен, как псевдокод из седьмой главы. Обратите внимание, как функции помогают сделать код более понятным.

Вот объявления функций, код которых вам нужно написать:

```
var pickWord = function () {
    // Возвращает случайно выбранное слово
};

var setupAnswerArray = function (word) {
    // Возвращает итоговый массив для заданного слова word
};

var showPlayerProgress = function (answerArray) {
    // С помощью alert отображает текущее состояние игры
};

var getGuess = function () {
    // Запрашивает ответ игрока с помощью prompt
};

var updateGameState = function (guess, word, answerArray) {
    // Обновляет answerArray согласно ответу игрока (guess)
    // возвращает число, обозначающее, сколько раз буква guess
    // встречается в слове, чтобы можно было обновить значение
    // remainingLetters
};

var showAnswerAndCongratulatePlayer = function (answerArray) {
    // С помощью alert показывает игроку отгаданное слово
    // и поздравляет его с победой
};
```

Show answer and congratulate player —  
показать ответ и поздравить игрока



ЧАСТЬ II

# Продвинутый JavaScript



# 9

## DOM И JQUERY

До сих пор мы использовали JavaScript для относительно простых задач, вроде вывода текста в консоль браузера или отображения диалогов `alert` и `prompt`. Однако помимо этого JavaScript позволяет взаимодействовать с HTML-элементами на веб-страницах, меняя их поведение и внешний вид. В этой главе мы поговорим о двух технологиях, которые помогут вам писать гораздо более мощный JavaScript-код: это DOM и jQuery.

DOM — это средство, позволяющее JavaScript-коду взаимодействовать с содержимым веб-страниц. Браузеры используют DOM для структурирования страниц и их элементов (параграфов, заголовков и т. д.), а JavaScript может разными способами манипулировать элементами DOM. Например, скоро вы узнаете, как при помощи JavaScript-программы менять заголовки HTML-документов, подставляя туда значение, полученное из диалога `prompt`.

Также мы познакомимся с удобным инструментом под названием jQuery, который кардинально упрощает работу с DOM. jQuery содержит набор функций, которые позволяют найти нужные вам элементы и проповеди с ними определенные действия.

В этой главе мы узнаем, как с помощью DOM и jQuery изменять существующие элементы DOM, а также создавать новые, полностью контролируя содержимое веб-страниц из JavaScript-кода. Также мы выясним, как использовать jQuery для анимации элементов DOM — например, для плавного появления и исчезновения изображений и текста.

DOM —  
Document  
Object Model —  
объектная  
модель  
документа

## Поиск элементов DOM

Когда вы открываете HTML-документ, браузер преобразовывает его элементы в древовидную структуру — дерево *DOM*. На рис. 9.1 изображено простое дерево DOM — мы уже встречали его в пятой главе, когда говорили об иерархии HTML. Браузер дает JavaScript-программистам возможность доступа к этой древовидной структуре при помощи специальных методов DOM.

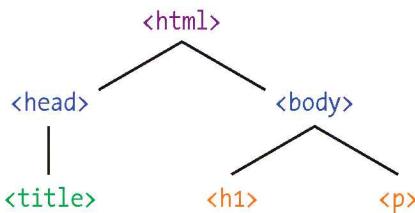


Рис. 9.1. DOM-дерево простого HTML-документа,  
наподобие созданного нами в пятой главе

## Идентификация элементов по id

HTML-элементу можно присвоить уникальное имя, или *идентификатор*, с помощью атрибута `id`. Например, у элемента `h1` задан атрибут `id`:

Main  
heading —  
главный  
заголовок

---

```
<h1 id="main-heading">Привет, мир!</h1>
```

---

Задав атрибуту `id` значение (в данном случае "main-heading"), мы получаем возможность впоследствии найти этот конкретный заголовок по его `id` и что-нибудь с ним сделать, не затрагивая остальные элементы, даже если это другие заголовки уровня `h1`.

Get element  
by ID —  
получить  
элемент по ID

## Поиск элемента с помощью `getElementById`

Обозначив элемент уникальным `id` (каждый `id` в документе должен иметь собственное, отличное от других значение), мы можем воспользоваться DOM-методом `document.getElementById`, чтобы найти элемент "main-heading":

---

```
var headingElement = document.getElementById("main-heading");
```

---

Вызовом `document.getElementById("main-heading")` мы даем браузеру команду отыскать элемент, `id` которого равен "main-heading".

Этот вызов вернет DOM-объект с соответствующим id, и мы сохраним этот объект в переменной headingElement.

Когда элемент найден, им можно управлять при помощи JavaScript-кода. Например, через свойство innerHTML мы можем узнать, что за текст находится внутри элемента, или заменить этот текст:

```
headingElement.innerHTML;
```

Эта команда возвращает содержимое headingElement — элемента, который мы нашли с помощью getElementById. В данном случае содержимое — это текст "Привет, мир!", находящийся между тегов <h1>.

## Меняем текст заголовка через DOM

Вот пример того, как менять текст заголовка с помощью DOM. Давайте создадим новый HTML-документ *dom.html* со следующим кодом:

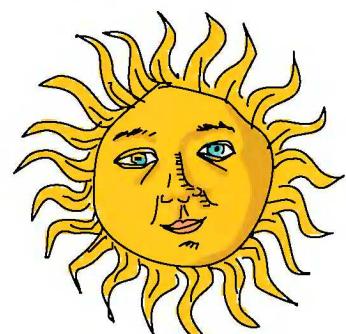
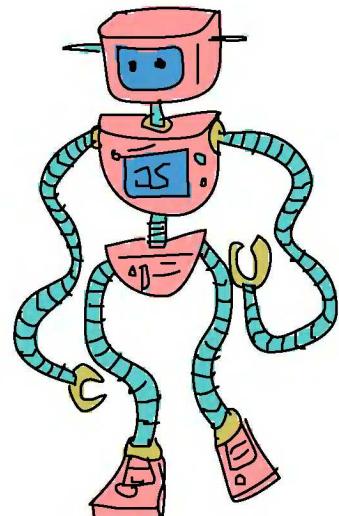
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Изучаем DOM</title>
</head>

<body>
    <h1 id="main-heading">Привет, мир!</h1>

    <script>
        ❶ var headingElement = document.getElementById("main-heading");
        ❷ console.log(headingElement.innerHTML);
        ❸ var newHeadingText = prompt("Введите новый заголовок:");
        ❹ headingElement.innerHTML = newHeadingText;
    </script>
</body>
</html>
```

New heading  
text — новый  
текст  
заголовка

В строке ❶ мы с помощью document.getElementById нашли элемент h1 (id которого равен «main-heading») и сохранили его в переменной headingElement. В строке ❷ мы вывели в консоль строку, возвращенную вызовом headingElement.innerHTML — то есть "Привет, мир!". В строке ❸ открыли диалог prompt, чтобы получить от пользователя новый заголовок, и сохранили введенный



пользователем текст в переменной newHeadingText. И наконец, в строке ④ присвоили сохраненное в newHeadingText значение свойству innerHTML элемента headingElement.

Открыв этот документ в браузере, вы должны увидеть диалог prompt, как на рис. 9.2.

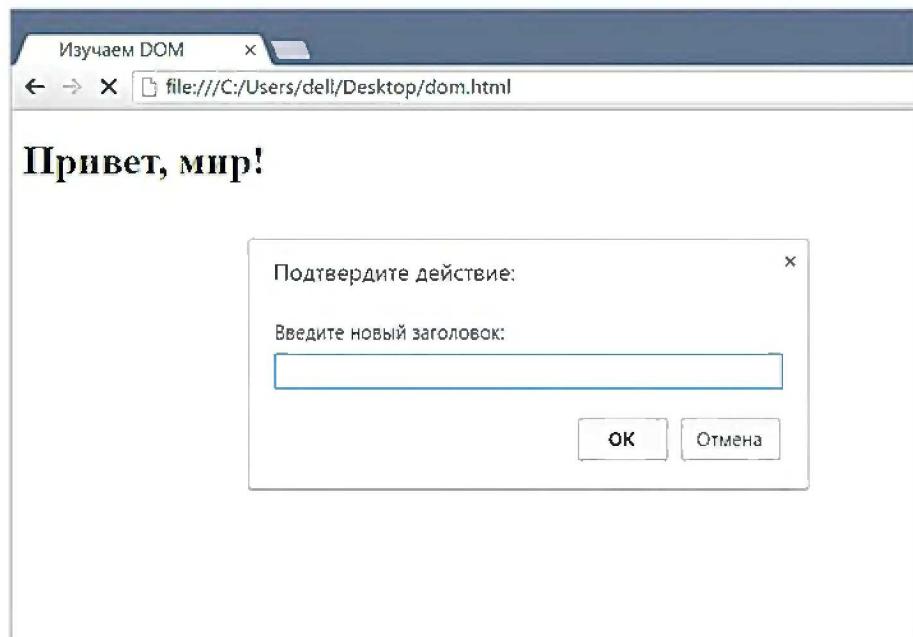


Рис. 9.2. Наша страничка с диалогом prompt

Ведите в этом диалоге строку «JAVASCRIPT ЭТО ЗДОРОВО» и нажмите «OK». Заголовок должен тотчас поменяться — как на рис. 9.3.



Рис. 9.3. Наша страничка после смены заголовка

Обращаясь к свойству innerHTML, можно поменять содержимое любого элемента DOM через JavaScript-код.

## Работа с деревом DOM через jQuery

Встроенные в браузер методы DOM всем хороши, но пользоваться ими бывает нелегко, поэтому многие программисты применяют специальную библиотеку под названием jQuery.

jQuery — это набор инструментов (в основном функций), которые сильно упрощают работу с DOM-элементами. Подключив эту библиотеку к нашей страничке, мы сможем вызывать ее функции и методы в дополнение к функциям и методам, встроенным в JavaScript и в браузер.

### Подключаем jQuery к HTML-странице

Прежде чем воспользоваться библиотекой jQuery, нужно, чтобы браузер ее загрузил, для чего достаточно одной строки HTML-кода:

```
<script src="https://code.jquery.com/jquery-2.1.0.js"></script>
```

Src — source — источник

Обратите внимание, что у тега `<script>` нет содержимого, зато есть атрибут `src`. Этот атрибут позволяет загрузить на страницу JavaScript-файл, указав его URL (то есть веб-адрес). В данном случае `https://code.jquery.com/jquery-2.1.0.js` — это URL конкретной версии jQuery (2.1.0) на сайте jQuery.

Введя этот адрес в строке браузера, вы увидите JavaScript-код, который будет загружен, если добавить на страницу тег `<script>` с указанным выше атрибутом `src`. Библиотека состоит из примерно 9000 строк JavaScript-кода, поэтому не надейтесь с ходу разобраться, как она устроена!

### Меняем текст заголовка с помощью jQuery

В разделе «Меняем текст заголовка через DOM» на с. 147 мы выяснили, как поменять заголовок, вызывая встроенные методы DOM. В этом же разделе мы доработаем код страницы, чтобы менять заголовок через jQuery. Откройте файл `dom.html` и внесите в него следующие правки:

```
!DOCTYPE html
<html>
<head>
  <meta charset="UTF-8">
  <title>Изучаем DOM</title>
</head>

<body>
  <h1 id="main-heading">Привет, мир!</h1>
  ①   <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
```

```
<script>
var newHeadingText = prompt("Введите новый заголовок:");
② $( "#main-heading" ).text(newHeadingText);
</script>
</body>
</html>
```

---

В строке ① мы добавили новый тег `<script>`, подгружающий jQuery. Когда библиотека загружена, можно использовать jQuery-функцию `$` для поиска HTML-элементов.

Функция `$` принимает один аргумент, который называется *строка селектора*. Эта строка указывает, какой элемент или элементы нужно найти в дереве DOM. В нашем случае это `"#main-heading"`. Символ `#` означает `id`, то есть селектор `"#main-heading"` ссылается на элемент, `id` которого равен `main-heading`.

Функция `$` возвращает объект jQuery, соответствующий найденным элементам. Например, вызов `$( "#main-heading" )` вернет jQuery-объект для элемента `h1` (`id` которого равен `"main-heading"`).

Теперь, когда у нас есть объект jQuery для элемента `h1`, мы можем изменить его текст, вызвав в строке ② метод jQuery-объекта `text` с новым заголовком в качестве аргумента. Заголовок поменяется на введенную пользователем строку, которая хранится в переменной `headingText`. Как и прежде, при открытии нашей страницы должен появиться диалог с запросом нового текста для элемента `h1`.

## Создание новых элементов через jQuery

**Append** —  
добавление  
записи

Помимо изменения существующих элементов, с помощью jQuery можно создавать новые элементы и добавлять их в дерево DOM. Для этого мы будем использовать метод jQuery-объекта `append`, передавая ему нужный HTML-код. Append преобразует HTML в DOM-элемент (соответствующий заданным в коде тегам) и добавит его к содержимому элемента, для которого он был вызван.

Например, чтобы поместить в конец страницы новый элемент `p`, добавим в наш JavaScript такой код:

```
$( "body" ).append("<p>Это новый параграф</p>");
```

---

Первая часть этой команды вызывает функцию `$` со строкой селектора `"body"`, чтобы найти тело (содержимое) нашего HTML-документа. Поиск не обязательно должен происходить по `id` — код `$( "body" )` ищет элемент `body`, и точно так же мы можем вызвать `$( "p" )` для поиска всех элементов `p`.

Далее мы вызываем для найденного объекта метод `append` — переданная ему строка преобразуется в DOM-элемент, а затем добавляется

внутрь элемента `body`, сразу перед закрывающим тегом. На рис. 9.4 показано, как будет выглядеть после этого наша страничка.

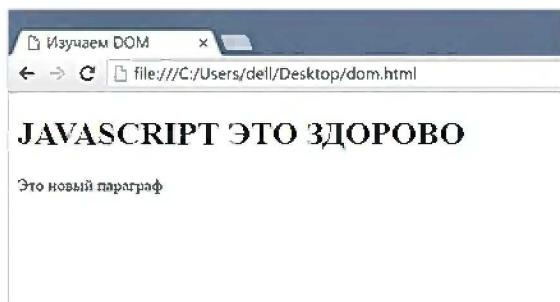


Рис. 9.4. Наш документ с добавленным элементом

Также `append` можно использовать в цикле `for` для добавления нескольких элементов:

```
for (var i = 0; i < 3; i++) {  
    var hobby = prompt("Назови одно из своих хобби!");  
    $("body").append("<p>" + hobby + "</p>");  
}
```

Hobby — хобби

Этот цикл повторяется трижды. При каждом повторении создается диалог `prompt`, запрашивающий у пользователя его хобби, после чего строка с хобби помещается между тегов `<p>` и передается методу `append`, который добавляет ее в конец элемента `body`. Введите этот код в наш документ `dom.html` и загрузите его в браузер. Результат должен выглядеть как на рис. 9.5.

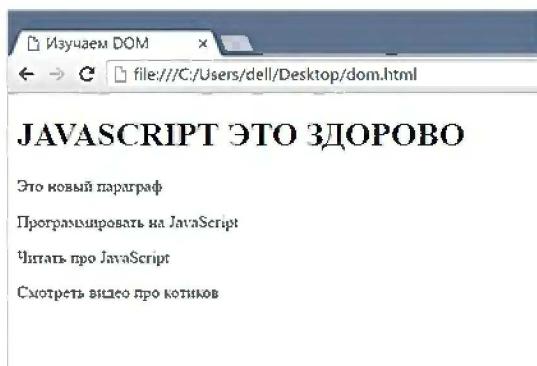


Рис. 9.5. Элементы, добавленные в цикле

**Fade out** —  
исчезать

## Анимация элементов средствами jQuery

На многих сайтах при показе и скрытии частей страницы используется анимация. Например, добавляя на страницу новый параграф с текстом, вы можете сделать так, чтобы он проявлялся постепенно, а не весь целиком сразу.

С помощью jQuery анимировать элементы совсем не сложно. К примеру, чтобы элемент медленно исчезал, мы можем воспользоваться методом `fadeOut`. Замените содержимое второго элемента `script` в `dom.html` на такой код:

```
$( "h1" ).fadeOut(3000);
```

Чтобы найти все элементы `h1`, мы использовали функцию `$`. Поскольку в `dom.html` элемент `h1` всего один (это заголовок с текстом «Привет, мир!»), именно его мы и получим в виде jQuery-объекта. Вызывая для этого объекта метод `fadeOut(3000)`, мы запускаем затухание заголовка до полного его исчезновения в течение трех секунд. (Аргумент `fadeOut` передается в миллисекундах, то есть тысячных долях секунды, поэтому значение 3000 даст анимацию в три секунды длиной.)

Когда вы перегрузите страницу с этим кодом, вы увидите, как элемент `h1` постепенно исчезает.

## Цепной вызов и анимация на jQuery

Если вызвать метод jQuery-объекта, этот метод, как правило, вернет первоначальный объект — тот, для которого он и был вызван. Например, `$( "h1" )` возвращает jQuery-объект со всеми элементами `h1`, а `$( "h1" ).fadeOut(3000)` возвращает *все тот же* jQuery-объект с элементами `h1`. Тогда изменить текст заголовка и включить его затухание можно так:

```
$( "h1" ).text("Этот текст скоро исчезнет").fadeOut(3000);
```

Подобный вызов нескольких методов подряд называют *цепным вызовом*.

**Fade in** —  
постепенно  
усиливаться

Можно запустить несколько анимаций одного и того же элемента. Например, использовать цепной вызов методов `fadeOut` и `fadeIn`, чтобы элемент исчез и тут же снова проявился:

```
$( "h1" ).fadeOut(3000).fadeIn(2000);
```

Анимация `fadeIn` заставляет невидимый элемент проявиться. jQuery понимает, что, когда вы делаете цепной вызов двух анимаций, вы, скорее всего, хотите, чтобы они сработали по очереди, одна после другой. В результате элемент `h1` будет затухать в течение трех секунд, а затем в течение двух секунд проявляться.

В jQuery есть еще два метода для анимации, похожие на `fadeOut` и `fadeIn`, — это `slideUp` и `slideDown`. При вызове `slideUp` элементы исчезают, упывая вверх, а при вызове `slideDown` появляются, опускаясь сверху. Замените второй элемент `script` в `dom.html` на следующий код и перегрузите страницу:

---

```
$( "h1" ).slideUp(1000).slideDown(1000);
```

---

Здесь мы нашли элемент `h1`, скрыли его с эффектом упывания вверх в течение одной секунды, а затем показали снова, опустив сверху вниз за одну секунду.

### ПОПРОБУЙТЕ!

Мы применяли `fadeIn`, чтобы показывать невидимые элементы. Но что если вызвать `fadeIn` для элемента, который уже видим, и что будет с элементом, который следует за элементом, находящимся в процессе анимации?

К примеру, добавьте в `dom.html` новый элемент `p` сразу после заголовка. Используйте `slideUp` и `slideDown`, чтобы скрыть и показать элемент `h1`, и посмотрите, как поведет себя элемент `p`. А если использовать `fadeOut` и `fadeIn`?

Что произойдет, если использовать `fadeOut` и `fadeIn` для одного и того же элемента, но без цепного вызова? Например:

---

```
$( "h1" ).fadeOut(1000);
$( "h1" ).fadeIn(1000);
```

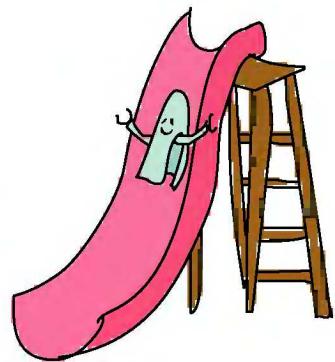
---

Попробуйте поместить этот код внутрь цикла `for`, настроенного на пять повторов. Что получится?

Как считаете, что делают jQuery-методы `show` и `hide`? Проверьте свои догадки на практике. Как может пригодиться `hide`, если нужно, чтобы уже видимый элемент постепенно проявился?

**Slide up** —  
здесь  
«поднять»

**Slide down** —  
здесь  
«опустить»



## Что мы узнали

В этой главе мы узнали, как модифицировать HTML-страницы из JavaScript, работая с элементами DOM. Как видите, jQuery дает более мощные способы поиска элементов, а также их изменения и даже анимации. Также мы узнали об HTML-атрибуте `id`, который позволяет присвоить элементу уникальный идентификатор.

В следующей главе мы выясним, как управлять моментом запуска JavaScript-кода — допустим, чтобы он выполнялся по таймеру или при нажатии на кнопку. Также мы разберемся, как выполнять фрагмент кода многократно, с паузой между запусками — например, чтобы раз в секунду обновлять показания времени.

## УПРАЖНЕНИЯ

Выполните эти упражнения, чтобы попрактиковаться в использовании jQuery и DOM.

### #1. Перечислите своих друзей (и сделайте их лучшими!)

Создайте массив с именами нескольких друзей. В цикле `for` создайте для каждого имени по одному элементу `p` и добавьте эти элементы в конец `<body>`, вызывая jQuery-метод `append`. С помощью jQuery измените текст элемента `h1`, чтобы вместо "Привет, мир!" там было "Мои друзья". Используйте метод `hide` и метод `fadeIn`, чтобы имена плавно возникали на экране.

Теперь измените созданные вами элементы `p`, добавив после каждого имени слово "лучший!". Подсказка: если найти сразу все элементы `p` с помощью `$(“p”)`, метод `append` можно вызывать для них всех разом.

### #2. Мигающий заголовок

Как с помощью `fadeOut` и `fadeIn` сделать так, чтобы заголовок мигнул пять раз с интервалом в секунду? Как сделать это в цикле `for`? А теперь измените цикл, чтобы заголовок появлялся и исчезал в первый раз за секунду, потом за две, потом за три и т. д.

### #3. Отложенная анимация

Для задержки анимации можно воспользоваться методом `delay`. С помощью `delay`, `fadeOut` и `fadeIn` заставьте какой-нибудь элемент плавно исчезнуть, а затем, через пять секунд, снова проявиться.

**Delay** —  
отложить

### #4. Метод `fadeTo()`

Поэкспериментируйте с методом `fadeTo`. Первый его аргумент — число миллисекунд, как и у прочих методов анимации, а второй — число от 0 до 1. Что произойдет, если запустить следующий код?

---

```
$(“h1”).fadeTo(2000, 0.5);
```

---

Как думаете, что делает второй аргумент? Попробуйте разные его значения в диапазоне от 0 до 1, чтобы выяснить, зачем он нужен.

# 10

## ИНТЕРАКТИВНОЕ ПРОГРАММИРОВАНИЕ

До сих пор JavaScript начинал работу сразу же после загрузки страницы, приостанавливаясь лишь при вызове некоторых функций, таких как `alert` или `confirm`. Однако порой не нужно выполнять весь код сразу — что если мы хотим запустить фрагмент кода спустя какое-то время или в ответ на действие пользователя?

В этой главе мы изучим разные способы управлять тем, когда именно выполняется наш код. Это называется *интерактивным программированием*. Оно позволяет создавать интерактивные веб-страницы, которые могут изменяться со временем и реагировать на действия пользователей.

### Отложенное выполнение кода и `setTimeout`

Вместо того чтобы вызывать функцию сразу, можно попросить JavaScript сделать это спустя определенное время. Такого рода отложенное выполнение называется *запуском по таймеру*, и для этого в JavaScript есть функция `setTimeout`. Данная функция принимает два аргумента (см. рис. 10.1): функцию, которую надо будет вызвать при срабатывании таймера, и само время ожидания в миллисекундах.

**Set timeout** —  
установить время  
задержки

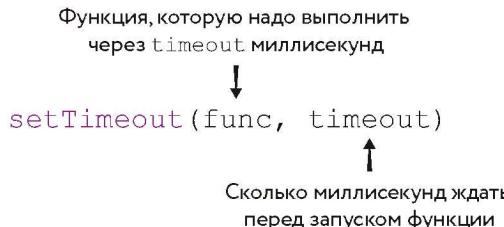


Рис. 10.1. Аргументы `setTimeout`

Следующий пример показывает, как открыть диалог `alert` через `setTimeout`.

---

```

❶ var timeUp = function () {
    alert("Время вышло!");
};

❷ setTimeout(timeUp, 3000);
  1

```

---

В строке ❶ мы создали функцию `timeUp`, открывающую диалог `alert` с сообщением "Время вышло!". В строке ❷ мы вызвали `setTimeout` с двумя аргументами: функцией, которую нужно запустить (`timeUp`), и числом миллисекунд (3000), которые должны пройти перед ее запуском. По сути, мы говорим: «Подожди три секунды и вызови `timeUp`». Сразу после вызова `setTimeout(timeUp, 3000)` ничего не произойдет, однако через три секунды сработает функция `timeUp`, открыв диалог `alert`.

Обратите внимание — вызов `setTimeout` вернул число 1. Это значение называют идентификатором (ID) таймера, который обозначает этот конкретный таймер (отложенный вызов функции). Заметим, что возвращаемое `setTimeout` значение может быть любым числом, ведь это просто идентификатор. Вызовите `setTimeout` снова, и он вернет другой ID таймера:

---

```

setTimeout(timeUp, 5000);
  2

```

---

Полученный ID можно передать функции `clearTimeout`, чтобы отменить этот конкретный таймер. Об этом я расскажу ниже.

**Clear timeout** —  
отменить  
задержку

## Отмена действия таймера



После задания отложенного вызова функции с помощью `setTimeout` может выясниться, что вызывать эту функцию больше не нужно. Представьте, что вы поставили будильник, чтобы он напомнил вам о домашнем задании, однако в итоге сделали все заранее и теперь хотите отключить будильник. Для отмены действия таймера используется функция `clearTimeout` с ID таймера (полученным ранее от `setTimeout`) в качестве аргумента. Предположим, вы установили таймер «сделай домашку» следующим образом:

Do homework  
alarm — здесь  
будильник  
«сделай  
домашку»

```
var doHomeworkAlarm = function () {  
    alert("Эй! Пора делать домашку!");  
};  
① var timeoutId = setTimeout(doHomeworkAlarm, 60000);
```

Функция `doHomeworkAlarm` создает диалог `alert`, напоминающий о домашке. Вызов `setTimeout(doHomeworkAlarm, 60000)` сообщает JavaScript, что функцию `doHomeworkAlarm` нужно вызвать через 60 000 миллисекунд (то есть 60 секунд). В строке **①** мы вызвали `setTimeout` и сохранили ID таймера в новой переменной `timeoutID`.

Теперь, чтобы отменить действие таймера, достаточно передать его ID функции `clearTimeout`, вот так:

```
clearTimeout(timeoutId);
```

Теперь `setTimeout` не будет вызывать функцию `doHomeworkAlarm`.

## Многократный запуск кода и `setInterval`

Set interval —  
задать  
интервал

Функция `setInterval` похожа на `setTimeout`, однако она вызывает переданную ей функцию повторно через определенные промежутки (*интервалы*) времени. Скажем, если вы хотите с помощью JavaScript обновлять показания часов, используйте `setInterval`, чтобы функция обновления вызывалась раз в секунду. `SetInterval` принимает два аргумента: функцию и интервал времени в миллисекундах, как показано на рис. 10.2.



Рис. 10.2. Аргументы `setInterval`

Например, так можно раз в секунду выводить в консоль сообщение:

---

```

❶ var counter = 1;                                Counter —
                                                    счетчик

❷ var printMessage = function () {                Print
    console.log("Ты смотришь в консоль уже " + counter + " сек");   message —
❸   counter++;                                    напечатать
};                                                 сообщение

❹ var intervalId = setInterval(printMessage, 1000);  

    Ты смотришь в консоль уже 1 сек  

    Ты смотришь в консоль уже 2 сек  

    Ты смотришь в консоль уже 3 сек  

    Ты смотришь в консоль уже 4 сек  

    Ты смотришь в консоль уже 5 сек  

    Ты смотришь в консоль уже 6 сек
❺ clearInterval(intervalId);

```

---

В строке ❶ мы создали новую переменную `counter` и присвоили ей значение 1. С помощью этой переменной мы будем вести учет времени (в секундах).

В строке ❷ мы создали функцию `printMessage`, которая выполняет две задачи. Во-первых, она печатает сообщение о том, сколько секунд вы уже смотрите в консоль. Во-вторых, далее в строке ❸ она увеличивает переменную `counter`.

Затем, в строке ❹, мы вызвали `setInterval`, передав ей функцию `printMessage` и число 1000, что означает «вызывай `printMessage` каждые 1000 миллисекунд». Так же как `setTimeout` возвращает ID таймера, `setInterval` возвращает *ID* *интервала*, который мы сохранили в переменной `intervalId`. Далее этот ID можно использовать для отмены периодического вызова функции `printMessage` — что мы и сделали в строке ❺ с помощью функции `clearInterval`.



## ПОПРОБУЙТЕ!

Измените предыдущий пример так, чтобы сообщение выводилось каждые пять секунд, а не раз в секунду.

## Анимация элементов с помощью setInterval

Отложенный вызов через `setInterval` можно использовать для анимации элементов в браузере. По сути, для этого нужно создать функцию, которая слегка сдвигает элемент, и затем передать ее `setInterval`, установив небольшое время повтора. При условии, что каждый сдвиг будет достаточно мал и величина интервала тоже, анимация получится очень плавной.

**Interactive** —  
интерактивный

Давайте анимируем положение фрагмента текста в HTML-документе, двигая его по горизонтали. Создайте файл `interactive.html` с таким содержимым:

---

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Интерактивное программирование</title>
</head>

<body>
    <h1 id="heading">Привет, мир!</h1>

    <script src="https://code.jquery.com/jquery-2.1.0.js"> ←
    </script>

    <script>
        // Скоро здесь будет JavaScript-код
    </script>
</body>
</html>
```

---

Теперь перейдем к JavaScript-коду. Как и прежде, поместите его в HTML-документ между тегами `<script>`.

**Left offset** —  
отступ слева  
**Move heading** —  
подвинуть  
заголовок

---

```
❶ var leftOffset = 0;

❷ var moveHeading = function () {
❸     $("#heading").offset({ left: leftOffset });
❹     leftOffset++;
❺     if (leftOffset > 200) {
```

```
    leftOffset = 0;  
}  
};  
  
❶ setInterval(moveHeading, 30);
```

---

Открыв наш документ в браузере, вы увидите, как элемент заголовка плавно сдвигается вправо, пока не пройдет расстояние в 200 пикселей. Затем он резко вернется на свое место, и все начнется снова. Разберемся, как это работает.

В строке ❶ мы создали переменную `leftOffset` (отступ слева), которой далее воспользуемся для задания позиции заголовка «Привет, мир!». Ее начальное значение — 0. Это значит, что заголовок начнет свое движение с левого края страницы.

Далее в строке ❷ мы создали функцию `moveHeading`, чтобы вызывать ее через `setInterval`. В коде `moveHeading`, в строке ❸, мы используем `$("#heading")` для поиска элемента с `id "heading"` (это элемент `h1`) и вызываем метод `offset`, чтобы задать смещение заголовка от левого края экрана, сдвигая его вправо.

Метод `offset` принимает объект, который может содержать свойство `left` для задания смещения от левого края или свойство `top` для смещения от верха страницы. В данном случае мы выбрали свойство `left` и присвоили ему значение переменной `leftOffset`. Если бы требовалось задать неизменное смещение, мы могли бы указать для свойства числовое значение. Скажем, вызов `$("#heading").offset({ left: 100 })` поместит заголовок на расстоянии 100 пикселей от левого края страницы.

В строке ❹ мы увеличили `leftOffset` на 1. Чтобы убедиться, что заголовок не уполз слишком далеко, в строке ❺ выполняется проверка — `leftOffset` больше 200? Если это так, сбрасываем значение до 0. Наконец, в строке ❻ мы вызвали `setInterval`, передав ей в качестве аргументов функцию `moveHeading` и число 30 (что означает 30 миллисекунд).

Этот код вызывает `moveHeading` каждые 30 миллисекунд, то есть примерно 33 раза в секунду. При каждом вызове `moveHeading` переменная `leftOffset` увеличивается, и далее значение этой переменной используется, чтобы задать положение заголовка. Поскольку функция вызывается периодически, а `leftOffset` каждый раз увеличивается на 1, заголовок плавно движется по странице, смещааясь на 1 пиксель каждые 30 миллисекунд.

## ПОПРОБУЙТЕ!

Вы можете ускорить анимацию, увеличив приращение `leftOffset` в функции `moveHeading` либо уменьшив время между вызовами `moveHeading`.

Как удвоить скорость движения заголовка? Попробуйте двумя способами. Чем отличаются результаты?

## Реакция на действия пользователя

Как мы выяснили, один из способов управления временем запуска кода — применение функций `setTimeout` и `setInterval`, которые вызывают функцию через фиксированный промежуток времени. Другой способ — выполнять код при определенных действиях пользователя, таких как клик мышкой, нажатие клавиши или просто перемещение мышки. Тогда пользователи смогут взаимодействовать с вашей страницей, получая соответствующий отклик.

Каждый раз, когда вы совершаете действие — кликаете, вводите текст или двигаете мышку, — в браузере возникает нечто под названием *событие*. Это способ, которым браузер сообщает «случилось вот это». На события можно подписываться, добавляя *обработчик события* к элементу, в котором это событие происходит. Добавляя обработчик, вы говорите JavaScript: «Если произойдет это событие, вызови эту функцию». Например, если вы хотите вызывать функцию при клике по заголовку, добавьте к элементу заголовка обработчик события. Сейчас мы разберемся, как это делается.

## Реакция на клики

**Click** — клик

Когда пользователь кликает по элементу в браузере, возникает событие `click`. С помощью jQuery ничего не стоит задать этому событию обработчик. Откройте созданный ранее файл `interactive.html`, выберите File → Save As, чтобы сохранить его под именем `clicks.html`, и замените содержимое второго элемента `script` таким кодом:

**Click handler** — обработчик клика

---

```
❶ var clickHandler = function (event) {  
❷   console.log("Клик! " + event.pageX + " " + event.pageY);  
};  
  
❸ $("h1").click(clickHandler);
```

---

В строке ❶ мы создали функцию `clickHandler` с единственным аргументом `event`. При ее вызове в аргументе `event` будет передан объект, содержащий информацию о событии, например о том, в каком месте был сделан клик. В строке ❷, в коде функции-обработчика, мы использовали `console.log` для вывода свойств `pageX` и `pageY` объекта `event`. Эти свойства хранят *x*- и *y*-координаты события — иными словами, они сообщают, где именно на странице произошел клик.

Наконец, в строке ❸ мы активировали обработчик кликов. Код `$("#h1")` находит элемент `h1`, а вызов `$("#h1").click(clickHandler)` означает: «В случае клика по элементу `h1` вызови функцию `clickHandler` и передай ей объект события». В данном случае обработчик извлекает из объекта `event` информацию об *x*- и *y*-координатах клика.

Перезагрузите нашу модифицированную страницу в браузере и кликните по элементу заголовка. При каждом клике в консоли должна появиться новая строка, как показано ниже. Каждая из строк оканчивается двумя числами:  $x$ - и  $y$ -координатами клика.

---

```
Клик! 88 43
Клик! 63 53
Клик! 24 53
Клик! 121 46
Клик! 93 55
Клик! 103 48
```

---



### КООРДИНАТЫ В БРАУЗЕРЕ

В веб-браузере, как и в большинстве других сред графического программирования, нулевая позиция  $x$ - и  $y$ -координат находится в верхнем левом углу экрана. По мере роста  $x$ -координаты точки смещаются к правому краю страницы, а по мере роста  $y$ -координаты — к низу страницы (рис. 10.3).

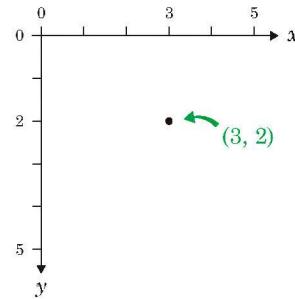


Рис. 10.3. Система координат в браузере, показан клик в координатах  $(3, 2)$

### Событие mousemove

Событие `mousemove` возникает всякий раз при перемещении мышки. Создайте файл с именем `mousemove.html` и введите туда следующий код:

**Mouse move** —  
перемещение  
мышки

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Перемещение мышки</title>
</head>

<body>
    <h1 id="heading">Привет, мир!</h1>

    <script src="https://code.jquery.com/jquery-2.1.0.js"></script>

    <script>
        $("html").mousemove(function (event) {
            $("#heading").offset({
                top: event.pageY - 10,
                left: event.pageX - 10
            });
        });
    </script>
```

❶  
❷

```
        left: event.pageX,
        top: event.pageY
    });
});
</script>
</body>
</html>
```

---

В строке ❶ мы вызовом `$( "html" ).mousemove` (обработчик) добавили обработчик события `mousemove`. В данном случае аргумент `обработчик` — это функция целиком. Она начинается после `mousemove` и продолжается до тега `</script>`. Мы использовали `$( "html" )`, чтобы найти элемент `html`, поэтому обработчик будет вызван при перемещении мышки в любом месте страницы: функция, которую мы передали в скобках после `mousemove`, будет вызываться всякий раз, когда пользователь передвинет мышку.

В этом примере вместо того, чтобы создать обработчик отдельно и передать методу `mousemove` имя функции (как мы это делали ранее с функцией `clickHandler`), мы передали `mousemove` непосредственно саму функцию. Это очень распространенный подход к написанию обработчиков, поэтому освоиться с такой записью весьма полезно.

В строке ❷, в коде функции-обработчика, мы нашли элемент заголовка и вызвали для него метод `offset`. Как я уже говорил, у объекта, который передается в качестве аргумента `offset`, могут быть свойства `left` и `top`. В данном случае мы присваиваем свойству `left` значение `event.pageX`, а свойству `top` — `event.pageY`. Теперь каждый раз при передвижении мышки заголовок будет перемещаться в позицию, где произошло событие. Иными словами, куда бы вы ни передвинули мышь, заголовок будет следовать за ней.



## Что мы узнали

Из этой главы мы узнали, как писать JavaScript-код, который выполняется тогда, когда вам это нужно. Функции `setTimeout` и `setInterval` отлично подходят, чтобы запускать код спустя заданное время. Если же вам нужно выполнять код в ответ на действия пользователя, к вашим услугам `click`, `mousemove` и другие события.

В следующей главе мы с помощью пройденного сможем написать игру!

## УПРАЖНЕНИЯ

Выполните эти упражнения, чтобы опробовать разные варианты интерактивного программирования.

### #1. Следом за кликами

Измените последний пример с `mousemove` так, чтобы заголовок следовал не за указателем мышки, а только за кликами: вы кликаете в любом месте страницы, и заголовок перемещается туда.

### #2. Создайте собственную анимацию

Используйте `setInterval` для анимации заголовка `h1`, двигая его по квадрату, вдоль краев страницы. Пусть он переместится на 200 пикселей вправо, на 200 пикселей вниз, 200 пикселей влево, 200 пикселей вверх, а затем начнет с начала. Подсказка: нужно запоминать текущее направление (вправо, вниз, влево или вверх), чтобы знать, увеличивать или уменьшать для заголовка отступы слева (`left`) и сверху (`top`). Кроме того, при достижении угла квадрата нужно будет менять направление.

### #3. Остановка анимации по клику

Доработайте упражнение #2: добавьте к двигающемуся элементу `h1` обработчик клика, который останавливает анимацию. Подсказка: отменить запуск кода по интервалу можно функцией `clearInterval`.

### #4. Напишите игру «Кликни по заголовку»

Доработайте упражнение #3 так, чтобы каждый раз, когда игрок кликает по заголовку, тот не останавливался, а ускорялся и кликнуть по нему становилось сложнее. Отслеживайте количество кликов по заголовку и меняйте его текст, отображая там это число. Когда игрок наберет 10 кликов, остановите анимацию, а текст заголовка измените на «Вы победили!».

Подсказка: чтобы ускорить движение, нужно будет отменить текущий вызов функции по интервалу, а потом задать новый, с меньшим временем повтора.

# 11

## ПИШЕМ ИГРУ «НАЙДИ КЛАД!»

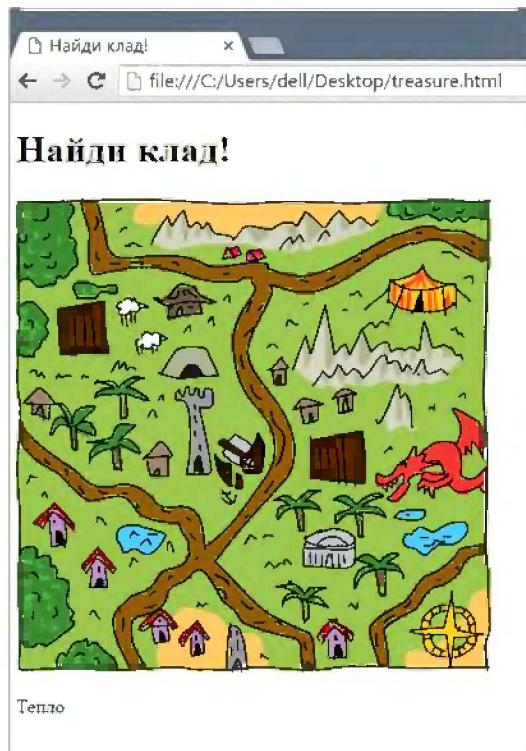


Рис. 11.1. Игра «Найди клад!»

Давайте опробуем полученные знания в деле и напишем игру! Цель игры — найти клад. Веб-страница будет отображать карту, на которой программа случайным образом выбирает точку, где спрятаны сокровища. Каждый раз, когда игрок кликает по карте, программа сообщает, насколько он близок к кладу. При клике по точке с кладом (или очень близко к ней) игра выводит поздравление и сообщает, сколько кликов ушло на поиски. На рис. 11.1 показан экран игры после того, как игрок кликнул по карте.

### Проектирование игры

Перед тем как писать код, давайте разберем общую структуру этой игры. Вот список задач, которые нужно выполнить для того, чтобы игра адекватно реагировала на клики по карте.

1. Создать страницу игры с картинкой (карты сокровищ) и местом, куда будут выводиться сообщения для игрока.

2. Выбрать на карте случайную точку, где спрятан клад.
3. Создать обработчик кликов. Каждый раз, когда игрок кликает по карте, обработчик кликов должен:
  - Увеличить счетчик кликов на 1.
  - Вычислить, насколько далеко место клика от места, где спрятан клад.
  - Отобразить на странице сообщение для игрока — «горячо» или «холодно».
  - Поздравить игрока, если он кликнул по кладу или вблизи него, и сообщить, сколько кликов ушло на поиски.

Я расскажу, как запрограммировать каждую из этих функций, а затем мы рассмотрим код игры целиком.

## Создаем веб-страницу с HTML-кодом

Давайте рассмотрим HTML-код игры. Мы воспользуемся новым элементом `img` для отображения карты клада, а для вывода игровых сообщений добавим на страницу элемент `p`. Введите следующий код в новый файл под названием `treasure.html`.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Найди клад!</title>
</head>

<body>
    <h1 id="heading">Найди клад!</h1>
    ① 
    ② <p id="distance"></p>
    ③ <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
    <script>
        // Здесь будет код игры
    </script>
</body>
</html>
```

`img` — image — изображение  
`Treasure` — сокровище

`Width` — ширина

`Height` — высота

`Distance` — расстояние

**Map** — карта

Элемент `img` служит для добавления изображений в HTML-документ. В отличие от прочих известных нам элементов, закрывающий тег `img` не нужен — потребуется лишь открывающий тег, который может содержать различные атрибуты. В строке ❶ мы добавили элемент `img` с `id` "map". С помощью атрибутов `width` и `height` мы задали ширину и высоту соответственно — и то и другое по 400, то есть наше изображение будет занимать 400 пикселей в ширину и 400 пикселей в высоту.

Чтобы указать, какое именно изображение нам нужно, мы использовали атрибут `src`, задав ему значение — веб-адрес картинки (строка ❷). В данном случае это ссылка на изображение `treasuremap.png`, которое находится на сайте издательства No Starch Press.

В строке ❸, после `img`, мы добавили пустой элемент `p`, задав ему `id` "distance" (расстояние). В этот элемент мы будем с помощью JavaScript выводить текст подсказок, сообщающих игроку, насколько он близок к цели.



## Выбор случайного места для клада

Теперь давайте писать JavaScript-код игры. Первая наша задача — выбрать на карте случайное место для клада. Поскольку размер карты  $400 \times 400$  пикселей, координаты ее верхнего левого угла равны `{ x: 0, y: 0 }`, а координаты нижнего правого угла — `{ x: 399, y: 399 }`.

## Получение случайных значений

Чтобы указать на карте сокровищ случайную точку, нам нужно выбрать случайное значение в диапазоне от 0 до 399 для координаты `x` и случайное значение в том же диапазоне для координаты `y`. Для этого напишем функцию, которая принимает размер в качестве аргумента и возвращает случайное число от 0 до этого размера (но не включая его):

**Get random number** —  
взять случайное  
число

**Size** — размер

---

```
var getRandomNumber = function (size) {
    return Math.floor(Math.random() * size);
};
```

---

Примерно такой же код мы использовали для получения случайных значений в предыдущих главах. Мы генерируем случайное число от 0 до 1 с помощью `Math.random`, умножаем его на аргумент `size` и затем используем `Math.floor` для округления до ближайшего снизу целого числа. Далее мы возвращаем полученный результат из функции. Вызов `getRandomNumber(400)` вернет случайное число от 0 до 399, что нам и требуется.

## Задаем координаты клада

Теперь используем функцию `getRandomNumber` для задания координат клада:

---

```
❶ var width = 400;  
var height = 400;  
  
❷ var target = {  
    x: getRandomNumber(width),  
    y: getRandomNumber(height)  
};
```

---

Target — цель

Во фрагменте кода начиная со строки ❶ задаются переменные `width` и `height`, соответствующие ширине и высоте элемента `img`, который мы используем в качестве карты. В строке ❷ мы создали объект под названием `target` с двумя свойствами `x` и `y`, обозначающими координаты клада. Значения `x` и `y` мы получаем из функции `getRandomNumber`. Каждый раз при запуске этого кода мы получим новую случайную позицию на карте и координаты этой позиции будут сохранены в свойствах `x` и `y` переменной `target`.

## Обработчик кликов

Обработчик кликов — функция, которая будет вызываться каждый раз, когда игрок кликнет по карте. Начнем писать эту функцию со следующего кода:

---

```
$("#map").click(function (event) {  
    // Здесь будет код обработчика  
});
```

---

Сначала мы используем `$("#map")`, чтобы найти карту (поскольку "map" — это `id` элемента `img`), а затем указываем обработчик кликов. Всякий раз, когда игрок кликнет по карте, начнется выполнение тела функции между фигурных скобок. Информация о клике будет передана в функцию через аргумент `event`.

В обработчике нужно выполнить несколько действий: увеличить счетчик кликов, вычислить, насколько точка клика отстоит от координат клада, и отобразить сообщения. Перед тем как писать код обработчика, мы создадим переменные и функции, которые помогут нам запрограммировать нужные действия.

## Подсчет кликов

Первое, что должен делать обработчик, — отслеживать число кликов. Для этого нам понадобится переменная `clicks`, которую мы создадим в начале программы (за пределами кода обработчика) и присвоим ей значение 0:

```
var clicks = 0;
```

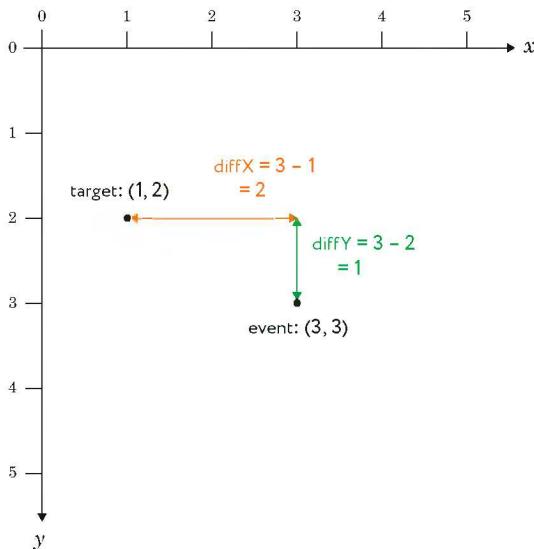
В код обработчика кликов мы включим команду `clicks++`, чтобы увеличивать счетчик каждый раз, когда игрок кликнет по карте.

## Вычисляем расстояние от клика до клада

Чтобы выяснить, «горячо» или «холодно» (вблизи клада сделан клик или далеко от него), нужно найти расстояние между точкой клика и местом, где лежит клад. Для этого создаем функцию `getDistance`, вот такую:

Get distance —  
получить  
расстояние

```
var getDistance = function (event, target) {
  var diffX = event.offsetX - target.x;
  var diffY = event.offsetY - target.y;
  return Math.sqrt((diffX * diffX) + (diffY * diffY));
};
```



Функция `getDistance` принимает два аргумента: `event` и `target`. Объект `event` — тот же самый, что передается обработчику кликов, и в нем содержится информация о событии. В частности, это свойства `offsetX` и `offsetY`, хранящие `x`- и `y`-координаты клика — как раз они нам и нужны.

В коде функции переменная `diffX` хранит горизонтальное расстояние между кликом и кладом, которое мы получаем, вычитая `target.x` (`x`-координата клада) из `event.offsetX` (`x`-координата клика). Тем же образом мы находим вертикальное расстояние, сохраняя его в переменной `diffY`. На рис. 11.2 показано, как вычисляются `diffX` и `diffY` для двух точек.

Рис. 11.2. Вычисление горизонтального и вертикального расстояний между кликом и кладом

## Используем теорему Пифагора

И наконец, чтобы найти расстояние между двумя точками в коде функции `getDistance`, используется *теорема Пифагора*. Эта теорема гласит, что для прямоугольного треугольника, где стороны, прилежащие к прямому углу, обозначены как  $a$  и  $b$ , а диагональная сторона (*гипотенуза*) обозначена как  $c$ ,  $a^2 + b^2 = c^2$ . Зная длины  $a$  и  $b$ , мы можем найти длину гипотенузы, взяв квадратный корень от  $a^2 + b^2$ .

Чтобы найти расстояние между кликом и кладом, мы рассматриваем эти две точки как углы прямоугольного треугольника (см. рис. 11.3). В функции `getDistance` переменная `diffX` — это длина горизонтальной стороны треугольника, а `diffY` — длина вертикальной стороны.

Найти нужное нам расстояние — значит найти длину гипотенузы, зная длины `diffX` и `diffY`. Пример такого вычисления показан на рис. 11.3.

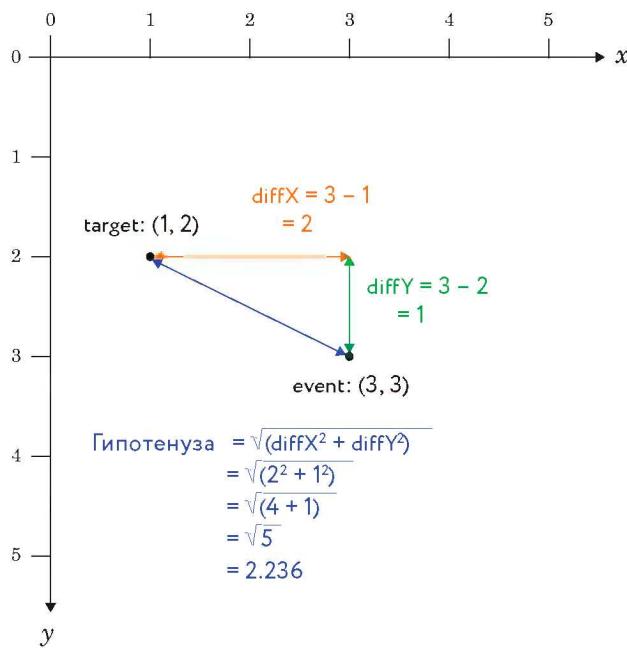
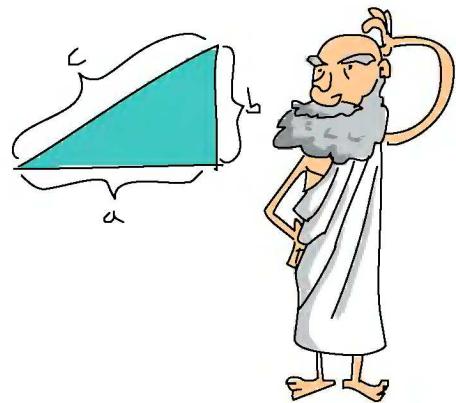


Рис. 11.3. Вычисление гипотенузы как расстояния между кликом и кладом

Чтобы найти гипотенузу, сначала нужно возвести `diffX` и `diffY` в квадрат. Затем мы складываем эти значения и извлекаем из суммы



квадратный корень с помощью JavaScript-функции `Math.sqrt`. Целиком формула для вычисления расстояния между кликом и кладом выглядит так:

---

```
Math.sqrt((diffX * diffX) + (diffY * diffY))
```

---

Функция `getDistance` вычисляет это выражение и возвращает результат.

## Сообщаем игроку, насколько он близок к цели

Зная расстояние между кликом и кладом, остается отобразить подсказку, которая сообщала бы игроку, насколько близко он подошел, — но без конкретных цифр. Для этого создадим следующую функцию `getDistanceHint`:

**Hint —**  
подсказка

---

```
var getDistanceHint = function (distance) {
    if (distance < 10) {
        return "Обожжешься!";
    } else if (distance < 20) {
        return "Очень горячо";
    } else if (distance < 40) {
        return "Горячо";
    } else if (distance < 80) {
        return "Тепло";
    } else if (distance < 160) {
        return "Холодно";
    } else if (distance < 320) {
        return "Очень холодно";
    } else {
        return "Замерзнешь!";
    }
};
```

---



Эта функция возвращает одну из строк в зависимости от переданного ей расстояния до клада. Если это расстояние меньше 10, она вернет «Обожжешься!». Для расстояния от 10 до 20 функция вернет «Очень горячо». По мере увеличения расстояния функция будет сообщать о все большем холода, вплоть до расстояния в 320 пикселей, начиная с которого функция возвращает строку «Замерзнешь!».

Отображать эти сообщения мы будем, задавая их как текстовое содержимое элемента `p` на нашей странице. Следующий код вычислит расстояние, получит нужную строку с сообщением и отобразит эту строку:

---

```
var distance = getDistance(event, target);
var distanceHint = getDistanceHint(distance);
$("#distance").text(distanceHint);
```

---

Как видите, сначала мы вызываем `getDistance`, сохраняя возвращенное значение в переменной `distance`. Затем мы передаем `distance` в функцию `getDistanceHint`, чтобы получить соответствующую строку подсказки и сохранить ее в переменной `distanceHint`.

Код `$("#distance").text(distanceHint);` находит элемент, id которого равен "distance" (в нашем случае это элемент p), и меняет его текст на значение `distanceHint`, так что всякий раз при клике по карте наша веб-страничка сообщает игроку, насколько близко он подошел к цели.

## Проверка на выигрыш

И наконец, наш обработчик кликов должен проверить, не попал ли игрок в цель. Поскольку один пиксель очень мал, мы не будем вынуждать игрока кликать в точности по месту с кладом, а засчитаем за победу клик на расстоянии менее восьми пикселей.

Этот код проверяет расстояние до клада, в случае победы сообщая об этом игроку:

---

```
if (distance < 8) {
    alert("Клад найден! Сделано кликов: " + clicks);
}
```

---

## Код игры

Теперь, когда у нас есть все части кода, объединим их в программу.

---

```
// Получить случайное число от 0 до size-1
var getRandomNumber = function (size) {
    return Math.floor(Math.random() * size);
};

// Вычислить расстояние от клика (event) до клада (target)
var getDistance = function (event, target) {
    var diffX = event.offsetX - target.x;
    var diffY = event.offsetY - target.y;
    return Math.sqrt((diffX * diffX) + (diffY * diffY));
};

// Получить для расстояния строку подсказки
var getDistanceHint = function (distance) {
```

---

```

        if (distance < 10) {
            return "Обожжешься!";
        } else if (distance < 20) {
            return "Очень горячо";
        } else if (distance < 40) {
            return "Горячо";
        } else if (distance < 80) {
            return "Тепло";
        } else if (distance < 160) {
            return "Холодно";
        } else if (distance < 320) {
            return "Очень холодно";
        } else {
            return "Замерзнешь!";
        }
    };

    // Создаем переменные
① var width = 400;
var height = 400;
var clicks = 0;

    // Случайная позиция клада
② var target = {
    x: getRandomNumber(width),
    y: getRandomNumber(height)
};

    // Добавляем элементу img обработчик клика
③ $("#map").click(function (event) {
    clicks++;

    // Получаем расстояние от места клика до клада
④ var distance = getDistance(event, target);

    // Преобразуем расстояние в подсказку
⑤ var distanceHint = getDistanceHint(distance);

    // Записываем в элемент #distance новую подсказку
⑥ $("#distance").text(distanceHint);

    // Если клик был достаточно близко, поздравляем с победой
⑦ if (distance < 8) {
    alert("Клад найден! Сделано кликов: " + clicks);
}
});

```

---

Начинается код с функций `getRandomNumber`, `getDistance` и `getDistanceHint`, о которых мы уже говорили. Затем в строке ① мы создали необходимые переменные: `width`, `height` и `clicks`. Далее в строке ② задается случайная позиция для клада.

В строке ❸ мы добавили элементу карты обработчик кликов. Первым делом этот обработчик увеличивает на 1 переменную `clicks`. Затем в строке ❹ он вычисляет расстояние между `event` (местом клика) и `target` (позицией клада). В строке ❺ мы использовали функцию `getDistanceHint` для преобразования этого расстояния в строку ("Холодно", "Тепло" и т. д.). В строке ❻ мы обновляем подсказку на экране, чтобы игрок видел, насколько он близок к цели. И наконец, в строке ❼ проверяем, уложился ли игрок в расстояние меньше 8 пикселей от клада, и если уложился, мы сообщаем ему о победе и количестве затраченных кликов.

Это весь JavaScript-код игры. Добавив его ко второму элементу `script` в файле `treasure.html`, вы сможете запустить игру в браузере! За сколько кликов вам удастся найти клад?

## Что мы узнали

В этой главе вы создали настоящую игру, опираясь на знания о работе с событиями. Также вы познакомились с элементом `img`, посредством которого можно добавлять на веб-странички изображения. И наконец, мы разобрались, как с помощью JavaScript узнать расстояние между двумя точками.

В следующей главе мы познакомимся с объектно-ориентированным программированием, которое даст нам новые возможности для организации и структурирования кода.

## УПРАЖНЕНИЯ

Вот несколько идей по изменению и усовершенствованию кода игры.

### #1. Увеличение игрового поля

Сделайте игру сложнее, увеличив размер игрового поля. Попробуйте изменить его размер до 800 пикселей в ширину и 800 в высоту.

### #2. Больше подсказок

Ведите в игру дополнительные подсказки (например, «Очень-очень холодно!») и добавьте соответствующие подсказкам расстояния.

### #3. Ограничение по кликам

Установите ограничение количества кликов и показывайте игроку сообщение «КОНЕЦ ИГРЫ», если он превысит это ограничение.

### #4. Отображение числа оставшихся кликов

После подсказки «Горячо» или «Холодно» выводите на экран число оставшихся кликов, чтобы проигрыш не был для игрока неожиданностью.

# 12

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В этой главе мы узнаем, как создавать и использовать объекты в рамках *объектно-ориентированного программирования*. Объектно-ориентированное программирование (ООП) — это способ проектирования и написания кода, когда все важные части программы являются объектами. Например, если вы пишете видеоигру «гонки», вы можете воспользоваться технологией ООП и запрограммировать объект «машина», а затем создать множество таких объектов, обладающих одинаковым набором свойств и одинаковой функциональностью.

### Простой объект

Как вы уже знаете из четвертой главы, объекты состоят из свойств, которые представляют собой пары «ключ-значение». Например, в следующем коде объект `dog` описывает собаку, и у него есть свойства — `name` (имя), `legs` (количество лап) и `isAwesome` (обозначает, хороша ли эта собака):

---

```
var dog = {  
    name: "Оладушек",  
    legs: 4,  
    isAwesome: true  
};
```

---

К свойствам созданного объекта можно обращаться через точечную нотацию (о ней мы говорили на с. 72 в разделе «Доступ к значениям внутри объектов»). Например, так мы можем получить свойство name нашего объекта dog:

```
dog.name;  
"Оладушек"
```

Кроме того, с помощью точечной нотации можно добавлять объекту новые свойства:

```
dog.age = 6;
```

При этом у объекта появится еще одна пара «ключ-значение» (age: 6), в чем можно легко убедиться:

```
dog;  
Object {name: "Оладушек", legs: 4, isAwesome: true, age: 6}
```

## Добавление к объектам новых методов

В последнем примере мы создали несколько свойств, которые хранят значения разных типов: строку ("Оладушек"), числа (4 и 6) и булево значение (true). Помимо строк, чисел и булевых значений в свойствах объектов можно хранить функции — тогда эти свойства называют *методами*. В сущности, мы уже пользовались некоторыми встроенными в JavaScript методами: например, это метод join для массивов и метод toUpperCase для строк.

А теперь давайте посмотрим, как создавать собственные методы. Один из способов — воспользоваться точечной нотацией. К примеру, научим нашу собаку лаять, добавив к объекту dog метод под названием bark:

Bark — лай

```
① dog.bark = function () {  
②   console.log("Гав-гав! Меня зовут " + this.name + "!");  
};  
  
③ dog.bark();  
Гав-гав! Меня зовут Оладушек!
```

В строке ① мы добавили к объекту dog свойство bark и задали в качестве его значения функцию. В строке ②, в теле этой функции, мы

использовали `console.log`, чтобы напечатать: «Гав-гав! Меня зовут Оладушек!» Обратите внимание на запись `this.name` — таким образом мы получаем значение, сохраненное в свойстве `name` этого объекта. Давайте разберемся подробнее, как работает ключевое слово `this`.

## Ключевое слово `this`

**This** — этот, это

Ключевое слово `this` можно использовать в теле метода, чтобы обращаться к объекту, для которого этот метод вызывается. Например, при вызове метода `bark` для объекта `dog`, `this` обозначает объект `dog`, а значит `this.name` — это свойство `dog.name`. Ключевое слово `this` делает методы более гибкими, позволяя добавлять один и тот же метод ко многим объектам так, чтобы он имел доступ к свойствам того объекта, для которого в данный момент вызывается.

## Используем один метод с разными объектами

**Speak** — говорить

Давайте создадим новую функцию `speak`, чтобы затем использовать ее как метод с разными объектами, обозначающими разных животных. В случае вызова для какого-нибудь объекта метод `speak` будет обращаться к имени объекта (`this.name`) и звуку, который издает животное (`this.sound`), чтобы вывести в консоль сообщение.

---

```
var speak = function () {
    console.log(this.sound + "! Меня зовут " + this.name + "!");
};
```

---

Теперь создадим еще один объект, чтобы добавить к нему функцию `speak` в качестве метода:

---

```
var cat = {
    sound: "Мяу",
    name: "Варежка",
①     speak: speak
};
```

---

**Cat** — кошка

Здесь мы создали новый объект `cat` со свойствами `sound`, `name` и `speak`. В строке ① мы присвоили свойству `speak` значение — созданную ранее функцию `speak`. Теперь `cat.speak` является методом, который можно вызывать командой `cat.speak()`. Поскольку в коде метода используется ключевое слово `this`, в случае вызова для объекта `cat` он получит доступ к свойствам именно этого объекта. Давайте проверим:

```
cat.speak();  
Мяу! Меня зовут Варежка!
```

Когда мы вызываем метод `cat.speak`, он запрашивает значения двух свойств объекта `cat`: `this.sound` (это "Мяу") и `this.name` (это "Варежка").

Ту же самую функцию `speak` можно использовать как метод и для других объектов:

```
var pig = {  
    sound: "Хрю",  
    name: "Чарли",  
    speak: speak  
};  
var horse = {  
    sound: "И-го-го",  
    name: "Мэри",  
    speak: speak  
};  
pig.speak();  
Хрю! Меня зовут Чарли!  
horse.speak();  
И-го-го! Меня зовут Мэри!
```

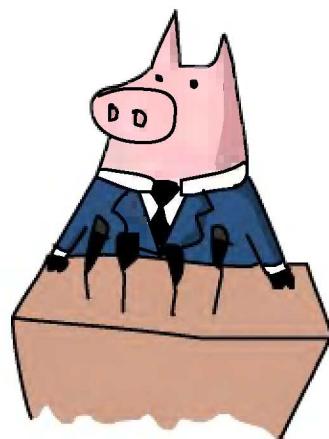


Pig —  
поросенок  
Horse —  
лошадь

Повторюсь: в коде метода ключевое слово `this` ссылается на объект, для которого метод был вызван. Другими словами, при вызове `horse.speak()` `this` означает объект `horse`, а при вызове `pig.speak()` — объект `pig`.

Чтобы использовать один и тот же код метода с разными объектами, достаточно добавить его в виде свойства каждому из этих объектов — это мы и сделали сейчас с функцией `speak`. Однако если в программе много объектов и методов, добавление методов вручную будет задачей весьма утомительной, а код при этом станет запутанным и неаккуратным. Только представьте, что вам нужен целый зоопарк объектов для сотни разных животных и вы хотите, чтобы у них всех было с десяток общих методов и свойств.

Конструкторы объектов позволяют задавать общие методы и свойства куда более удобным способом, и сейчас мы в этом убедимся.



## Создание объектов с помощью конструкторов

В JavaScript **конструктор** — это функция, которая создает объекты, давая им набор заранее определенных свойств и методов. Представьте себе, что это машина по созданию объектов, вроде фабрики, которая штампует тысячи копий одного и того же товара. Задав конструктор, вы сможете создавать с его помощью любое количество одинаковых объектов. Чтобы опробовать этот подход в деле, мы напишем основу игры «гонки», где используем конструктор для создания парка машин с одинаковыми базовыми свойствами, а также методами для перемещения в разные стороны и изменения скорости.

### Устройство конструктора

**New** — новый

При каждом вызове конструктор создает объект, добавляя ему нужные свойства. Если для вызова обычной функции мы указывали ее имя и следом круглые скобки, то для вызова конструктора используется ключевое слово **new** (которое сообщает JavaScript, что вы собираетесь использовать функцию как конструктор), а следом — имя конструктора и скобки. На рис. 12.1 показан синтаксис вызова конструктора.



Рис. 12.1. Синтаксис вызова конструктора с именем *Car* и двумя аргументами



Большинство JavaScript-программистов называют конструкторы с заглавной буквы, чтобы отличать их от обычных функций.

**Car** — машина

### Создаем конструктор *Car*

Давайте создадим конструктор *Car*, который будет добавлять к каждому созданному объекту свойства *x* и *y*. Мы будем использовать эти свойства как координаты, задающие позицию машины на экране при отображении.

#### Создаем HTML-документ

Прежде чем писать код конструктора, нам нужно создать новый HTML-документ. Создайте новый файл под именем *cars.html* и введите в него следующий HTML-код:

---

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Гонки</title>
</head>
<body>
    <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
    <script>
        // Здесь будет JavaScript-код
    </script>
</body>
</html>
```

---

### Функция-конструктор Car

Теперь введите следующий код между тегов `<script>` в файле `cars.html` (вместо комментария `// Здесь будет JavaScript-код`), чтобы создать конструктор `Car`, добавляющий каждый раз при вызове конструктора пару координат.

---

```
<script>
var Car = function (x, y) {
    this.x = x;
    this.y = y;
};
</script>
```

---

Наш новый конструктор принимает два аргумента, `x` и `y`, и добавляет новому объекту свойства `this.x` и `this.y`, сохраняя в них переданные значения `x` и `y`. Таким образом, каждый раз при вызове конструктора `Car` будет создан новый объект со свойствами `x` и `y`, значения которых соответствуют переданным аргументам.

### Вызов конструктора Car

Как я уже говорил, ключевое слово `new` указывает JavaScript, что мы вызываем конструктор для создания нового объекта. Например, чтобы создать объект-машину с именем `tesla`, откройте файл `cars.html` в браузере и введите в JavaScript-консоли Chrome такой код:

---

```
var tesla = new Car(10, 20);
tesla;
Car {x: 10, y: 20}
```

---

Код `Car(10, 20)` указывает JavaScript, что нужно создать объект, используя функцию `Car` в качестве конструктора и передав ей аргументы `10` и `20` для свойств `x` и `y`, и затем вернуть этот объект. Полученный новый объект мы сохраняем в переменной `tesla`.

Далее, когда мы ввели `tesla`, консоль Chrome напечатала имя конструктора и свойства `x` и `y`: `Car {x: 10, y: 20}`.



## Рисуем машины

**Draw car** —  
нарисовать  
машину

Для отображения объектов, созданных конструктором `Car`, создадим функцию под названием `drawCar` — она будет помещать изображение машины в позицию `(x, y)`, соответствующую свойствам `x` и `y` каждого нашего объекта. Разобравшись, как эта функция работает, мы перепишем ее в объектно-ориентированном виде — см. раздел «Добавляем метод `draw` к прототипу `Car`» на с. 185. Введите этот код между тегов `<script>` в файле `cars.html`:

```
script
var Car = function (x, y) {
    this.x = x;
    this.y = y;
};

① var drawCar = function (car) {
    ② var carHtml = '';
    ③ var carElement = $(carHtml);
    ④ carElement.css({
        position: "absolute",
        left: car.x,
        top: car.y
    });
    ⑤ $("body").append(carElement);
};


```

**Position** —  
положение

**Left** — слева,  
влево

**Top** — вверх,  
вверх

**Append** —  
добавить

В строке ❶ мы создали строку с HTML-кодом, содержащим ссылку на изображение машины (чтобы использовать двойные кавычки в HTML-коде, при создании строки поставьте одинарные кавычки). В строке ❷ мы передаем carHTML в функцию \$, которая преобразует HTML в jQuery-элемент. Это значит, что в переменной carElement теперь хранится jQuery-элемент с информацией о теге `<img>` и мы сможем изменить свойства этого элемента перед тем, как добавить его на страницу.

В строке ❸ мы вызываем для carElement метод `css`, чтобы задать изображению машины координаты. Этот код устанавливает отступ слева согласно координате `x` объекта и отступ сверху согласно его координате `y`. Иными словами, в окне браузера левый край изображения будет отстоять от левой границы окна на `x` пикселей, а верхний край изображения будет отстоять от верхней границы окна на `y` пикселей.

! В этом примере метод `css` используется подобно методу `offset`, который мы применяли для перемещения элементов по странице в десятой главе. К сожалению, `offset` не так хорошо работает с несколькими элементами, а нам нужно отображать сразу несколько машин, поэтому в этом примере мы будем пользоваться `css`.



И наконец, в строке ❹ мы с помощью jQuery добавляем carElement к элементу `body` нашего HTML-документа. После этого carElement появится на странице. (Чтобы вспомнить, как работает метод `append`, вернитесь к разделу «Создание новых элементов через jQuery» на с. 150.)

## Проверка функции drawCar

Давайте убедимся, что функция `drawCar` работает как положено. Добавьте этот код в файл `cars.html` (после остального JavaScript-кода), чтобы создать две машины.

```
$("body").append(carElement);  
};  
var tesla = new Car(20, 20);  
var nissan = new Car(100, 200);  
drawCar(tesla);  
drawCar(nissan);  
</script>
```

Здесь мы использовали конструктор `Car` для создания двух объектов — первого с координатами (20, 20) и второго с координатами (100, 200) — и затем отобразили их в окне браузера с помощью `drawCar`.

Открыв теперь *cars.html*, вы должны увидеть изображения двух машин, как на рис. 12.2.

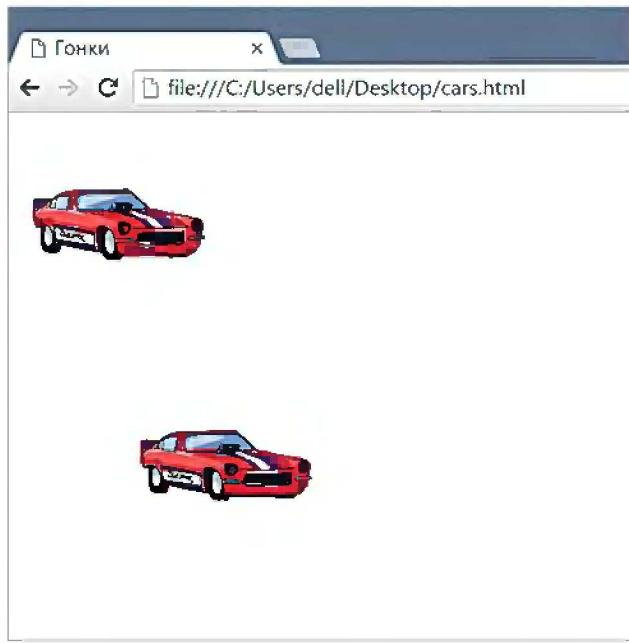


Рис. 12.2. Отображение машин с помощью drawCar

### Настройка объектов через прототипы

Есть другой, более объектно-ориентированный подход к рисованию машин — задать для каждого объекта-машины метод отрисовки (назовем его `draw`). Тогда вместо `drawCar(tesla)` можно будет написать `tesla.draw()`. В объектно-ориентированном программировании принято, чтобы объекты обладали собственной функциональностью, организованной в виде методов. В нашем случае функция `drawCar` изначально предназначена для изображения объектов-машин, поэтому стоит сделать ее частью каждого объекта, а не использовать как отдельную функцию.

Прототипы JavaScript — это механизм, который упрощает использование общей функциональности (то есть методов) разными объектами. У всех конструкторов есть свойство `prototype`, к которому можно добавлять методы; любой метод, добавленный к свойству `prototype`, будет доступен всем объектам, которые созданы с помощью этого конструктора.

На рис. 12.3 показан синтаксис добавления метода к свойству `prototype`.

```

    Имя конструктора      Имя метода
    ↓                  ↓
Car.prototype.draw = function () {
    // Тело метода
}

```

Рис. 12.3. Синтаксис добавления метода к свойству `prototype`

## Добавляем метод `draw` к прототипу `Car`

Давайте добавим метод `draw` к свойству `Car.prototype`, чтобы он появился у всех объектов, созданных вызовом `new Car`. Выберите `File → Save As` и сохраните файл `cars.html` под именем `cars2.html`. Далее замените весь JavaScript-код, находящийся во втором элементе `script`, следующим кодом:

---

```

❶ var Car = function (x, y) {
    this.x = x;
    this.y = y;
};

❷ Car.prototype.draw = function () {
    var carHtml = '';

❸     this.carElement = $(carHtml);

    this.carElement.css({
        position: "absolute",
        left: this.x,
        top: this.y
    });

    $("body").append(this.carElement);
};

var tesla = new Car(20, 20);
var nissan = new Car(100, 200);

tesla.draw();
nissan.draw();

```

---

После создания конструктора в строке ❶, в строке ❷ мы добавили к `Car.prototype` новый метод `draw`. Таким образом, `draw` станет частью всех объектов, созданных конструктором `Car`.

Код метода `draw` представляет собой слегка измененную функцию `drawCar`. Сначала мы создаем строку с HTML-кодом, сохраняя ее в переменной `carHTML`. В строке ❸ мы создаем jQuery-элемент для этого

**Move right** —  
подвинуть  
вправо

HTML, на этот раз сохраняя его в свойстве объекта `this.carElement`. Далее в строке ④ мы, обращаясь к `this.x` и `this.y`, задаем координаты верхнего левого угла изображения. (В конструкторе `this` соответствует объекту, который в данный момент создается.)

Когда вы запустите этот код, страничка будет выглядеть так же, как на рис. 12.2: функциональность кода осталась прежней, мы лишь изменили его структуру. Преимущество этого подхода в том, что теперь код рисования является частью объекта (машины), а не отдельной функцией.

## Добавляем метод `moveRight`

Теперь добавим несколько методов для перемещения машин по экрану, начиная с метода `moveRight`, передвигающего машину на 5 пикселей вправо относительно текущей позиции. Добавьте следующий код после определения `Car.prototype.draw`:

---

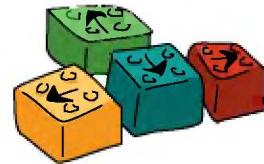
```
        this.carElement.css({
            position: "absolute",
            left: this.x,
            top: this.y
        });

        $("body").append(this.carElement);
    };

    Car.prototype.moveRight = function () {
        this.x += 5;

        this.carElement.css({
            left: this.x,
            top: this.y
        });
    };
}
```

---



Мы сохранили метод `moveRight` в свойстве `Car.prototype`, чтобы сделать его частью всех объектов, созданных с помощью конструктора `Car`. Командой `this.x += 5` мы увеличиваем координату `x` машины на 5, чтобы переместить ее на 5 пикселей вправо. Затем мы вызываем метод `css` для `this.carElement`, чтобы обновить позицию машины в браузере.

Попробуйте вызвать метод `moveRight` из консоли браузера. Сначала обновите окно с документом `cars2.html`, затем откройте консоль и введите такие строки:

---

```
tesla.moveRight();
tesla.moveRight();
tesla.moveRight();
```

---

Каждый раз при вводе tesla.moveRight верхняя машина должна передвинуться вправо на 5 пикселей. Вы можете использовать этот метод в игре для перемещения машины по трассе.

### ПОПРОБУЙТЕ!

Передвигните вправо nissan. Сколько раз нужно вызывать moveRight для nissan, чтобы эта машина догнала tesla?

С помощью setInterval и moveRight анимируйте nissan, чтобы машина поехала от левой границы окна к правой.

## Добавляем методы для движения влево, вверх и вниз

Теперь давайте добавим в наш код методы для остальных направлений, чтобы машины могли двигаться по экрану в разные стороны. Эти методы почти не отличаются от moveRight, поэтому введем их все разом.

Добавьте следующие методы в файл cars2.html сразу после moveRight:

```
Car.prototype.moveRight = function () {
    this.x += 5;
    this.carElement.css({
        left: this.x,
        top: this.y
    });
}
Car.prototype.moveLeft = function () {
    this.x -= 5;
    this.carElement.css({
        left: this.x,
        top: this.y
    });
}
Car.prototype.moveUp = function () {
    this.y -= 5;
    this.carElement.css({
        left: this.x,
        top: this.y
    });
}
Car.prototype.moveDown = function () {
    this.y += 5;
    this.carElement.css({
        left: this.x,
        top: this.y
    });
}
```

**Move right** — подвинуть вправо

**Move left** — подвинуть влево

**Move up** — подвинуть вверх

**Move down** — подвинуть вниз

Каждый из этих методов передвигает машину на 5 пикселей в указанном направлении, увеличивая или уменьшая на 5 одно из свойств объекта-машины `x` или `y`.



### Что мы узнали

В этой главе вы познакомились с основами объектно-ориентированного программирования на JavaScript: узнали, как задавать конструкторы для создания новых объектов и как использовать свойство `prototype` этих конструкторов, чтобы добавлять объектам общие методы.

В программах, написанных объектно, большинство функций реализовано в виде методов.

Например, чтобы отобразить машину, мы вызываем для объекта-машины метод `draw`, а чтобы передвинуть ее вправо, вызываем метод `moveRight`. Конструкторы и прототипы — это встроенные механизмы языка JavaScript, предназначенные для создания объектов с общим набором методов.

Объектный подход к написанию JavaScript-кода поможет вам структурировать программы. Если код хорошо структурирован, вам будет проще вспомнить, как он работает, если спустя некоторое время вы решите внести в него изменения (это особенно важно для больших программ или для работы в команде с другими программистами, которым может понадобиться изучить ваш код или изменить его). Например, ближе к концу этой книги мы создадим игру «Змейка», для которой понадобится написать немало строк кода; объекты и методы пригодятся нам, чтобы структурировать программу и реализовать изрядную часть ее функциональности.

В следующей главе мы выясним, как с помощью элемента `canvas` рисовать на веб-странице линии и другие фигуры и анимировать их.

## УПРАЖНЕНИЯ

Выполните эти упражнения, чтобы попрактиковаться в работе с объектами и прототипами.

### #1. Рисование в конструкторе Car

Добавьте вызов метода `draw` в конструктор `Car`, чтобы объекты автоматически отображались в окне браузера после их создания.

### #2. Добавьте свойство speed

Доработайте конструктор `Car`, чтобы он добавлял создаваемым объектам свойство `speed` (скорость) со значением 5. Используйте это свойство в методах перемещения вместо числа 5.

Затем попробуйте задавать различные значения скорости, чтобы машины двигались быстрее или медленнее.

### #3. Гонки

Доработайте методы `moveLeft`, `moveRight`, `moveUp` и `moveDown`, чтобы вместо перемещения машин всегда ровно на 5 пикселей они принимали величину сдвига в качестве аргумента. К примеру, в этом случае для перемещения машины `nissan` на 10 пикселей вправо нужно будет дать команду `nissan.moveRight(10)`.

Теперь используйте `setInterval`, чтобы двигать две машины (`nissan` и `tesla`) вправо, каждые 30 миллисекунд смешая их на случайное расстояние от 0 до 5 пикселей. Вы увидите, как машины едут по экрану, то и дело меняя скорость. Попробуйте угадать, какая из машин достигнет границы окна первой.



ЧАСТЬ III

# Графика



# 13

## ЭЛЕМЕНТ CANVAS

Возможности JavaScript не ограничиваются работой с текстом и числами. Также можно писать код для рисования картинок с помощью HTML-элемента canvas, который представляет собой что-то вроде чистого холста или листа бумаги. Вы можете рисовать на этом «холсте» практически все что угодно: чертить линии, контуры, выводить текст — пределов нет, кроме вашего воображения!

Эта глава посвящена основам рисования на «холсте». В дальнейших главах мы, отталкиваясь от этих основ, приступим к созданию видеоигры на JavaScript, использующей элемент canvas.

### Создаем «холст»

Прежде чем начать работу с элементом canvas, создайте новый HTML-документ со следующим кодом и сохраните его в файле *canvas.html*:

---

```
<!DOCTYPE html>
<html>
<head>
    <title>Canvas</title>
</head>

<body>
    <canvas id="canvas" width="200" height="200"></canvas>

    <script>
```

```
// Здесь будет JavaScript-код
</script>
</body>
</html>
```

Как видите, в строке ❶ мы создали элемент `canvas`, дав атрибуту `id` значение `"canvas"`, чтобы затем обращаться к нему с помощью этого `id`. Атрибуты `width` и `height` задают ширину и высоту «холста» — в данном случае его размер  $200 \times 200$  пикселей.

## Рисование на «холсте»

Теперь, когда у нас есть страница с элементом `canvas`, давайте попробуем при помощи JavaScript-кода нарисовать на «холсте» прямоугольник. Введите следующий код в файл `canvas.html` между тегов `<script>`.

Ctx —  
context —  
контекст  
  
Get context —  
получить  
контекст

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
ctx.fillRect(0, 0, 10, 10);
```

Давайте разберем этот код строчка за строчкой.

## Ищем и сохраняем элемент `canvas`

Первым делом находим элемент `canvas` с помощью команды `document.getElementById("canvas")`. Как мы знаем из девятой главы, метод `getElementById` возвращает объект DOM для элемента с указанным `id`. Теперь сохраним этот объект в переменной с именем `canvas`: `var canvas = document.getElementById("canvas")`.



## Получаем контекст рисования

Теперь нужно получить контекст рисования для элемента `canvas`. Контекст рисования — это JavaScript-объект, обладающий методами и свойствами, при помощи которых можно рисовать на «холсте». Чтобы получить этот объект, мы вызываем для `canvas` метод `getContext`, передавая ему строку `"2d"`, — это означает, что мы собираемся формировать на «холсте» двухмерное изображение. Сохраним контекст в переменной `ctx` с помощью команды `var ctx = canvas.getContext("2d")`.

Rect —  
rectangle —  
прямо-  
угольник

## Рисуем квадрат

И наконец, в последней строке мы рисуем на «холсте» равносторонний прямоугольник (то есть квадрат), вызывая метод контекста рисования `fillRect`, который принимает четыре аргумента: это, в порядке очередности, *x*- и *y*-координаты верхнего левого угла квадрата (0, 0), а также его ширина и высота (10, 10). В данном случае мы просим JavaScript нарисовать прямоугольник 10 × 10 пикселей в координатах (0, 0) — то есть в верхнем левом углу «холста».

Запустив этот код, вы должны увидеть на экране черный квадратик, как на рис. 13.1.



Рис. 13.1. Наш первый рисунок на «холсте»

## Рисуем несколько квадратов

Теперь попробуем кое-что поинтереснее. Вместо рисования единственного квадратика изобразим в цикле множество квадратов, расположенных по диагонали. Замените код между тегами `<script>` следующим кодом. Запустив его, вы должны увидеть восемь черных квадратиков, как на рис. 13.2:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
for (var i = 0; i < 8; i++) {
    ctx.fillRect(i * 10, i * 10, 10, 10);
}
```

Первые две строки кода остались прежними. В третьей строке мы задаем цикл, который повторяется восемь раз. В теле этого цикла мы вызываем метод контекста рисования `fillRect`.



Рис. 13.2. Рисование нескольких квадратов в цикле `for`



X- и у-координаты левого верхнего угла каждого из квадратов высчитываются на основе значения переменной цикла *i*. При первом повторе цикла, когда *i* = 0, координаты квадрата будут (0, 0), поскольку  $0 \times 10$  это 0. Следовательно, команда `ctx.fillRect(i * 10, i * 10, 10, 10)` нарисует квадрат со стороной 10 пикселей в координатах (0, 0). Это левый верхний квадрат на рис. 13.2.

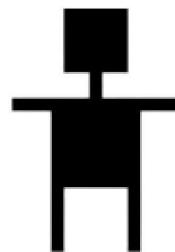
Второй повтор цикла, когда *i* = 1, соответствует координатам (10, 10), поскольку  $1 \times 10$  это 10. На этот раз команда `ctx.fillRect(i * 10, i * 10, 10, 10)` нарисует квадрат в координатах (10, 10), однако размер его сторон будет по-прежнему равен 10 пикселям (ведь аргументы для ширины и высоты остались прежними). Это второй сверху квадрат на рис. 13.2.

Поскольку при каждом повторе цикла *i* увеличивается на 1, координаты *x* и *y* каждый раз увеличиваются на 10 пикселей, а размер стороны остается равным 10. В ходе оставшихся шести повторов будут нарисованы шесть нижних квадратов.

### ПОПРОБУЙТЕ!

Теперь, когда вы знаете, как рисовать на «холсте» прямоугольники и квадраты, попробуйте изобразить этого робота с помощью метода `fillRect`.

Подсказка: вам нужно нарисовать шесть прямоугольников. Голова — это квадрат со стороной 50 пикселей, а ширина шеи, рук и ног — 10 пикселей.



### Выбор цвета

Style — стиль

По умолчанию при вызове `fillRect` JavaScript рисует черный квадрат. Чтобы поменять цвет, нужно изменить у контекста рисования свойство `fillStyle`. После того как `fillStyle` примет значение нового цвета, следующие фигуры будут рисоваться этим цветом до тех пор, пока вы снова не измените значение `fillStyle`.

Самый простой способ задать цвет для `fillStyle` — присвоить ему название цвета в виде строки. Например:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
① ctx.fillStyle = "Red";
ctx.fillRect(0, 0, 100, 100);
```

Red — красный

В строке ① мы говорим контексту рисования, что с этого момента все, что мы рисуем, должно быть красным (Red). Если запустить этот код, на экране появится ярко-красный квадрат, как на рис. 13.3.

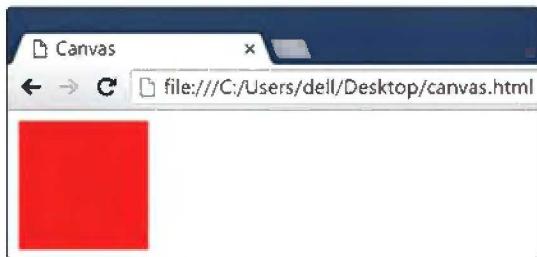


Рис. 13.3. Красный квадрат

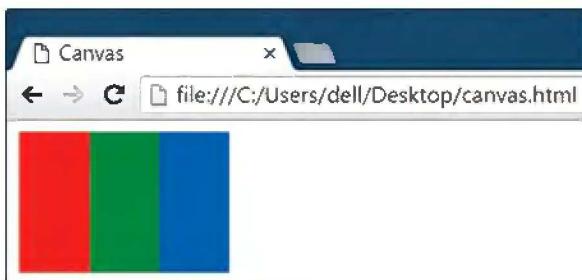


JavaScript понимает английские названия более 100 цветов, например *Green*, *Blue*, *Orange*, *Red*, *Yellow*, *Purple*, *White*, *Black*, *Pink*, *Turquoise*, *Violet*, *SkyBlue*, *PaleGreen* и др. Полный список можно найти на сайте CSS-Tricks: <http://css-tricks.com/snippets/css/named-colors-and-hex-equivalents/>.

<b>Green</b> —	зеленый,
<b>Blue</b> —	синий,
<b>Orange</b> —	оранжевый,
<b>Yellow</b> —	желтый,
<b>Purple</b> —	пурпурный,
<b>White</b> —	белый,
<b>Black</b> —	черный,
<b>Pink</b> —	розовый,
<b>Turquoise</b> —	бирюзовый,
<b>Violet</b> —	фиолетовый,
<b>SkyBlue</b> —	небесно-голубой,
<b>PaleGreen</b> —	светло-зеленый

### ПОПРОБУЙТЕ!

Зайдите на сайт CSS-Tricks (<http://css-tricks.com/snippets/css/named-colors-and-hex-equivalents/>), выберите три цвета на свой вкус и нарисуйте три прямоугольника. Пусть каждый из них будет 50 пикселей в ширину и 100 пикселей в высоту, и пусть они располагаются сторона к стороне, без пробелов. Должно получиться нечто вроде:



...хотя я уверен, что вы найдете цвета поинтереснее, чем красный, зеленый и синий!

### Рисование контуров прямоугольников

Как видите, метод `fillRect` рисует заполненные прямоугольники. Хорошо, если именно это вам и нужно, но порой может понадобиться изобразить лишь контур прямоугольника, так, будто он обведен ручкой или карандашом. Для этого предназначен метод `strokeRect`. Например, если запустить следующий код, на экране появится контур небольшого прямоугольника, такой как на рис. 13.4:

**Stroke** — здесь «контур»

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
ctx.strokeRect(10, 10, 100, 20);
```



Рис. 13.4. Контур прямоугольника, нарисованный с помощью strokeRect

Метод `strokeRect` принимает те же аргументы, что и `fillRect`: это  $x$ - и  $y$ -координаты верхнего левого угла, а затем ширина и высота прямоугольника. В данном случае прямоугольник изображен с отступом в 10 пикселей от левого верхнего угла «холста», ширина его равна 100 пикселям, а высота 20 пикселям.

Line width —  
толщина  
линий

Изменить цвет контура можно с помощью свойства `strokeStyle`, а чтобы задать толщину линии, нужно свойство `lineWidth`. Например:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
❶ ctx.strokeStyle = "DeepPink";
❷ ctx.lineWidth = 4;
ctx.strokeRect(10, 10, 100, 20);
```

Здесь в строке ❶ мы задали цвет линии `DeepPink` (насыщенный розовый), а в строке ❷ установили ширину линии 4 пикселя. Результат изображен на рис. 13.5.



Рис. 13.5. Розовый контур прямоугольника с шириной линии 4 пикселя

## Рисование линий или путей

Путем называют последовательность линий на «холсте». Чтобы изобразить путь на элементе `canvas`, нужно задать  $x$ - и  $y$ -координаты начала и конца каждой из составляющих путь линий. Используя продуманные

комбинации начальных и конечных координат, можно рисовать разные фигуры. Например, вот как изобразить бирюзовый крестик, показанный на рис. 13.6:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
❶ ctx.strokeStyle = "Turquoise";
❷ ctx.lineWidth = 4;
❸ ctx.beginPath();
❹ ctx.moveTo(10, 10);
❺ ctx.lineTo(60, 60);
❻ ctx.moveTo(60, 10);
❼ ctx.lineTo(10, 60);
❽ ctx.stroke();
```

Begin path —  
начать путь

Line to —  
проводить  
линию к

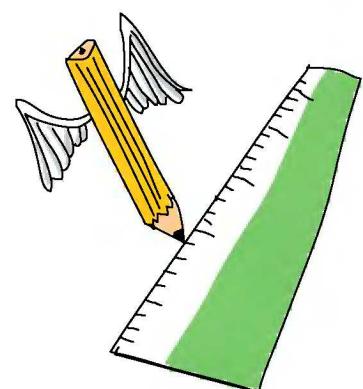


Рис. 13.6. Бирюзовый крестик, нарисованный командами `moveTo` и `lineTo`

В строках ❶ и ❷ мы задали цвет линии и ее толщину. В строке ❸ мы вызываем для контекста рисования (который хранится в `ctx`) метод `beginPath`, который определяет начало рисования нового пути. В строке ❹ вызов метода `moveTo` с двумя аргументами — координатами *x* и *y* — приводит к тому, что наше виртуальное JavaScript-перо отрывается от «холста» и, не оставляя за собой следа, перемещается в точку с этими координатами.

Чтобы нарисовать линию, мы вызываем в строке ❺ метод `lineTo` с координатами *x* и *y*. В результате виртуальное перо опускается на «холст» и, двигаясь к заданным координатам, чертит за собой линию. В данном случае мы рисуем линию из точки (10, 10) в точку (60, 60) — это диагональ, идущая от верхнего левого угла «холста» к его нижнему правому углу, которая является первой линией нашего крестика.

В строке ❻ мы опять вызываем `moveTo`, устанавливая новую позицию для рисования, и в строке ❼ вызываем `lineTo`, чтобы прочертить линию из (60, 10) в (10, 60). Эта диагональная линия, идущая из верхнего правого угла «холста» в нижний левый угол, довершает наш крестик.



Однако это еще не конец! Мы лишь описали то, что собираемся нарисовать, а «холст» по-прежнему пуст. Поэтому в строке ❸ мы вызываем метод `stroke`, благодаря чему линии наконец-то появляются на экране.

### ПОПРОБУЙТЕ!

Попробуйте нарисовать этого веселого человечка при помощи методов `beginPath`, `moveTo`, `lineTo` и `stroke`. Изобразить голову (это квадрат  $20 \times 20$  пикселей с шириной линии 4 пикселя) можно, воспользовавшись методом `strokeRect`.



### Заливка путей цветом

Fill —  
заполнить

Мы уже знакомы с методом для рисования прямоугольных контуров `strokeRect`, методом `fillRect` для рисования заполненных цветом прямоугольников, а также методом `stroke` для обводки путей. Эквивалентом `fillRect` для путей является метод `fill`. Если вам нужно заполнить замкнутый путь цветом, а не просто обвести его, используйте `fill` вместо `stroke`. Например, следующий код рисует домик синего цвета, как на рис. 13.7.

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
ctx.fillStyle = "Blue";
ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(100, 60);
ctx.lineTo(130, 30);
ctx.lineTo(160, 60);
ctx.lineTo(160, 100);
ctx.lineTo(100, 100);
❶ ctx.fill();
```



Рис. 13.7. Синий домик, залитый цветом с помощью метода `fill`

Разберем этот код. Установив синий цвет рисования (`Blue`), мы задаем путь при помощи `beginPath`, а затем методом `moveTo` передвигаем точку начала рисования в позицию  $(100, 100)$ . Далее мы пять раз (по разу для каждого угла домика) вызываем `lineTo` с разными наборами

координат. Последний из вызовов `lineTo` замыкает путь, возвращаясь к первоначальной точке (100, 100).

На рис. 13.8 показан тот же самый домик с подписанными значениями координат.

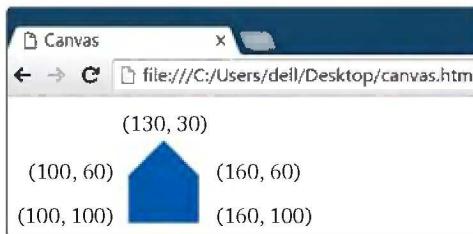


Рис. 13.8. Домик с рис. 13.7, координаты подписаны

И наконец, в строке ❶ вызовом `fill` мы заполнили путь выбранным ранее синим цветом.

## Рисование дуг и окружностей

Кроме прямых линий на «холсте» можно рисовать дуги и окружности — для этого предназначен метод `arc`. Чтобы изобразить окружность, нужно задать ее центр и радиус (расстояние от центра до контура), а также указать, какую часть окружности нужно нарисовать, задав начальный и конечный угол. Таким образом, можно нарисовать как полную окружность, так и ее часть — дугу.

Arc — дуга

Начальный и конечный углы измеряются в радианах. Полная окружность начинается с угла 0 (от ее правого края) и продолжается до угла  $\pi \times 2$  радиан. То есть, чтобы нарисовать полную окружность, нужно передать методу `arc` углы 0 и  $\pi \times 2$ . На рис. 13.9 показана окружность с подписанными значениями углов в радианах, а также градусах. Как  $360^\circ$ , так и  $\pi \times 2$  радиан соответствуют полной окружности.

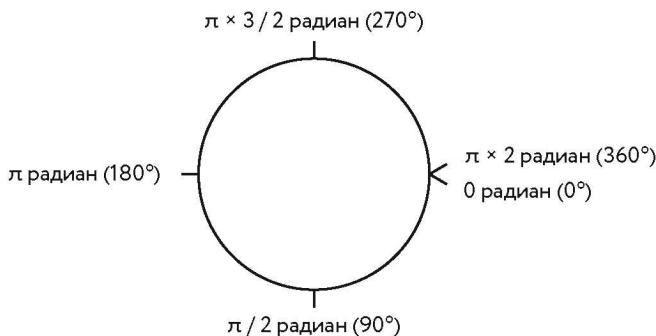


Рис. 13.9. Градусы и радианы, отсчитанные от правого края окружности, по часовой стрелке

Следующий код изобразит на экране четверть окружности, половину окружности и полную окружность, как показано на рис. 13.10.

```
ctx.lineWidth = 2;  
ctx.strokeStyle = "Green";  
  
ctx.beginPath();  
❶ ctx.arc(50, 50, 20, 0, Math.PI / 2, false);  
ctx.stroke();  
  
ctx.beginPath();  
❷ ctx.arc(100, 50, 20, 0, Math.PI, false);  
ctx.stroke();  
  
ctx.beginPath();  
❸ ctx.arc(150, 50, 20, 0, Math.PI * 2, false);  
ctx.stroke();
```

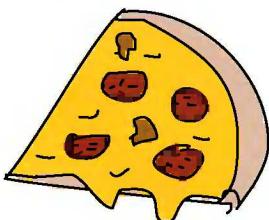
Pi — английская запись числа π



Рис. 13.10. Рисование четверти окружности, половины окружности и полной окружности

В следующих трех разделах мы рассмотрим каждую из этих фигур.

### Рисуем четверть окружности



Первый фрагмент кода рисует четверть окружности. В строке ❶, после вызова `beginPath`, мы вызываем метод `arc`, задав центр окружности в точке (50, 50) и радиус 20 пикселей. Начальный угол мы установили в 0 (то есть дуга будет рисоваться с правого края окружности), а конечный угол в `Math.PI / 2`. `Math.PI` — это обозначение числа π (пи) в JavaScript. Полная окружность — это  $\pi \times 2$  радиан, π радиан соответствует половине окружности,  $\pi / 2$  радиан (значение, которое мы используем в данном случае) — четверть окружности. Начальный и конечный углы показаны на рис. 13.11.

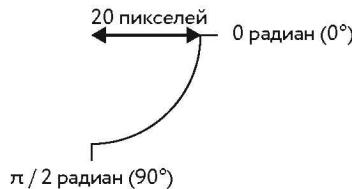


Рис. 13.11. Начальный угол ( $0$  радиан, или  $0^\circ$ ) и конечный угол ( $\pi / 2$  радиан, или  $90^\circ$ ) четверти окружности

В качестве последнего аргумента мы указали `false` — это значит, что рисовать дугу нужно по часовой стрелке. Если вам понадобится рисовать против часовой стрелки, передайте последним аргументом `true`.

## Рисуем половину окружности

Теперь нарисуем половину окружности. При вызове метода `arc` в строке ❷ мы указали центр в точке `(100, 50)` — это на 50 пикселях правее центра первой дуги, `(50, 50)`. Радиус снова равен 20 пикселям, и мы опять начинаем рисовать с угла в  $0$  радиан, однако конечный угол теперь равен `Math.PI`, что соответствует половине окружности. Начальный и конечный углы показаны на рис. 13.12.

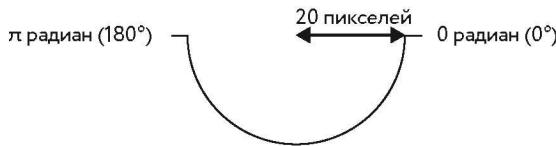


Рис. 13.12. Начальный угол ( $0$  радиан, или  $0^\circ$ ) и конечный угол ( $\pi$  радиан, или  $180^\circ$ ) половины окружности

## Рисуем окружность

В строке ❸ мы рисуем полную окружность. Ее центр — в точке `(150, 50)`, а радиус равен 20 пикселям. Мы рисуем эту окружность, начиная с угла в  $0$  радиан и заканчивая углом в `Math.PI * 2` радиан, что соответствует полному обороту вокруг центра. Начальный и конечный углы показаны на рис. 13.13.

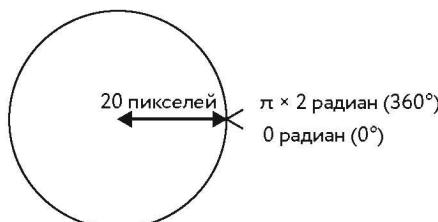


Рис. 13.13. Начальный угол ( $0$  радиан, или  $0^\circ$ ) и конечный угол ( $\pi \times 2$  радиан, или  $360^\circ$ ) полной окружности

## Рисование нескольких окружностей с помощью функции

Если нужно рисовать только полные окружности, метод `arc` несколько сложноват: окружности всегда рисуют от 0 до  $\pi \times 2$  радиан, а направление (по часовой стрелке или против) не имеет значения. Кроме того, чтобы изобразить окружность на экране, каждый раз придется вызывать `ctx.beginPath` перед методом `arc` и `ctx.stroke` после него. Поэтому имеет смысл написать функцию, которая позволит рисовать окружности, не углубляясь в детали, а указывая лишь значения `x`, `y` и радиуса. Давайте так и сделаем.

---

```
var circle = function (x, y, radius) {
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    ctx.stroke();
};
```

---

В теле функции мы первым делом вызываем `ctx.beginPath`, указывая этим, что собираемся создать путь. Затем вызываем метод `ctx.arc`, передавая ему значения аргументов `x`, `y` и `radius`. Как и раньше, мы используем 0 для начального угла и `Math.PI * 2` для конечного угла, а также `false` для рисования по часовой стрелке.

Теперь с помощью этой функции можно изобразить любое количество окружностей, задавая лишь координаты их центров и радиусы. Например, следующий код рисует набор разноцветных окружностей одна в другой:

---

```
ctx.lineWidth = 4;
ctx.strokeStyle = "Red";
circle(100, 100, 10);
ctx.strokeStyle = "Orange";
circle(100, 100, 20);
ctx.strokeStyle = "Yellow";
circle(100, 100, 30);
ctx.strokeStyle = "Green";
circle(100, 100, 40);
ctx.strokeStyle = "Blue";
circle(100, 100, 50);
ctx.strokeStyle = "Purple";
circle(100, 100, 60);
```

---

Результат будет выглядеть как на рис. 13.14. Сначала мы задаем ширину линии — 4 пикселя. Затем устанавливаем `strokeStyle` в `Red` (красный цвет) и вызываем функцию `circle`, чтобы нарисовать

окружность в точке (100, 100) с радиусом 10 пикселей. Это центральное красное кольцо.

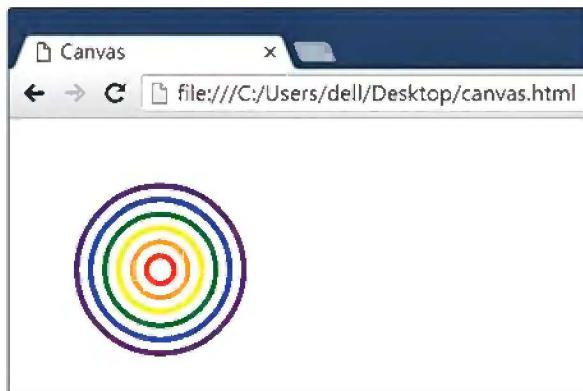


Рис. 13.14. Цветные концентрические окружности, нарисованные с помощью функции `circle`

Таким же образом мы рисуем оранжевую окружность — координаты остаются прежними, но радиус теперь равен 20 пикселям. Затем желтую окружность — снова там же, но с радиусом 30 пикселей. Три последние окружности (зеленая, синяя и фиолетовая) также имеют общий центр с предыдущими, а их радиусы последовательно возрастают.

### ПОПРОБУЙТЕ!

Как изменить функцию `circle`, чтобы она заполняла окружности цветом, а не обводила их? Добавьте четвертый аргумент — булево значение, которое определяет, нужно ли заполнять окружность или рисовать контур. Можно назвать этот аргумент `fillCircle`.

Нарисуйте с помощью измененной функции такого снеговика, используя как заполненные окружности, так и пустые.



### Что мы узнали

В этой главе мы познакомились с HTML-элементом `canvas`. Благодаря ему мы можем с легкостью изображать на экране прямоугольники, линии и окружности, управляя их положением, шириной линий, цветом и т. д.

В следующей главе мы узнаем, как с помощью некоторых приемов, изученных в девятой главе, сделать так, чтобы картинки двигались.



## УПРАЖНЕНИЯ

Выполните эти упражнения, чтобы попрактиковаться в рисовании с помощью canvas.

### #1. Функция, рисующая снеговика

**Draw snowman** —  
нарисовать снеговика

Создайте на основе вашего кода для рисования снеговика (см. с. 205) функцию drawSnowman, которая рисует снеговика в указанной позиции на «холсте», — так, чтобы в результате вызова:

```
drawSnowman(50, 50);
```

снеговик появился в позиции (50, 50).

### #2. Рисование по массиву точек

**Draw points** —  
нарисовать точки

Напишите функцию drawPoints, которая принимает массив с координатами точек:

```
var points = [[50, 50], [50, 100], [100, 100], [100, 50], [50, 50]];
drawPoints(points);
```

и рисует линию, соединяющую эти точки. В данном случае она должна прочертить линию от (50, 50) до (50, 100) и далее до (100, 100), до (100, 50) и обратно до (50, 50).

Теперь передайте в эту функцию следующий массив:

**Mystery** —  
загадочные

```
var mysteryPoints = [[50, 50], [50, 100], [25, 120], [100, 50],
[70, 90], [100, 90], [70, 120]];
drawPoints(mysteryPoints);
```

Подсказка: в points[0][0] находится первая x-координата, а в points[0][1] первая y-координата.

### #3. Рисование мышкой

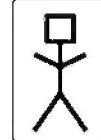
С помощью jQuery и события mousemove напишите код, рисующий окружность радиусом в 3 пикселя под курсором мышки каждый раз, когда вы передвигаете мышку над «холстом». Поскольку это событие возникает при малейшем перемещении курсора, окружности по мере движения мышки будут сливаться в линию.

Подсказка: чтобы вспомнить, как обрабатывать событие mousemove, загляните в десятую главу.

#### #4. Человечек в «Виселице»

В седьмой главе мы написали вариант игры «Виселица». Теперь вы можете сделать ее более похожей на настоящую игру, рисуя части тела человечка всякий раз, когда игрок вводит неверную букву.

Подсказка: отслеживайте, сколько раз игрок указал букву, которой нет в слове. Напишите функцию, которая принимает это число в качестве аргумента и в зависимости от его значения рисует более или менее завершенного человечка.



# 14

## АНИМАЦИИ С ПОМОЩЬЮ CANVAS

Создание анимаций с помощью canvas в JavaScript похоже на покадровую мультипликацию — вы рисуете картинку, делаете паузу, стираете картинку и затем перерисовываете ее в новом месте. Казалось бы, слишком много действий, однако JavaScript может обновлять положение картинки очень быстро, так, что получается плавная анимация. В десятой главе мы узнали, как анимировать элементы DOM, а в этой главе будем анимировать рисунки с canvas.

### Движение по странице

Давайте воспользуемся элементом canvas и функцией setInterval, чтобы изобразить плавно двигающийся по странице квадрат. Создайте новый файл под названием *canvasanimation.html* и добавьте в него следующий HTML-код:

---

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Анимация с canvas</title>
</head>
<body>
    <canvas id="canvas" width="200" height="200"></canvas>
    <script>
        // Здесь будет JavaScript-код
    </script>
</body>
</html>
```

---

Animation —  
анимация

Теперь введите между тегов <script> такой код:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");

var position = 0;

setInterval(function () {
①   ctx.clearRect(0, 0, 200, 200);
②   ctx.fillRect(position, 0, 20, 20);

③   position++;
④   if (position > 200) {
      position = 0;
    }
⑤ }, 30);
```

**Clear rect** —  
очистить  
прямоугольник

В самом начале создаются «холст» и контекст рисования. Затем мы создаем переменную `position` и даем ей значение 0 (`var position = 0`). Эта переменная понадобится, чтобы управлять перемещением квадрата слева направо.

Затем мы вызываем `setInterval`, чтобы запустить анимацию. Первый аргумент `setInterval` — это функция, которая при каждом вызове рисует новый квадрат.

## Очистка «холста»

В теле переданной `setInterval` функции, в строке ①, мы вызываем `clearRect`, чтобы очистить прямоугольную область на «холсте». Метод `clearRect` принимает четыре аргумента, которые задают позицию и размер области для очистки. Аналогично `fillRect` первые два аргумента соответствуют *x*- и *y*-координатам верхнего левого угла прямоугольника, а два последних аргумента — его ширине и высоте. Вызов `ctx.clearRect(0, 0, 200, 200)` очищает квадрат  $200 \times 200$  пикселей, начиная с верхнего левого угла «холста». Поскольку наш «холст» как раз размером  $200 \times 200$  пикселей, в результате он очистится целиком.



## Рисование квадрата

После очистки «холста», в строке ②, мы вызываем `ctx.fillRect(position, 0, 20, 20)`, чтобы изобразить квадрат со стороной 20 пикселей в точке `(position, 0)`. Сразу после старта программы квадрат будет нарисован в позиции `(0, 0)`, поскольку `position` в начале равна 0.

## Изменение позиции

Далее в строке ❸ мы увеличиваем позицию на 1 командой `position++`. Затем в строке ❹ мы командой `if (position > 200)` проверяем, не стала ли переменная `position` больше 200, и если стала, сбрасываем ее в 0.

## Просмотр анимации в браузере

Когда вы загрузите эту страницу в браузер, `setInterval` будет вызывать указанную функцию каждые 30 миллисекунд, или около 33 раз в секунду (такой интервал задан во втором аргументе `setInterval` в строке ❺). При каждом вызове функция очищает «холст», рисует квадрат в позиции (`position, 0`) и увеличивает на 1 переменную `position`. В результате квадрат плавно движется по экрану. Когда он, пройдя 200 пикселей, достигнет края холста, его позиция обнулится.

На рис. 14.1 показаны первые пять шагов анимации в области верхнего левого угла «холста».

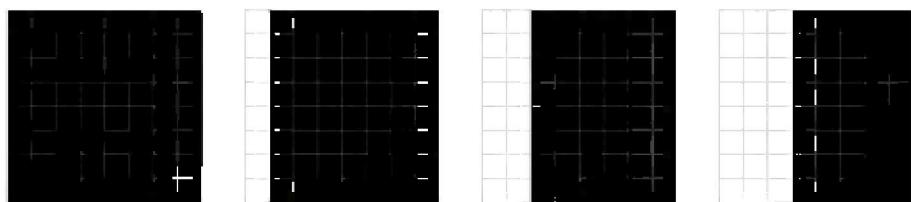


Рис. 14.1. Область верхнего левого угла «холста» в увеличении для первых пяти шагов анимации. На каждом шаге `position` увеличивается на 1, и квадрат передвигается на 1 пиксель вправо

## Изменение размера квадрата

Немного изменив последний пример, можно изобразить квадрат, который не движется, а увеличивается в размере. Вот как будет выглядеть в этом случае код:

---

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var size = 0;
setInterval(function () {
    ctx.clearRect(0, 0, 200, 200);
    ctx.fillRect(0, 0, size, size);
    size++;
    if (size > 200) {
        size = 0;
    }
}, 30);
```

---

Как видите, мы сделали две вещи. Во-первых, вместо переменной `position` теперь используется переменная `size`, предназначенная для управления размером квадрата. Во-вторых, вместо того чтобы при помощи этой переменной установить горизонтальную позицию, мы командой `ctx.fillRect(0, 0, size, size)` задаем ширину и высоту. В результате в верхнем левом углу «холста» рисуется квадрат со стороной `size`. Поскольку изначально `size` равен 0, квадрат невидим. При следующем вызове функции `size` будет равна 1, что даст нам квадрат со стороной в 1 пиксель. И так при каждом новом вызове будет рисоваться квадрат на 1 пиксель больше предыдущего. Запустив код, вы увидите, как в левом верхнем углу страницы возникает квадрат и растет до тех пор, пока не заполнит собой весь «холст». Затем, когда выполнится условие `size > 200`, квадрат исчезнет и снова начнет расти из левого верхнего угла.

На рис. 14.2 показана область левого верхнего угла «холста» в увеличении для первых пяти шагов анимации.

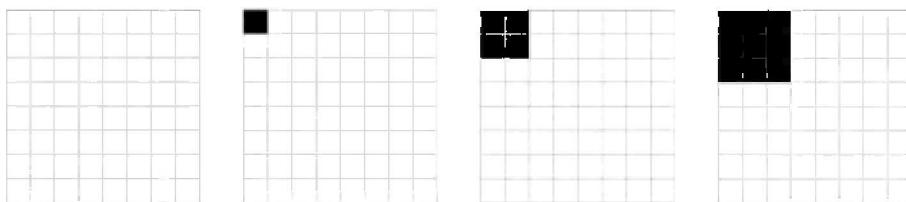


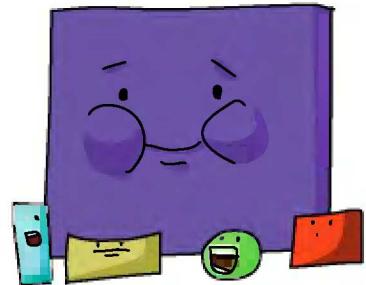
Рис. 14.2. На каждом шаге анимации `size` увеличивается на 1, и также на 1 увеличиваются ширина и высота квадрата

## Случайная пчела

Теперь, когда мы знаем, как двигать объекты по экрану и увеличивать их, давайте изобразим кое-что забавное — а именно пчелу, которая случайным образом перемещается по «холсту»! Мы сделаем пчелу из нескольких окружностей, вот так:



Работать эта анимация будет примерно так же, как в примере с квадратом: задаем начальную позицию и далее для каждого шага анимации очищаем «холст», рисуем пчелу в текущей позиции и обновляем позицию. Однако чтобы пчела двигалась случайным образом, нам придется использовать более сложную логику изменения позиции, чем для квадрата. Несколько следующих разделов будут посвящены созданию кода для этой анимации.



## Новая функция circle

Изображение нашей пчелы состоит из нескольких окружностей, поэтому сначала мы напишем функцию, рисующую заполненные или обведенные окружности:

```
var circle = function (x, y, radius, fillCircle) {
    ctx.beginPath();
    ❶  ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    ❷  if (fillCircle) {
    ❸      ctx.fill();
    ❹  } else {
    ❺      ctx.stroke();
    ❻  }
};
```

Функция принимает четыре аргумента: `x`, `y`, `radius` и `fillCircle`. Похожую функцию мы использовали в главе 13, однако теперь добавили еще один аргумент — `fillCircle`. При вызове функции этот аргумент должен равняться `true` или `false` — это указывает, следует ли рисовать заполненную окружность или только ее контур.

В теле функции, в строке ❶, мы используем метод `arc` для создания окружности с центром в точке (`x`, `y`) и радиусом, равным `radius`. Затем в строке ❷ проверяем, соответствует ли аргументу `fillCircle` значение `true`. Если это так, мы рисуем заполненную окружность, вызывая в строке ❸ `ctx.fill`. Иначе рисуем контур вызовом `ctx.stroke` в строке ❹.

## Рисуем пчелу

**Draw bee** —  
нарисовать  
пчелу

Теперь создадим функцию `drawBee`, которая будет рисовать пчелу в координатах (`x`, `y`), используя для этого функцию `circle`. Вот код функции `drawBee`:

```
var drawBee = function (x, y) {
    ❶  ctx.lineWidth = 2;
    ctx.strokeStyle = "Black";
    ctx.fillStyle = "Gold";

    ❷  circle(x, y, 8, true);
    circle(x, y, 8, false);
    circle(x - 5, y - 11, 5, false);
    circle(x + 5, y - 11, 5, false);
    circle(x - 2, y - 1, 2, false);
    circle(x + 2, y - 1, 2, false);
};
```



В первом фрагменте кода, начиная со строки ❶, мы устанавливаем нужные для рисования свойства `lineWidth`, `strokeStyle` и `fillStyle`. Для `lineWidth` задаем значение 2 пикселя, а для `strokeStyle` цвет Black (черный). Это значит, что окружности, которые мы используем для рисования тела пчелы, ее крыльшек и глаз, будут с жирной черной обводкой. А `fillStyle` мы зададим значение `Gold` (золотой), чтобы закрасить тело пчелы приятным желтым цветом.

В следующем фрагменте кода, начиная со строки ❷, мы рисуем набор окружностей, из которых состоит наша пчела. Давайте рассмотрим каждую окружность по отдельности.

Первая окружность — это заполненное цветом тело пчелы с центром в точке  $(x, y)$ , радиусом 8 пикселей:

```
circle(x, y, 8, true);
```

Поскольку мы задали `fillStyle` значение `Gold`, окружность будет заполнена желтым цветом, вот так:



Вторая окружность — это черный контур вокруг тела пчелы, его размер и позиция такие же, как у первой окружности:

```
circle(x, y, 8, false);
```

Вместе с первой окружностью получится вот что:



Теперь рисуем крыльшки. Первое крыло — это окружность-контур с радиусом 5 пикселей, центр которой на 5 пикселей левее и на 11 пикселей выше центра тела пчелы. Второе крыло точно такое же, однако его центр на 5 пикселей правее центра тела.

```
circle(x - 5, y - 11, 5, false);
circle(x + 5, y - 11, 5, false);
```

Вместе с этими окружностями наша пчела выглядит так:



И наконец, рисуем глаза. Первый на 2 пикселя левее и на 1 пиксель выше центра тела, с радиусом 2 пикселя. Второй глаз такой же, но на 2 пикселя правее центра тела.

```
circle(x - 2, y - 1, 2, false);
circle(x + 2, y - 1, 2, false);
```

В итоге получается пчела, центр которой задается аргументами `x` и `y`, переданными в функцию `drawBee`.



## Изменение позиции пчелы

Update —  
обновить

Coordinate —  
координата

Для случайного изменения `x`- и `y`-координат пчелы — чтобы создавалось впечатление, будто она летает туда-сюда по «холсту», — мы создадим функцию `update`. Эта функция принимает единственную координату: мы будем обновлять по одной координате за раз, чтобы пчела двигалась случайным образом вправо-влево и вверх-вниз. Вот код функции `update`:

```
var update = function (coordinate) {
  ❶  var offset = Math.random() * 4 - 2;
  ❷  coordinate += offset;

  ❸  if (coordinate > 200) {
    coordinate = 200;
  }
  ❹  if (coordinate < 0) {
    coordinate = 0;
  }

  ❺  return coordinate;
};
```



Изменяем координату на величину смещения

В строке ❶ мы создаем переменную `offset` — это смещение, определяющее, на сколько нужно изменить текущую координату. Мы вычисляем его как `Math.random() * 4 - 2`, получая случайное число в диапазоне от `-2` до `2`. Логика такая: сам по себе вызов `Math.random()` вернет случайное значение от `0` до `1`, следовательно, `Math.random() * 4` даст число от `0` до `4`. И вычтя `2`, мы получим наше случайное число от `-2` до `2`.

В строке ❷ мы используем команду `coordinate += offset`, чтобы изменить координату на величину смещения `offset`. Если смещение

положительное, координата увеличится, если отрицательное — уменьшится. Например, если координата равна 100, а смещение равно 1, после выполнения кода в строке ❷ координата примет значение 101. Однако если координата равна 100, а смещение равно  $-1$ , новым значением координаты станет 99.

#### Проверка выхода за границу

В строках ❸ и ❹ мы проверяем, не вылетела ли пчела за границу «холста», то есть не стала ли координата больше 200 или меньше 0. Если она стала больше 200, мы присваиваем ей значение 200, а если стала меньше 0, сбрасываем ее в 0.

#### Возвращаем обновленную координату

И наконец, в строке ❽ мы возвращаем координату с помощью `return`. Это дает возможность использовать новое значение в остальной части программы. Впоследствии мы будем применять это полученное из функции `update` значение таким образом:

---

```
x = update(x);
y = update(y);
```

---

## Анимируем пчелу

Теперь, когда у нас есть функции `circle`, `drawBee` и `update`, можно написать код для анимации нашей неугомонной пчелы.

---

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");

var x = 100;
var y = 100;

setInterval(function () {
    ❶   ctx.clearRect(0, 0, 200, 200);

    ❷   drawBee(x, y);
    ❸   x = update(x);
    ❹   y = update(y);

    ❺   ctx.strokeRect(0, 0, 200, 200);
}, 30);
```

---

Как обычно, сначала мы получаем «холст» `canvas` и контекст рисования `ctx`. Затем создаем переменные `x` и `y`, задавая им обеим значение 100.

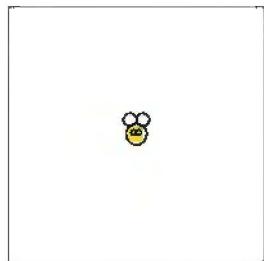


Рис. 14.3. Пчела в позиции (100, 100)



Рис. 14.4. Случайная анимация пчелы

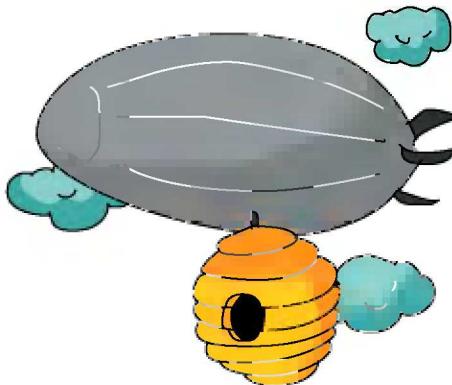
Таким образом, начальная позиция нашей пчелы — точка (100, 100), то есть центр «холста», как показано на рис. 14.3.

Затем мы вызываем `setInterval`, передавая первым аргументом функцию, которую надо будет вызывать каждые 30 миллисекунд. Внутри этой функции, в строке ①, мы первым делом очищаем «холст» вызовом `clearRect`. Затем в строке ② мы рисуем пчелу в позиции (x, y). При первом вызове функции пчела будет нарисована в позиции (100, 100), как на рис. 14.3, а при каждом последующем вызове — в новой, случайному образом измененной позиции (x, y).

Далее, начиная со строки ③, мы обновляем значения x и y. Функция `update` принимает число, добавляет к нему случайное значение в диапазоне от -2 до 2 и возвращает обновленное число. А код `x = update(x)` фактически означает «изменить x на небольшую случайную величину».

И наконец, в строке ④ мы вызываем `strokeRect`, чтобы обвести рамкой границы «холста» — чтобы было ясно, когда пчела подлеет к границе. Без этой рамки границы холста будут невидимы.

Запустив этот код, вы увидите желтую пчелу, которая случайным образом перемещается по «холсту». На рис. 14.4 показано несколько кадров анимации.



## Отскакивающий мяч

Теперь давайте изобразим летающий по «холсту» мяч. При столкновении с одной из границ он будет отскакивать назад, меняя направление, словно резиновый.

Мы создадим для нашего мяча JavaScript-объект, написав для этого конструктор Ball. Объект будет хранить скорость мяча и направление его движения с помощью двух свойств, xSpeed и ySpeed. Горизонтальная скорость мяча будет определяться значением xSpeed, а вертикальная — ySpeed.

Создайте для этой анимации файл под названием *ball.html* и добавьте в него следующий HTML-код:

---

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Прыгающий мяч</title>
</head>
<body>
    <canvas id="canvas" width="200" height="200"></canvas>
    <script>
        // Здесь будет JavaScript-код
    </script>
</body>
</html>
```

---

## Конструктор Ball

Первым делом напишем конструктор Ball, с помощью которого мы будем создавать наш мяч. Введите следующий код между тегов `<script>` в файле *ball.html*:

---

```
var Ball = function () {
    this.x = 100;
    this.y = 100;
    this.xSpeed = -2;
    this.ySpeed = 3;
};
```

---

Конструктор довольно простой: он задает начальную позицию мяча (`this.x` и `this.y`), его горизонтальную скорость (`this.xSpeed`) и скорость вертикальную (`this.ySpeed`). Начальной позицией будет точка (100, 100) — это центр нашего 200 × 200-пиксельного «холста».

Ball — мяч  
Speed —  
скорость

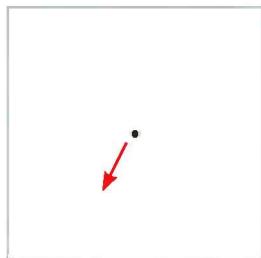


Рис. 14.5. Начальное положение мяча, направление движения показано стрелкой

Начальное значение `this.xSpeed` равно `-2`, это значит, что мяч будет смещаться на 2 пикселя влево на каждом шаге анимации. Начальное значение `this.ySpeed` равно `3`, то есть на каждом шаге анимации мяч будет также смещаться на 3 пикселя вниз. Следовательно, мяч будет двигаться по диагонали (3 пикселя вниз и 2 влево).

На рис. 14.5 показано начальное положение мяча и направление движения.

## Рисуем мяч

Далее напишем метод `draw` для отрисовки мяча. Мы добавим этот метод к свойству `prototype` конструктора `Ball`, чтобы все объекты, созданные при помощи `Ball`, могли его использовать:

```
var circle = function (x, y, radius, fillCircle) {
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    if (fillCircle) {
        ctx.fill();
    } else {
        ctx.stroke();
    }
};
Ball.prototype.draw = function () {
    circle(this.x, this.y, 3, true);
};
```

Сначала идет код функции `circle` — все той же, из раздела «Новая функция `circle`» со с. 212. Затем мы добавили к `Ball.prototype` новый метод `draw`. Этот метод просто-напросто вызывает `circle(this.x, this.y, 3, true)`, рисуя окружность с центром в точке `(this.x, this.y)`, иными словами, в позиции мяча. Радиус окружности — 3 пикселя, а в качестве последнего аргумента мы передаем `true`, чтобы окружность была заполненной.

## Перемещение мяча

Чтобы мяч двигался, нам нужно лишь обновлять значения свойств `x` и `y` в соответствии с текущей скоростью. Делать это мы будем в теле метода `move`, вот такого:

```
Ball.prototype.move = function () {
    this.x += this.xSpeed;
    this.y += this.ySpeed;
};
```

Мы используем команду `this.x += this.xSpeed`, чтобы прибавить значение горизонтальной скорости к `this.x`. Аналогично `this.y += this.ySpeed` прибавляет вертикальную скорость к `this.y`. Например, в самом начале анимации мяч находится в позиции (100, 100), `this.xSpeed` равняется -2, `this.ySpeed` равняется 3. Метод `move` при его вызове вычтет 2 из значения `x` и прибавит 3 к значению `y`, в результате чего мяч окажется в позиции (98, 103), то есть переместится на 2 пикселя влево и на 3 пикселя вниз, как показано на рис. 14.6.

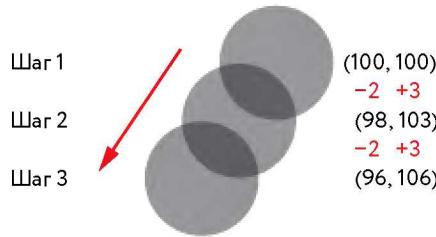
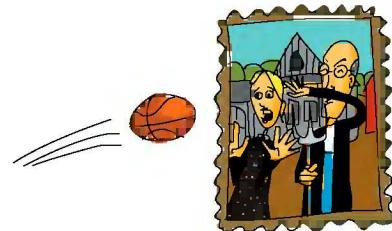
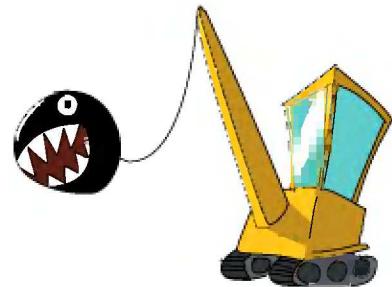


Рис. 14.6. Первые три шага анимации.  
Показано, как меняются значения свойств `x` и `y`



## Отскоки мяча

На каждом шаге анимации нужно проверять, не столкнулся ли мяч с границей «холста». Если столкнулся, следует обновить свойство `xSpeed` или `ySpeed`, инвертировав его значение (то есть умножив на -1). Например, при столкновении с нижней границей мы должны инвертировать `this.ySpeed`: если в `this.ySpeed` находится значение 3, его надо поменять на -3. А если в `this.ySpeed` уже значение -3, то инвертирование, напротив, поменяет его на 3.

Мы дадим этому методу имя `checkCollision` (проверка столкновения):

---

```

Ball.prototype.checkCollision = function () {
①  if (this.x < 0 || this.x > 200) {
    this.xSpeed = -this.xSpeed;
}
②  if (this.y < 0 || this.y > 200) {
    this.ySpeed = -this.ySpeed;
}
};

```

---

В строке ① мы выясняем, не столкнулся ли мяч с левой или правой границей, сравнивая свойство `x` с 0 (если `x` меньше 0, мяч столкнулся с левой границей) и с 200 (если `x` больше 200, мяч столкнулся с правой границей). Если любая из этих проверок даст `true`, значит мяч начал выходить за пределы «холста» и его горизонтальное направление нужно инвертировать. Мы делаем это, задавая свойству `this.xSpeed` значение `-this.xSpeed`. Например, если `this.xSpeed` равняется `-2` и мяч столкнулся с левой границей, `this.xSpeed` примет значение `2`.

В строке ② мы выполняем аналогичную проверку для верхней и нижней границ. Если `this.y` меньше 0 или больше 200, значит мяч столкнулся или с верхней, или с нижней границей соответственно. В этом случае мы задаем `this.ySpeed` значение `-this.ySpeed`.

На рис. 14.7 показано, что происходит при столкновении мяча с левой границей. Сначала значение `this.xSpeed` равняется `-2`, но после столкновения оно меняется на `2`. Однако в `this.ySpeed` по-прежнему остается значение `3`.

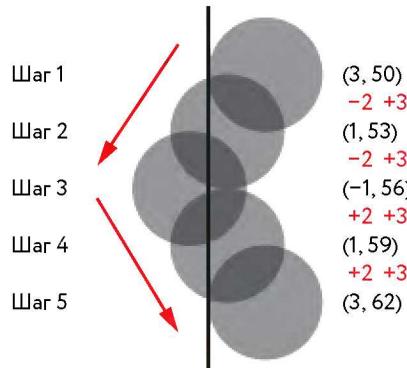


Рис. 14.7. Так меняется `this.xSpeed` после столкновения с левой границей

Как видно по рис. 14.7, на третьем шаге мяч сталкивается с границей — его центральная точка уходит за границу «холста». При этом часть мяча станет невидимой, однако этот кадр промелькнет столь быстро, что во время анимации будет едва заметным.

## Анимация мяча

Теперь можно написать код, отвечающий за анимацию. Этот код создает объект-мяч и с помощью setInterval вызывает методы отрисовки и обновления позиции мяча на каждом шаге анимации.

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");

❶ var ball = new Ball();

❷ setInterval(function () {
❸   ctx.clearRect(0, 0, 200, 200);

❹   ball.draw();
   ball.move();
   ball.checkCollision();

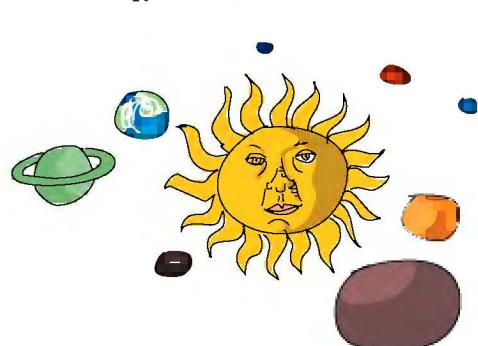
❺   ctx.strokeRect(0, 0, 200, 200);
❻ }, 30);
```

Как обычно, сначала мы получаем элемент canvas и контекст рисования. Затем в строке ❶ создаем объект-мяч вызовом new Ball() и сохраняем его в переменной ball. Далее в строке ❷ мы вызываем setInterval, передавая первым аргументом функцию, а вторым (в строке ❻) — число 30. Как нам уже известно, это означает «вызывай эту функцию каждые 30 миллисекунд».

Функция, которую мы передаем setInterval, выполняет несколько задач. Сначала в строке ❸ она очищает «холст» командой ctx.clearRect(0, 0, 200, 200). После этого, начиная со строки ❹, она вызывает методы объекта ball: draw, move, и checkCollision. Метод draw рисует мяч в его текущей позиции (x, y). Метод move обновляет позицию мяча на основе значений xSpeed и ySpeed. И наконец, метод checkCollision меняет направление движения мяча, если тот столкнулся с границей «холста».

Последнее, что делает вызываемая через setInterval функция, — в строке ❺ рисует вокруг «холста» рамку командой ctx.strokeRect(0, 0, 200, 200) — чтобы границы, от которых отскакивает мяч, были видимы.

Когда вы запустите этот код, мяч сразу же начнет двигаться вниз и влево. Затем он врезается в нижнюю границу и отскочит вверх и влево. И так он будет летать по «холstu», отскакивая от границ, пока вы не закроете окно браузера.



## Что мы узнали

В этой главе мы объединили наши знания об анимации из главы 11 с навыками рисования на «холсте», создав несколько анимаций на основе элемента `canvas`. Начали мы с простого двигающегося и меняющего размер квадрата. Затем мы научили пчелу случайнym образом перемещаться по экрану и, наконец, создали анимацию отскакивающего от стен мяча.

Все эти анимации устроены одинаково: мы рисуем фигуру определенного размера в определенной позиции, потом изменяем этот размер или позицию, а потом очищаем «холст» и перерисовываем фигуру. Кроме того, нужно следить за `x`- и `y`-координатами движущихся по «холсту» объектов. В примере с пчелой мы изменяли `x`- и `y`-координаты, складывая их со случайнym значением. А в случае с отскакивающим мячом — прибавляли к координатам значения `xSpeed` и `ySpeed`. В следующей главе мы добавим нашим экспериментам с «холстом» интерактивности, что позволит управлять изображениями с клавиатуры.

## УПРАЖНЕНИЯ

Вот некоторые идеи по усовершенствованию анимации отскакивающего мяча из этой главы.

### #1. Увеличение размеров «холста»

Наш «холст» размером  $200 \times 200$  пикселей как-то маловат. Что если увеличить его до  $400 \times 400$  пикселей?

Вместо того чтобы вводить числовые размеры «холста» в тексте программы, можно создать переменные для ширины и высоты и брать их значения из объекта `canvas`. Используйте следующий код:

---

```
var width = canvas.width;
var height = canvas.height;
```

---

Если везде в программе использовать эти переменные вместо чисел, вам останется лишь изменить атрибуты элемента `canvas` в HTML-коде, чтобы опробовать новый размер «холста». Попробуйте сделать его шириной в 500 пикселей и высотой в 300. Ну как, программа все еще работает?

### #2. Случайные значения `this.xSpeed` и `this.ySpeed`

Чтобы анимация была интереснее, в конструкторе `Ball` задавайте свойствам `this.xSpeed` и `this.ySpeed` разные случайные значения в диапазоне от -5 до 5.

### #3. Больше мячиков!

Вместо анимации одного мячика создайте пустой массив и в цикле `for` добавьте в него 10 мячей. Теперь измените функцию, вызываемую по `setInterval`, использовав цикл `for` для перемещения и проверки столкновений каждого из мячей.

### #4. Цветные мячи

Как насчет разноцветных отскакивающих мячиков? Задайте в конструкторе `Ball` новое свойство `color` (цвет) и используйте его для задания цвета в методе `draw`. Воспользуйтесь функцией `pickRandomWord` из восьмой главы, чтобы задать каждому из мячей случайный цвет из такого массива:

---

```
var colors = ["Red", "Orange", "Yellow", "Green", "Blue", "Purple"];
```

---

# 15

## УПРАВЛЕНИЕ АНИМАЦИЯМИ С КЛАВИАТУРЫ

Мы уже научились работать с «холстом», рисовать и раскрашивать объекты, заставлять их двигаться, отскакивать и увеличиваться в размере. Теперь давайте оживим наши программы, добавив в них интерактивности!

Из этой главы вы узнаете, как научить анимации реагировать на нажатия кнопок клавиатуры — чтобы игрок мог управлять анимацией с помощью клавиш-стрелок или иных заданных в программе клавиш (скажем, общепринятого для игр сочетания W, A, S, D). Например, вместо того чтобы мяч сам по себе летал по экрану, мы можем дать игроку возможность управлять его перемещениями с помощью клавиш-стрелок.

### События клавиатуры

Состояние клавиатуры можно отслеживать в JavaScript с помощью *событий клавиатуры*. Каждый раз, когда пользователь нажимает клавишу, генерируется событие, во многом напоминающее события мышки, о которых мы говорили в десятой главе. Тогда мы использовали jQuery, чтобы определить, где находился курсор в момент возникновения события. Для определения нажатой клавиши при возникновении события клавиатуры также можно воспользоваться jQuery. Например, в этой главе мы научимся перемещать мяч влево, вправо, вверх или вниз, когда пользователь нажимает на клавиатуре стрелку влево, вправо, вверх или вниз.

Мы будем отслеживать событие `keydown`, которое возникает при нажатии клавиши, используя jQuery для задания обработчика этого события. Таким образом, при каждом возникновении события `keydown`

**Key down** —  
нажатая  
клавиша

наш обработчик сможет определить нажатую клавишу и нужным образом на это отреагировать.

## Создаем HTML-файл

Для начала создайте HTML-файл со следующим кодом, сохранив его под именем `keyboard.html`.

Keyboard —  
клавиатура

```
<!DOCTYPE html>
<html>
<head>
    <title>Keyboard input</title>
</head>
<body>
    <canvas id="canvas" width="400" height="400"></canvas>
    <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
    <script>
        // Здесь будет JavaScript-код
    </script>
</body>
</html>
```

Keyboard  
input —  
входящая  
информация  
с клавиатуры

## Добавим обработчик события keydown

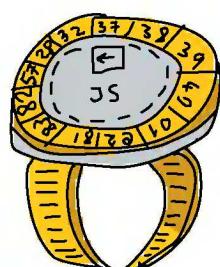
Теперь добавим JavaScript-код, реагирующий на событие `keydown`. Введите этот код между тегов `<script>` в файле `keyboard.html`.

```
$( "body" ).keydown( function (event) {
    console.log(event.keyCode);
});
```

В первой строке мы использовали jQuery-функцию `$` для поиска HTML-элемента `body`, вызвав затем метод `keydown`. Аргумент этого метода — функция, которая будет вызываться при каждом нажатии клавиши. Информация о событии передается в функцию через объект `event`. В данном случае мы хотим узнать, какая клавиша была нажата, и эти сведения хранятся в свойстве `keyCode` объекта `event`.

В теле функции мы используем `console.log`, чтобы вывести значение свойства `keyCode`: это число, соответствующее нажатой клавише. Каждой из клавиш соответствует уникальный числовой код, например код клавиши «пробел» равен 32, а код стрелки влево — 37.

Отредактировав файл `keyboard.html`, сохраните его и загрузите в браузер. Откройте консоль, чтобы видеть, что печатает программа, и кликните по основному окну браузера, чтобы JavaScript мог отслеживать



нажатия клавиш. Теперь понажимайте клавиши на клавиатуре — в консоль будут выводиться соответствующие этим клавишам коды.

Например, если вы напечатаете «привет», в консоли появится нечто вроде:

---

```
71  
72  
66  
68  
84  
78
```

---

У каждой нажатой клавиши свой код — для «П» это 71, для «Р» это 72 и т. д.

### ПОПРОБУЙТЕ!

Понажмайте разные клавиши, чтобы узнать их коды. Какие значения соответствуют стрелкам вверх, вниз, влево и вправо? А клавишам SHIFT и ENTER? У каждой буквенной и цифровой клавиши тоже есть свой уникальный код.

## Перевод кодов в названия с помощью объекта

Чтобы работать с клавиатурой было проще, воспользуемся объектом для перевода кодов клавиш в их названия. В следующем примере мы создадим объект keyNames, ключи которого соответствуют кодам клавиш, а значения — их англоязычным названиям. Замените в файле *keyboard.html* прежний JavaScript-код на вот такой:

Key name —  
название  
клавиши

---

```
var keyNames = {  
    32: "space", //пробел  
    37: "left", //влево  
    38: "up", //вверх  
    39: "right", //вправо  
    40: "down" //вниз  
};  
  
$("body").keydown(function (event) {  
    ① console.log(keyNames[event.keyCode]);  
});
```

---

Сначала мы создали объект keyNames, добавив ему в качестве свойств числа 32, 37, 38, 39 и 40. Это пары «ключ-значение», где ключами являются коды клавиш (такие как 32, 37 и т. д.), а соответствующими

значениями — названия клавиш (например, "space" для «пробела» и "left" для стрелки влево).

Теперь этот объект можно использовать для поиска названия клавиши по ее коду. Например, чтобы узнать название для кода 32, введите keyNames[32] и получите строку "space".

В строке ❶ мы используем объект keyNames в теле обработчика событий, чтобы найти название только что нажатой клавиши. Если находящийся в event.keyCode код есть среди ключей объекта keyNames, обработчик выведет в консоль соответствующее название, иначе он напечатает undefined.

Загрузите файл *keyboard.html* в браузер. Откройте консоль, кликните по основному окну браузера и понажимайте разные клавиши. При нажатии на любую из пяти клавиш, перечисленных в объекте keyNames (то есть стрелок или пробела), программа напечатает их название. Иначе она выведет undefined.

### ПОПРОБУЙТЕ!

Добавьте в объект keyNames дополнительные пары «ключ-значение», чтобы можно было получить названия других клавиш. Пусть это будут коды и названия для SHIFT, ENTER/RETURN, и ALT/OPTION.

## Управляем мячом с клавиатуры

Теперь, зная, как определить нажатую клавишу, мы можем написать программу для управления мячом с клавиатуры. Наша программа будет рисовать мяч и перемещать его вправо. Нажатия клавиш-стрелок будут менять направление мяча, а нажатие на «пробел» остановит его движение. Если мяч уйдет за границу «холста», он появится с противоположной стороны. Например, если мяч уйдет за правую границу, он снова появится у левой границы, продолжая двигаться в прежнем направлении, как показано на рис. 15.1.

Мы воспользуемся объектом под названием keyActions, чтобы определять, какая клавиша нажата, и использовать эту информацию для смены направления полета мяча. А для периодического обновления позиции мяча и перерисовки его в новых координатах мы воспользуемся функцией setInterval.

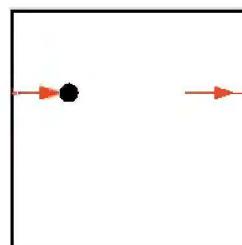


Рис. 15.1. Если мяч вылетит за правую границу «холста», он опять появится слева

Actions —  
действия



## Настройка «холста»

Сначала нам нужно найти элемент `canvas` и получить контекст рисования. Откройте файл `keyboard.html` и замените весь JavaScript внутри второй пары тегов `<script>` на следующий код:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var width = canvas.width;
var height = canvas.height;
```

В первой строке мы с помощью `document.getElementById` находим элемент `canvas`. Во второй строке вызываем метод `getContext`, чтобы получить контекст рисования. Затем, в следующих двух строках, мы сохраняем ширину и высоту «холста» в переменных `width` и `height`. Таким образом, когда в программе нам понадобятся эти размеры, можно будет использовать переменные, а не вводить каждый раз числовые значения. А если нам захочется поменять размер «холста», достаточно будет отредактировать HTML, и JavaScript-код будет по-прежнему работать.

## Функция `circle`

Мы будем использовать все ту же функцию `circle`, с помощью которой рисовали мяч в главе 14. Введите этот код после предыдущего кода:

```
var circle = function (x, y, radius, fillCircle) {
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    if (fillCircle) {
        ctx.fill();
    } else {
        ctx.stroke();
    }
};
```

## Создаем конструктор `Ball`

Теперь добавим конструктор `Ball`, чтобы создавать с его помощью объект-мяч. Мы применим тот же способ перемещения мяча, что и в главе 14: свойства `xSpeed` и `ySpeed` будут управлять горизонтальной и вертикальной скоростью движения. Добавьте этот код после функции `circle`:

---

```
var Ball = function () {
    this.x = width / 2;
    this.y = height / 2;
    this.xSpeed = 5;
    this.ySpeed = 0;
};
```

---

Здесь мы задали свойствам `x` и `y` (координатам мяча) значения `width / 2` и `height / 2`, чтобы мяч появился в центре «холста». Также мы установили `this.xSpeed` в 5, а `this.ySpeed` в 0 — это значит, что сразу после запуска программы мяч будет двигаться вправо (на каждом шаге анимации его `x`-координата будет увеличиваться на 5 пикселей, а `y`-координата останется неизменной).

## Создаем метод move

В этом разделе мы создадим и добавим к `Ball.prototype` метод `move`, перемещающий мяч в новую позицию в зависимости от его текущей позиции, а также значений `xSpeed` и `ySpeed`. Добавьте код этого метода после конструктора `Ball`:

---

```
Ball.prototype.move = function () {
    this.x += this.xSpeed;
    this.y += this.ySpeed;

❶    if (this.x < 0) {
        this.x = width;
    } else if (this.x > width) {
        this.x = 0;
    }
    if (this.y < 0) {
        this.y = height;
    } else if (this.y > height) {
        this.y = 0;
    }
};
```

---



Сначала мы обновляем `this.x` и `this.y` на основе значений `this.xSpeed` и `this.ySpeed` — так же, как мы делали это в главе 14 (см. «Перемещение мяча» на с. 218). Затем следует код, обрабатывающий выход мяча за границы «холста».

Конструкция `if...else` в строке ❶ проверяет, не вышел ли мяч за границу «холста». Если так и есть, этот код перенесет мяч на противоположную сторону «холста». Например, если мяч вышел за левую границу, он должен снова появиться с правой стороны. Иными словами, если `this.x` меньше 0, мы присваиваем `this.x` значение `width`, в результате чего

мяч оказывается у правой границы. Последующие строки конструкции if... else обрабатывают остальные три границы таким же образом.



### Создаем метод draw

Рисовать мяч мы будем с помощью метода draw. Добавьте этот код после метода move:

```
Ball.prototype.draw = function () {  
    circle(this.x, this.y, 10, true);  
};
```

Этот метод вызывает функцию `circle`, передавая ей свойства `x` и `y` как координаты центра, 10 в качестве радиуса и `true` в качестве аргумента `fillCircle`. На рис. 15.2 показан мяч, который получится в результате.

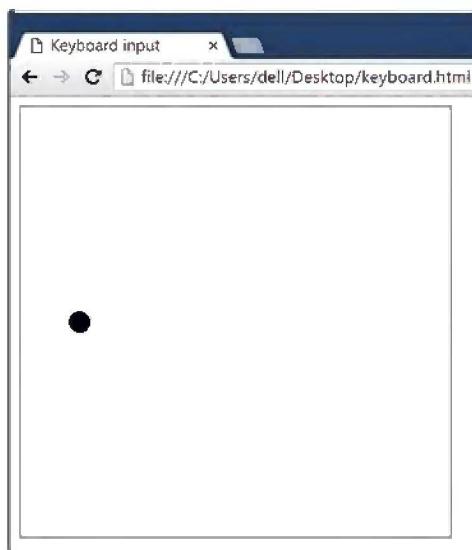


Рис. 15.2. Мяч как заполненная окружность с радиусом 10

### Создаем метод setDirection

**Set direction —**  
задать  
направление

Теперь нужно выяснить, как задавать направление движения мяча. Мы будем делать это в методе `setDirection`, который будет вызываться из обработчика события `keydown` (речь о котором пойдет ниже). Обработчик передаст методу `setDirection` информацию о нажатой клавише в виде строки ("left", "up", "right", "down" или "stop").

На основе этого `setDirection` будет менять свойства мяча `xSpeed` и `ySpeed` — таким образом, чтобы направление полета мяча соответствовало направлению, выбранному нажатием клавиши. Например, если методу передана строка "down", мы сбросим `this.xSpeed` в 0, а `this.ySpeed` присвоим значение 5. Добавьте этот код после метода `draw`:

---

```
Ball.prototype.setDirection = function (direction) {
  if (direction === "up") {
    this.xSpeed = 0;
    this.ySpeed = -5;
  } else if (direction === "down") {
    this.xSpeed = 0;
    this.ySpeed = 5;
  } else if (direction === "left") {
    this.xSpeed = -5;
    this.ySpeed = 0;
  } else if (direction === "right") {
    this.xSpeed = 5;
    this.ySpeed = 0;
  } else if (direction === "stop") {
    this.xSpeed = 0;
    this.ySpeed = 0;
  }
};
```

---

Тело этого метода представляет собой одну большую конструкцию `if... else`. Новое направление передается в аргументе `direction` — если это "up", мы сбросим свойство `xSpeed` в 0, а `ySpeed` присвоим значение `-5`. Остальные направления обрабатываются аналогично. Наконец, если в метод передана строка "stop", мы сбросим в 0 оба свойства `xSpeed` и `ySpeed`, что приведет к остановке мяча.

## Реакция на нажатия клавиш

Следующий фрагмент кода создает объект-мяч с помощью конструктора `Ball` и отслеживает события `keydown`, чтобы менять направление мяча в соответствии с нажатыми клавишами. Добавьте этот код после метода `setDirection`:

---

```
① var ball = new Ball();
② var keyActions = {
  32: "stop",
  37: "left",
  38: "up",
  39: "right",
```

```
    40: "down"
};

❸ $("body").keydown(function (event) {
❹   var direction = keyActions[event.keyCode];
❺   ball.setDirection(direction);
});
```

---

В строке ❶ мы создаем объект мяча вызовом new Ball(). В строке ❷ создаем объект keyActions для преобразования кодов клавиш в соответствующие им направления. Этот объект практически ничем не отличается от объекта keyNames со с. 226, лишь коду 32 (клавиша «пробел») соответствует обозначение "stop", а не "space", поскольку мы хотим, чтобы нажатие на пробел останавливало движение мяча.

В строке ❸ мы с помощью jQuery-функции \$ находим элемент body и вызываем его метод keydown, чтобы обрабатывать события клавиатуры. Функция, переданная в этот метод, будет вызываться каждый раз при нажатии клавиши на клавиатуре.

В теле этой функции, в строке ❹, мы используем выражение keyActions[event.keyCode] для поиска названия нажатой клавиши и присваиваем это значение переменной direction. Таким образом, в direction оказывается направление: "left", если нажата стрелка влево, "right" для стрелки вправо, "up" для стрелки вверх, "down" для стрелки вниз и "stop" для пробела. Если была нажата какая-то другая клавиша, direction примет значение undefined и на анимацию это никак не повлияет.

И наконец, в строке ❺ мы вызываем метод объекта-мяча setDirection, передавая в него строку с направлением. Как уже было сказано, setDirection обновляет свойства xSpeed и ySpeed в соответствии с новым направлением.

## Анимация мяча

Нам осталось лишь анимировать мяч. Следующий код наверняка покажется вам знакомым, поскольку примерно то же самое мы делали в главе 14. Аналогично предыдущим анимациям, для периодического обновления позиции мяча здесь используется функция setInterval. Добавьте эти строки после кода из предыдущего раздела:

---

```
setInterval(function () {
  ctx.clearRect(0, 0, width, height);
  ball.draw();
  ball.move();
  ctx.strokeRect(0, 0, width, height);
}, 30);
```

---

Мы используем `setInterval`, чтобы вызывать функцию анимации раз в 30 миллисекунд. Функция сначала очищает весь «холст» с помощью `clearRect`, а затем вызывает методы объекта-мяча `draw` и `move`. Как мы уже знаем, метод `draw` просто рисует окружность в текущей позиции мяча, а метод `move` обновляет позицию мяча в соответствии со значениями его свойств `xSpeed` и `ySpeed`. И наконец, наша функция рисует с помощью `strokeRect` рамку, делая границы «холста» видимыми.

## Код программы

Мы уже рассмотрели все части кода, но для вашего удобства вот вся программа целиком.

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var width = canvas.width;
var height = canvas.height;
var circle = function (x, y, radius, fillCircle) {
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    if (fillCircle) {
        ctx.fill();
    } else {
        ctx.stroke();
    }
};

// Конструктор Ball
var Ball = function () {
    this.x = width / 2;
    this.y = height / 2;
    this.xSpeed = 5;
    this.ySpeed = 0;
};

// Обновляем позицию мяча соответственно его скорости
Ball.prototype.move = function () {
    this.x += this.xSpeed;
    this.y += this.ySpeed;
    if (this.x < 0) {
        this.x = width;
    } else if (this.x > width) {
        this.x = 0;
    } else if (this.y < 0) {
        this.y = height;
    } else if (this.y > height) {
        this.y = 0;
    }
};
```



```

// Рисуем мяч в его текущей позиции
Ball.prototype.draw = function () {
    circle(this.x, this.y, 10, true);
};

// Задаем направление движения по строке с названием действия
Ball.prototype.setDirection = function (direction) {
    if (direction === "up") {
        this.xSpeed = 0;
        this.ySpeed = -5;
    } else if (direction === "down") {
        this.xSpeed = 0;
        this.ySpeed = 5;
    } else if (direction === "left") {
        this.xSpeed = -5;
        this.ySpeed = 0;
    } else if (direction === "right") {
        this.xSpeed = 5;
        this.ySpeed = 0;
    } else if (direction === "stop") {
        this.xSpeed = 0;
        this.ySpeed = 0;
    }
};

// Создаем объект-мяч
var ball = new Ball();

// Объект для перевода кодов клавиш в названия действий
var keyActions = {
    32: "stop", // остановка
    37: "left", // влево
    38: "up", // вверх
    39: "right", // вправо
    40: "down" // вниз
};

// Обработчик события keydown, будет вызван при каждом нажатии
// клавиши
$("body").keydown(function (event) {
    var direction = keyActions[event.keyCode];
    ball.setDirection(direction);
});

// Функция анимации, вызывается раз в 30 мс
setInterval(function () {
    ctx.clearRect(0, 0, width, height);
    ball.draw();
    ball.move();
    ctx.strokeRect(0, 0, width, height);
}, 30);

```

---

## Запуск программы

Наша программа завершена. При запуске вы увидите черный мяч, движущийся слева направо по «холсту», как на рис. 15.3. Дойдя до правого края, мяч должен появиться с левой стороны, продолжая движение вправо. При нажатии клавиш-стрелок мяч должен менять направление, а при нажатии на «пробел» остановиться.

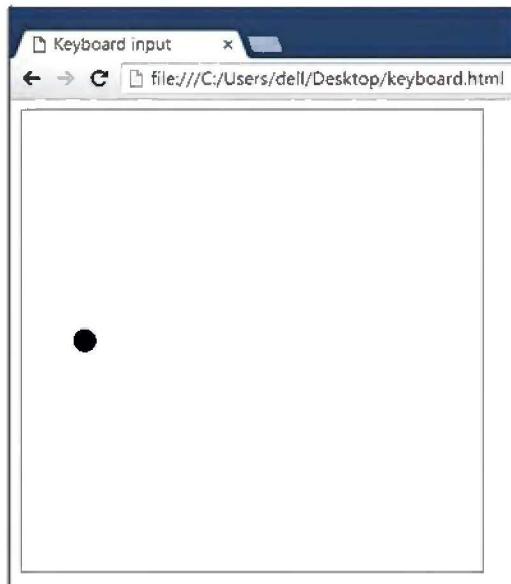


Рис. 15.3. Скриншот анимации мяча

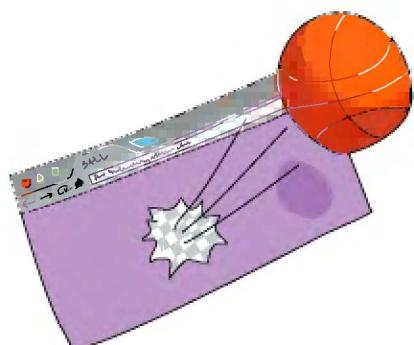


Если анимация не реагирует на нажатия клавиш, кликните по странице, чтобы убедиться, что у программы есть доступ к клавиатуре.

## Что мы узнали

В этой главе мы научились писать программы, реагирующие на нажатия клавиш. Мы создали анимацию мяча, направление движения которого можно менять с клавиатуры.

Теперь мы знаем, как рисовать на «холсте», создавать анимации и управлять этими анимациями с клавиатуры, а значит, уже готовы написать простую игру, использующую элемент `canvas!` В следующих главах мы создадим вариант классической видеоигры «Змейка», пустив в ход все, что научились делать.



## УПРАЖНЕНИЯ

Вот несколько способов сделать последнюю анимацию сложнее и интереснее.

### #1. Отскоки от границ «холста»

Измените код так, чтобы мяч отскакивал от границ при столкновении, а не появлялся с другой стороны.

Подсказка: при столкновении просто измените направление на противоположное.

### #2. Управление скоростью

Сейчас на каждом шаге анимации мяч перемещается на 5 пикселей, поскольку `setDirection` всегда задает свойствам `xSpeed` и `ySpeed` значения `-5` или `5`. Создайте в конструкторе `Ball` новое свойство `speed`, хранящее скорость мяча, и используйте его в методе `setDirection` вместо цифры `5`.

Теперь добавьте в код возможность задавать скорость (`speed`) нажатием цифровых клавиш от `1` до `9`.

Подсказка: создайте объект `speeds` с разными значениями скоростей и в обработчике события `keydown` выбирайте из него соответствующую клавишу скорость.

### #3. Гибкое управление

Измените код так, чтобы при нажатии клавиши `Z` мяч замедлял движение, а при нажатии `X` разгонялся. Затем сделайте так, чтобы клавиша `C` уменьшала размер мяча, а `V` — увеличивала.

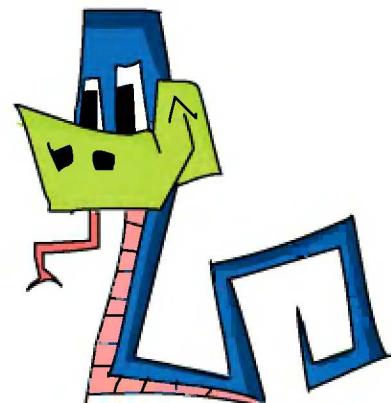
Что произойдет, если скорость станет отрицательной? А размер? Добавьте в код проверку, гарантирующую, что скорость и размер никогда не станут меньше `0`.

# 16

## ПИШЕМ ИГРУ «ЗМЕЙКА»: ЧАСТЬ 1

В этой и следующей главах мы разработаем свою версию классической игры «Змейка». В «Змейке» игрок управляет с клавиатуры змеей, направляя ее вверх, вниз, влево или вправо. По мере движения змейки по экрану на нем появляются яблоки, которые змейка поедает, становясь при этом длиннее. Однако если змейка столкнется с препятствием или съест свой хвост, игра закончится.

При создании этой игры нам понадобится множество техник и инструментов, изученных в предыдущих главах, включая jQчегу, рисование на «холсте», анимации и интерактивность. В этой главе мы рассмотрим общую структуру игры и напишем код для рисования рамки, вывода игрового счета и окончания игры. В следующей мы напишем код для самой змейки и яблока, а затем соберем части воедино и сможем поиграть!



### Игровой процесс

На рис. 16.1 показано, как будет выглядеть законченная игра. В процессе игры нужно следить за четырьмя объектами и выводить их на экран: это рамка (серого цвета), счет игры (надпись черным), сама змейка (синего цвета) и яблоко (светло-зеленое).

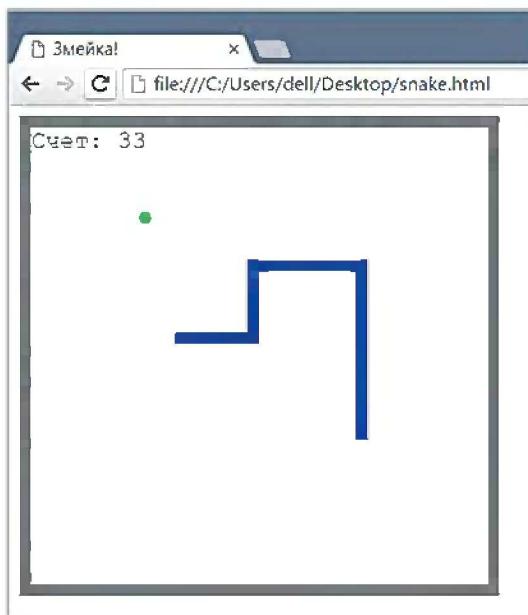


Рис. 16.1. Наша «Змейка»

## Структура игры

Перед тем как начинать писать код, давайте рассмотрим общую структуру игры. Этот псевдокод описывает, что наша программа должна делать:

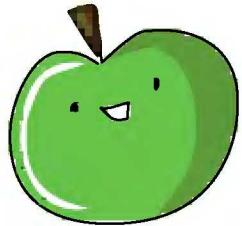
---

```
Настроить «холст»
Установить счет игры в 0
Создать змейку
Создать яблоко
Каждые 100 миллисекунд {
    Очистить «холст»
    Напечатать текущий счет игры
    Сдвинуть змейку в текущем направлении
    Если змейка столкнулась со стеной или своим хвостом {
        Закончить игру
    } Иначе Если змейка съела яблоко {
        Увеличить счет на 1
        Переместить яблоко на новое место
        Увеличить длину змейки
    }
    Для каждого сегмента тела змейки {
        Нарисовать сегмент
    }
    Нарисовать яблоко
```

```
    Нарисовать рамку
}
Когда игрок нажмет клавишу {
    Если это клавиша-стрелка {
        Обновить направление движения змейки
    }
}
```

---

Постепенно мы напишем код для каждого из этих шагов. Но сначала давайте поговорим о некоторых основных частях нашей программы и выясним, какие из инструментов JavaScript нам понадобятся для их реализации.



## Использование setInterval для анимации в игре

Как видно в псевдокоде, каждые 100 миллисекунд нам нужно вызывать набор функций и методов для обновления и перерисовки всех объектов на игровом поле. Так же как в главах 14 и 15, мы будем использовать `setInterval` для периодического вызова всех этих функций. Вот как будет выглядеть вызов `setInterval` в законченной программе:

```
var intervalId = setInterval(function () {
    ctx.clearRect(0, 0, width, height);
    drawScore();
    snake.move();
    snake.draw();
    apple.draw();
    drawBorder();
}, 100);
```

---

**Draw score** —  
нарисовать счет  
**Snake** —  
змейка  
**Draw border** —  
нарисовать границу

В функции, которую мы передаем первым аргументом `setInterval`, первая строка кода очищает «холст» вызовом `clearRect`, после чего можно рисовать следующий кадр анимации. Далее следуют вызовы нескольких функций и методов. Обратите внимание, что они приблизительно соответствуют командам псевдокода.

Также отметим, что ID интервала, который возвращает функция `setInterval`, сохраняется в переменной `intervalId`. Это значение понадобится нам, чтобы остановить анимацию, когда игра закончится (см. «Конец игры» на с. 249).

## Создание игровых объектов

В этой игре мы воспользуемся изученным в главе 12 объектно-ориентированным стилем программирования для реализации двух основных игровых объектов: змейки и яблока. Мы создадим конструктор

**Apple** —  
яблоко

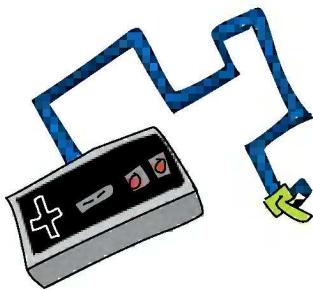
**Block** — секция

для каждого из этих объектов (назовем их *Snake* и *Apple*) и добавим методы (такие как *move* и *draw*) к прототипам этих конструкторов.

Кроме того, мы будем рассматривать игровое поле как сетку с квадратными ячейками и напишем конструктор под названием *Block* для создания объектов, представляющих ячейки в этой сетке. Эти объекты мы будем использовать как позиции сегментов тела змейки, плюс еще один объект-ячейка будет хранить текущую позицию яблока. Также у объектов-ячеек будут методы, позволяющие отображать сегменты змейки и яблока на «холсте».

## Управление с клавиатуры

В псевдокоде есть фрагмент, посвященный обработке нажатия клавиш. Для управления змейкой с помощью клавиш-стрелок мы используем возможности jQuery по обработке событий клавиатуры, как мы это уже делали в предыдущей главе. Мы будем определять, какая клавиша нажата, проверяя ее код, и в соответствии с этим менять направление движения змейки.



## Начинаем писать игру

Мы вкратце рассмотрели устройство будущей программы, теперь пора браться за код! В этой главе мы подготовим HTML-файл, «холст» и некоторые переменные, которые нам понадобятся. Затем займемся наиболее простыми функциями, необходимыми в игре: функцией рисования рамки вокруг игрового поля, функцией отображения текущего счета и функцией окончания игры. В следующей главе мы создадим конструкторы и методы для змейки и яблока, обработчик нажатий на клавиши-стрелки, а потом соединим все вместе.

## Создаем HTML-файл

Начнем с HTML-файла — введите следующий код в текстовом редакторе и сохраните его под именем *snake.html*.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Змейка!</title>
</head>
<body>

  ❶ <canvas id="canvas" width="400" height="400"></canvas>
```

```
❷ <script src="https://code.jquery.com/jquery-2.1.0.js"></script>

❸ <script>
// Здесь будет JavaScript-код
</script>
</body>
</html>
```

---

В строке ❶ мы создали элемент canvas размером  $400 \times 400$  пикселей. Здесь будут отображаться все объекты нашей игры. В строке ❷ загружаем библиотеку jQuery, а в строке ❸ добавляем еще пару тегов `<script>`, внутри которых будет располагаться JavaScript-код игры. Его-то мы и начнем писать.

## Определяем переменные canvas, ctx, width и height

Для начала определим переменные `canvas` и `ctx`, необходимые для рисования на «холсте», а также переменные `width` и `height` для хранения ширины и высоты элемента `canvas`.

---

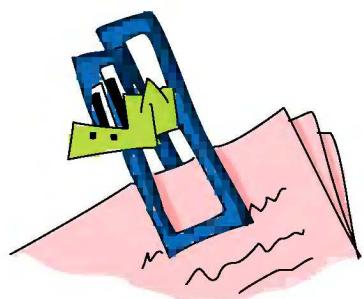
```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var width = canvas.width;
var height = canvas.height;
```

---

В HTML-коде заданы ширина и высота по 400 пикселей. Если вы измените эти размеры в HTML, переменные `width` и `height` также примут соответствующие значения.

## Делим «холст» на ячейки

Теперь создадим переменные, которые помогут нам представить «холст» в виде сетки из ячеек  $10 \times 10$  пикселей, как на рис. 16.2. Сетка будет невидимой (то есть вы не увидите ее на экране в процессе игры), однако все игровые объекты будут выровнены по ней.



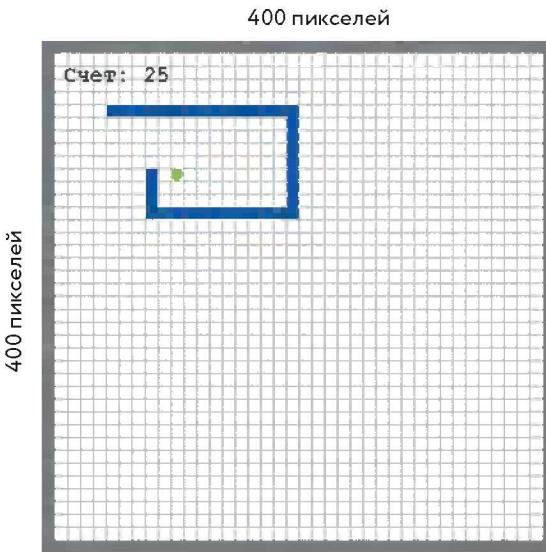


Рис. 16.2. 10-пиксельная сетка, разбивающая игровой экран на ячейки

И змейка, и яблоко будут в одну ячейку шириной, чтобы вписаться в эту сетку. На каждом шаге анимации змейка будет передвигаться в текущем направлении ровно на одну ячейку.

Для создания ячеек мы воспользуемся такими переменными:

**Block size** —  
размер секции

**Width  
in blocks** —  
ширина  
в ячейках

**Height  
in blocks** —  
высота  
в ячейках

---

```

❶ var blockSize = 10;
❷ var widthInBlocks = width / blockSize;
    var heightInBlocks = height / blockSize;
```

---

В строке ❶ мы создали переменную с именем `blockSize`, задав ей значение 10, поскольку мы хотим, чтобы наши ячейки были по 10 пикселей в ширину и в высоту. В строке ❷ мы создали переменные `widthInBlocks` и `heightInBlocks`. Значение `widthInBlocks` мы вычислили как ширину «холста», деленную на размер ячейки, получив таким образом ширину «холста» в ячейках. Аналогично `heightInBlocks` хранит высоту «холста» в ячейках. Сейчас и ширина, и высота «холста» равны 400 пикселям, поэтому обе переменные `widthInBlocks` и `heightInBlocks` примут значение 40. Если вы подсчитаете количество клеток на рис. 16.2 (включая рамку), получится как раз по 40 клеток в ширину и высоту.

## Переменная score

Наконец, создадим переменную score.

```
var score = 0;
```

Мы будем использовать эту переменную для подсчета набранных игроком очков. Поскольку это самое начало программы, мы присвоим ей значение 0. Каждый раз, когда змейка проглотит яблоко, мы будем увеличивать значение score на 1.



## Рисуем рамку

Далее напишем функцию drawBorder для рисования рамки вокруг «холста». Рамка будет шириной ровно в одну ячейку.

Наша функция будет рисовать четыре длинных и тонких прямоугольника, по одному для каждой стороны рамки. Толщина каждого прямоугольника равняется blockSize (10 пикселей), а длина — полной ширине или высоте «холста».

```
var drawBorder = function () {
    ctx.fillStyle = "Gray";
    ❶ ctx.fillRect(0, 0, width, blockSize);
    ❷ ctx.fillRect(0, height - blockSize, width, blockSize);
    ❸ ctx.fillRect(0, 0, blockSize, height);
    ❹ ctx.fillRect(width - blockSize, 0, blockSize, height);
};
```

Сначала мы задаем для fillStyle значение "Gray" (серый), поскольку хотим, чтобы рамка была серой. Затем в строке ❶ рисуем верхнюю часть рамки — прямоугольник в позиции (0, 0), то есть в верхнем левом углу «холста», шириной width (400 пикселей) и высотой blockSize (10 пикселей).

Затем в строке ❷ рисуем нижнюю часть рамки — прямоугольник в позиции (0, height - blockSize) или (0, 390). Это точка на 10 пикселей выше нижней границы «холста» и прямо у его левой границы. Так же как и у верхней части рамки, ширина этого прямоугольника равна width, а высота — blockSize.

На рис. 16.3 показаны верхняя и нижняя части рамки.

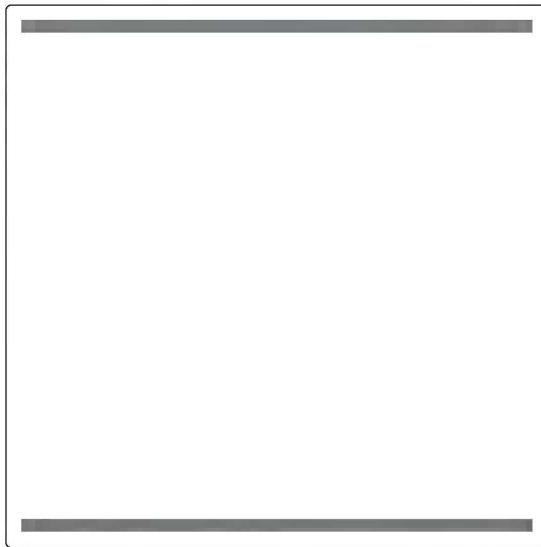


Рис. 16.3. Верхняя и нижняя части рамки

В строке **③** мы рисуем левую часть рамки, а в строке **④** — правую. На рис. 16.4 показана рамка с добавлением этих двух частей.

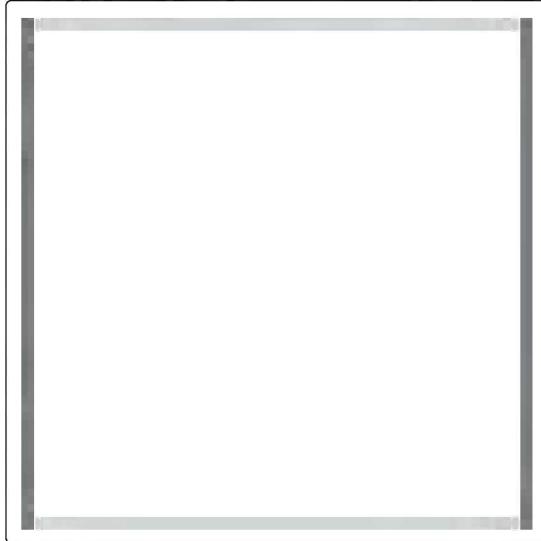


Рис. 16.4. Левая и правая части рамки  
(верхняя и нижняя части показаны светло-серым цветом)

## Отображение счета

Теперь давайте напишем функцию `drawScore` для отображения текущего счета игры в верхнем левом углу «холста», как показано на рис. 16.1 на с. 238. Для отображения текста эта функция будет использовать метод контекста рисования `fillText`, который принимает строку, а также *x* и *y*-координаты позиции, в которой эту строку нужно отобразить. Например,

```
ctx.fillText("Привет, мир!", 50, 50);
```

напечатает на «холсте» строку «Привет, мир!» в позиции (50, 50). На рис. 16.5 показано, как это будет выглядеть.



Рис. 16.5. Стока «Привет, мир!», напечатанная в позиции (50, 50)

Взгляните, мы напечатали на «холсте» строку текста! Но что если мы хотим поменять внешний вид этого текста, выбрать другой размер, шрифт или выравнивание? Для игрового счета в нашей «Змейке» больше подойдет другой шрифт, размер букв стоит увеличить, а разместить надпись нужно в верхнем левом углу, прямо под рамкой. Поэтому прежде чем написать нашу функцию `drawScore`, давайте получше изучим метод `fillText` и выясним, как можно управлять внешним видом текста, отображаемого на «холсте».

**Baseline** —  
базовая,  
опорная  
линия

**Bottom** —  
ниж

**Top** —  
верх

**Middle** —  
середина

## Настройка опорной линии текста

Позиция, определяющая, где отображается текст, называется *опорной линией*. По умолчанию по опорной линии выравнивается нижний левый угол строки текста, в результате чего текст отображается над заданной позицией и правее нее.

Чтобы изменить расположение текста относительно опорной линии, можно поменять значение свойства `textBaseline`. По умолчанию это свойство имеет значение "bottom", но также вы можете задать `textBaseline` значения "top" или "middle". На рис. 16.6 показано, как в каждом из этих случаев текст выравнивается относительно позиции печати, переданной функции `fillText` (показана красной точкой).

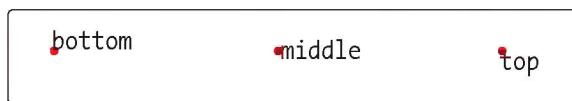


Рис. 16.6. Отображение текста при разных значениях `textBaseline`

Например, чтобы текст отображался под опорной линией, введите:

```
ctx.textBaseline = "top";
ctx.fillText("Привет, мир!", 50, 50);
```

Теперь при вызове `fillText` текст отобразится под точкой (50, 50), как на рис. 16.7.

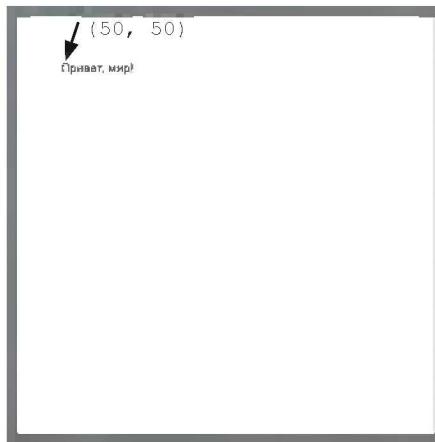


Рис. 16.7. Стока «Привет, мир!» для `textBaseline = "top"`

Аналогичным образом, чтобы выбрать горизонтальное положение текста относительно позиции печати, можно задавать свойству

`textAlign` значения "left", "center" или "right". На рис. 16.8 показано, что при этом получится.

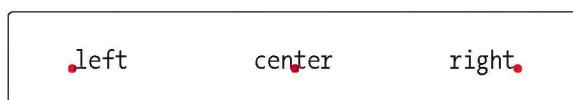


Рис. 16.8. Отображение текста при разных значениях `textAlign`

**Text align** — выравнивание текста

**Center** — центр (амер.)

## Задаем размер и шрифт

Мы можем настраивать размер текста и шрифт, которым он печатается, меняя значение свойства контекста рисования `font`. Вот примеры использования некоторых шрифтов:

```
❶ ctx.font = "20px Courier";
ctx.fillText("Courier", 50, 50);

ctx.font = "24px Comic Sans MS";
ctx.fillText("Comic Sans", 50, 100);

ctx.font = "18px Arial";
ctx.fillText("Arial", 50, 150);
```

**Font** — шрифт

Свойство `font` принимает строку, в которой следует указать размер и название нужного шрифта. Например, в строке ❶ мы задали свойству `font` значение "20px Courier", то есть текст будет отображен 20-пиксельным шрифтом Courier. На рис. 16.9 показано, как эти три шрифта выглядят на «холсте».

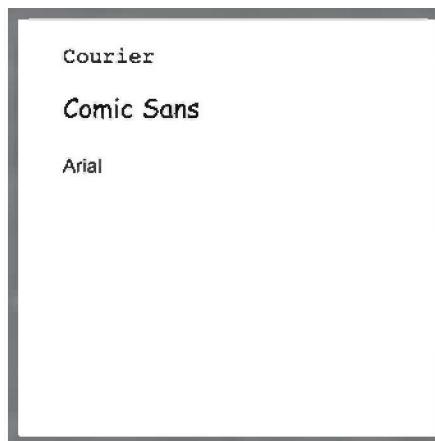


Рис. 16.9. 20-пиксельный Courier, 24-пиксельный Comic Sans  
и 18-пиксельный Arial

## Пишем функцию drawScore

Теперь можно продолжать — напишем функцию drawScore, которая отображает на «холсте» строку с текущим счетом игры.

```
var drawScore = function () {
    ctx.font = "20px Courier";
    ctx.fillStyle = "Black";
    ctx.textAlign = "left";
    ctx.textBaseline = "top";
    ctx.fillText("Счет: " + score, blockSize, blockSize);
};
```

Эта функция выставляет 20-пиксельный шрифт Courier ("20px Courier"), с помощью fillStyle задает черный цвет печати, включает выравнивание по левому краю с помощью свойства textAlign и задает свойству textBaseline значение "top".

Затем мы вызываем fillText со строкой "Счет: " + score. В переменной score хранится текущий счет игры в виде числа. Начальный счет мы установили в 0 (см. «Переменная score» на с. 243), поэтому сначала эта функция напечатает строку "Счет: 0".

Вызывая fillText, мы передаем ему в качестве x- и у-координат значение blockSize. Мы задали для blockSize значение 10, поэтому текст будет отображен в позиции (10, 10), то есть внутри верхнего левого угла рамки. А поскольку мы задали свойству textBaseline значение "top", текст будет расположен под опорной линией, как на рис. 16.10.

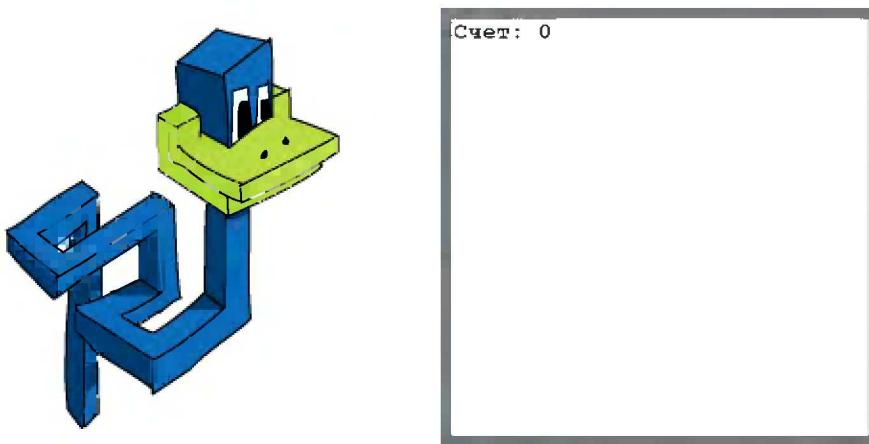


Рис. 16.10. Позиция строки со счетом игры

## Конец игры

Мы будем вызывать функцию `gameOver` в конце игры, когда змейка врезается в стену (рамку) или в собственный хвост. Эта функция использует `clearInterval` для остановки игры и отображает на «холсте» надпись «Конец игры». Вот как выглядит ее код:

```
var gameOver = function () {
    clearInterval(intervalId);
    ctx.font = "60px Courier";
    ctx.fillStyle = "Black";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("Конец игры", width / 2, height / 2);
};
```

Game over —  
игра окончена

Сначала мы останавливаем игру вызовом `clearInterval` с переменной `intervalId` в качестве аргумента. Это отменит периодический запуск функции анимации (см. «Использование `setInterval` для анимации в игре» на с. 239).

Затем мы устанавливаем 60-пиксельный шрифт `Courier` черного цвета, выравнивание текста по центру и задаем для `textBaseline` значение `"middle"` (середина). После этого вызываем метод `fillText`, передавая ему строку «Конец игры», значение `width / 2` в качестве координаты `x` и `height / 2` в качестве координаты `y`. В результате надпись «Конец игры» появится в центре «холста», как на рис. 16.11.

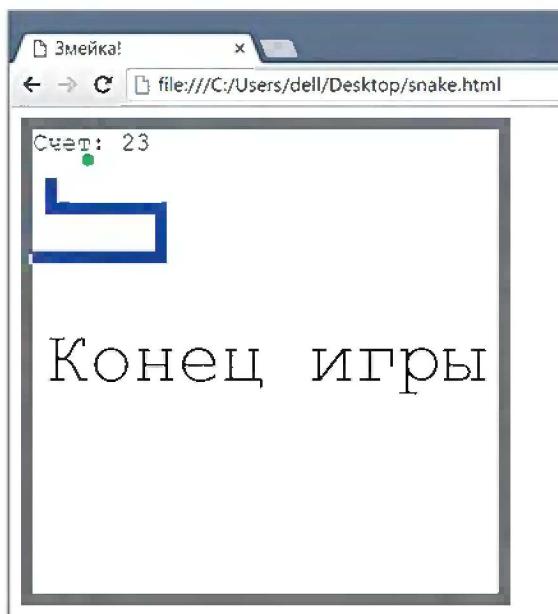


Рис. 16.11. Экран «Конец игры» после того, как змейка врезалась в левую стенку

## Что мы узнали

В этой главе мы рассмотрели общую структуру игры «Змейка» и написали несколько нужных для нее функций, научились печатать на «холсте» текст и настраивать его размер, шрифт и выравнивание.

В следующей главе мы завершим игру, написав код для змейки и яблока, а также обработчик событий клавиатуры.



## УПРАЖНЕНИЯ

Вот несколько упражнений, которые вы можете выполнить перед тем, как переходить к написанию самой игры.

### #1. Соберите код воедино

Хоть программа еще не завершена, вы можете проверить работу функций рисования рамки и вывода игрового счета. Возьмите HTML-файл (см. «Создаем HTML-файл» на с. 240) и добавьте в него код для настройки «холста», создания переменной `score`, рисования рамки и отображения счета. Теперь, чтобы увидеть рамку и счет, вам нужно лишь вызвать функции `drawBorder` и `drawScore`. Результат должен выглядеть так же, как на рис. 16.10. Также можете попробовать вызвать функцию `gameOver`, однако перед этим вам придется удалить из нее строку с командой `clearInterval(intervalId)` — в программе еще нет переменной `intervalId`, поэтому вызов функции без удаления этой строки приведет к ошибке.

### #2. Анимация счета

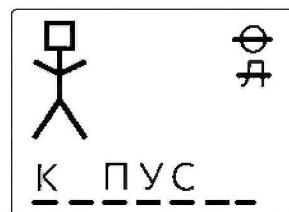
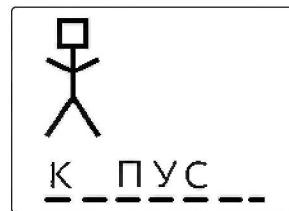
Добавьте собственный вызов `setInterval` с функцией, которая увеличивает переменную `score` на 1 и затем выводит новый счет на экран с помощью `drawScore` каждые 100 миллисекунд. Не забывайте, что каждый раз перед выводом счета нужно очищать «холст» вызовом метода `clearRect`.

### #3. Добавьте вывод текста к «Виселице»

Вспомните упражнение #4 из главы 13 — рисование на «холсте» человечка для игры «Виселица». Усовершенствуйте игру, используя метод `fillText` для вывода текущего слова под человечком, как показано на рисунке.

Подсказка: для подчеркивания каждой буквы я использовал линии длиной 30 пикселей, с расстоянием 10 пикселей между ними.

Можете усложнить задачу — отображайте также неправильные ответы в виде зачеркнутых букв, как на рисунке справа.



# 17

## ПИШЕМ ИГРУ «ЗМЕЙКА»: ЧАСТЬ 2

В этой главе мы закончим писать «Змейку». Мы уже подготовили поле и в общих чертах разобрались, как будет работать игра. Теперь пришел черед создать объекты для змейки и яблока, а также запрограммировать обработчик событий клавиатуры, чтобы игрок мог управлять змейкой с помощью клавиш-стрелок. А в конце главы мы увидим полный код программы.

При создании кода для змейки и яблока мы воспользуемся объектно-ориентированным подходом, изученным в главе 12, — у каждого объекта будет набор методов и конструктор. Также объекты для змейки и яблока будут использовать функционал более простого объекта, представляющего собой ячейку в сетке игрового поля. Начнем с написания конструктора для простого объекта-ячейки.

### Создаем конструктор `Block`

В этом разделе мы напишем конструктор `Block`, который будет создавать объекты, представляющие собой отдельные ячейки невидимой сетки игрового поля. У каждого объекта-ячейки будут свойства `col` и `row`, определяющие позицию ячейки. На рис. 17.1 показана сетка игрового поля, а также нумерация строк и столбцов. И хотя сетка не отображается на экране, наша игра спроектирована таким образом, чтобы яблоко и сегменты тела змейки всегда были выровнены по ней.

На рис. 17.1 ячейка с яблоком находится в столбце 10 и строке 10. А голова змейки (слева от яблока) — в столбце 8 и строке 10.

**Col** —  
column —  
столбец  
**Row** —  
строка

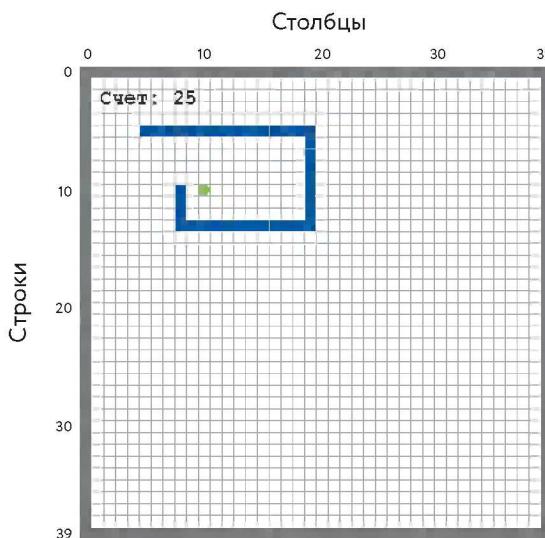


Рис. 17.1. Номера столбцов и строк, используемые в конструкторе Block

А вот код конструктора Block:

---

```
var Block = function (col, row) {
    this.col = col;
    this.row = row;
};
```

---

Значения строки и столбца передаются в конструктор Block как аргументы и сохраняются в свойствах col и row нового объекта.

Теперь можно воспользоваться нашим конструктором и создать объект, соответствующий конкретной ячейке игрового поля. Например, вот как будет выглядеть создание объекта для ячейки, находящейся в столбце 5 и строке 5:

---

```
var sampleBlock = new Block(5, 5);
```

---

Sample block —  
шаблон  
ячейки

## Добавляем метод drawSquare

Наш объект-ячейка позволяет задать позицию в сетке, но, чтобы в данной позиции что-то появилось, это что-то нужно нарисовать на «холсте». Следующим этапом мы добавим два метода, `drawSquare` и `drawCircle`, предназначенные для рисования в заданной ячейке квадрата и окружности. Начнем с `drawSquare`:

```
Block.prototype.drawSquare = function (color) {  
    ① var x = this.col * blockSize;  
    ② var y = this.row * blockSize;  
    ctx.fillStyle = color;  
    ctx.fillRect(x, y, blockSize, blockSize);  
};
```



Из главы 12 мы знаем, что, если добавить метод к свойству конструктора `prototype`, этот метод будет доступен всем объектам, созданным при помощи данного конструктора. Поэтому, добавляя метод `drawSquare` к `Block.prototype`, мы делаем его доступным для всех объектов-ячеек.

Этот метод рисует квадрат в позиции, заданной свойствами `col` и `row` (столбец и строка), и принимает единственный аргумент `color`, определяющий цвет. Чтобы нарисовать на «холсте» квадрат, нужно задать `x`- и `y`-координаты его верхнего левого угла, поэтому в строках ① и ② мы вычисляем эти значения `x` и `y` для нашей ячейки, умножая ее свойства `col` и `row` на `blockSize`. Затем мы присваиваем свойству контекста рисования `fillStyle` значение цвета, переданное в аргументе `color`.

И наконец, мы вызываем метод `ctx.fillRect`, передавая ему вычисленные значения `x` и `y`, а также `blockSize` в качестве ширины и высоты прямоугольника.

Так можно создать объект-ячейку в столбце 3, строке 4 и нарисовать там квадрат:

```
var sampleBlock = new Block(3, 4);  
sampleBlock.drawSquare("LightBlue");
```

На рис. 17.2 показан этот квадрат на «холсте», с пометками, показывающими, как вычисляются его позиция и размер.

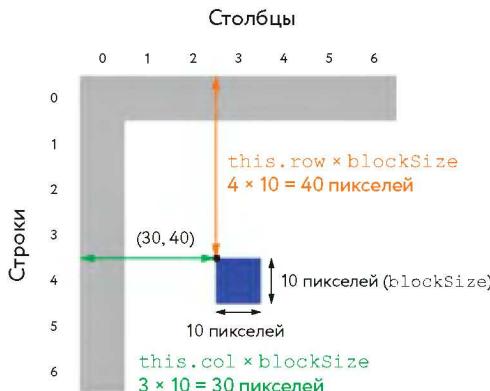


Рис. 17.2. Расчет значений для рисования квадрата

## Добавляем метод drawCircle

Настал черед метода `drawCircle`. Он очень похож на `drawSquare`, однако вместо квадрата рисует заполненную окружность.

```
Block.prototype.drawCircle = function (color) {
    var centerX = this.col * blockSize + blockSize / 2;
    var centerY = this.row * blockSize + blockSize / 2;
    ctx.fillStyle = color;
    circle(centerX, centerY, blockSize / 2, true);
};
```

Сначала мы вычисляем *x*- и *y*-координаты центра окружности, создав для этого переменные `centerX` и `centerY`. Как и прежде, мы умножаем свойства `col` и `row` на `blockSize`, но в этот раз еще и прибавляем `blockSize / 2`, поскольку нам нужны координаты центра окружности, который находится в середине ячейки (как показано на рис. 17.3).

Мы задаем свойству контекста `fillStyle` значение аргумента `color`, аналогично тому, как делали это для `drawSquare`, а затем вызываем уже знакомую нам функцию `circle`, передавая ей `centerX` и `centerY` в качестве координат, `blockSize / 2` в качестве радиуса и `true`, чтобы окружность была заполнена цветом. Это все та же функция `circle` из главы 14, поэтому нужно, как и прежде, добавить ее определение в код (вы увидите его в полном тексте программы).

Нарисовать окружность в столбце 4, строке 3 можно так:

```
var sampleCircle = new Block(4, 3);
sampleCircle.drawCircle("LightGreen");
```

На рис. 17.3 показана наша окружность с вычислениями ее радиуса и координат центра.

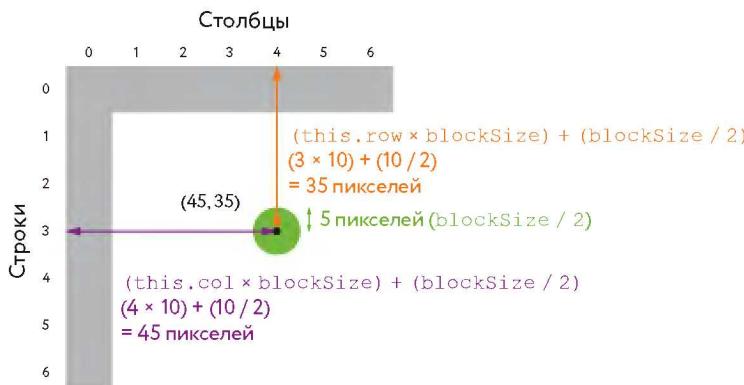


Рис. 17.3. Расчет значений для рисования окружности



**Equal** —  
равняется

## Добавляем метод equal

В коде игры нужно будет проверять, не находятся ли два объекта-ячейки в одной и той же позиции. Например, если позиции яблока и головы змейки совпадут, это будет означать, что змея съела яблоко. С другой стороны, если совпадут позиции головы змейки и сегмента ее тела, это будет означать, что змейка врезалась в собственный хвост.

Чтобы сравнивать позиции ячеек было легче, мы создадим метод `equal` и добавим его к свойству `prototype` конструктора `Block`. Вызов `equal` для некоего объекта-ячейки с другим объектом в качестве аргумента вернет `true`, если позиции двух объектов-ячеек совпадают (и `false` в противном случае). Вот код метода:

**Other block** —  
другая ячейка

```
Block.prototype.equal = function (otherBlock) {  
    return this.col === otherBlock.col && this.row === otherBlock.row;  
};
```

Метод `equal` весьма прост: если свойства `col` и `row` двух ячеек (`this` и `otherBlock`) совпадают (то есть `this.col` равняется `otherBlock.col` и `this.row` равняется `otherBlock.row`), значит ячейки находятся в одной позиции и метод вернет `true`.

Например, создадим два объекта-ячейки под названиями `apple` (яблоко) и `head` (голова) и проверим, совпадают ли их позиции:

```
var apple = new Block(2, 5);  
var head = new Block(3, 5);  
head.equal(apple);  
false
```

Свойства `row` у объектов `apple` и `head` совпадают (равны 5), но свойства `col` различаются. Однако если мы присвоим переменной `head` новый объект, расположенный на 1 столбец левее прежнего, метод сообщит нам, что оба объекта находятся в одной позиции:

```
head = new Block(2, 5);  
head.equal(apple);  
true
```

Обратите внимание: не имеет значения, напишем мы `head.equal(apple)` или `apple.equal(head)`, в обоих случаях будет выполнено одно и то же сравнение.

Мы воспользуемся методом `equal` позже, чтобы проверить, не съела ли змейка яблоко и не столкнулась ли она с собственным хвостом.

## Создаем змейку

Теперь займемся змейкой. Мы будем хранить ее положение в массиве под названием `segments`, состоящем из объектов-ячеек. Чтобы передвинуть змейку, мы будем добавлять новую ячейку в начало массива `segments` и убирать ячейку с конца. Первый элемент массива будет соответствовать змеиной голове.

**Segment** —  
часть, отрезок

## Пишем конструктор `Snake`

Первым делом нам нужен конструктор для создания объекта-змейки:

```
var Snake = function () {
  ❶  this.segments = [
    new Block(7, 5),
    new Block(6, 5),
    new Block(5, 5)
  ];
  ❷  this.direction = "right";
  ❸  this.nextDirection = "right";
};
```

Сегменты тела змейки

Свойство `segments` в строке ❶ — это массив объектов-ячеек, каждый из которых соответствует сегменту змеиного тела. В самом начале игры этот массив содержит три ячейки в позициях (7, 5), (6, 5) и (5, 5). На рис. 17.4 изображены эти три начальных сегмента.

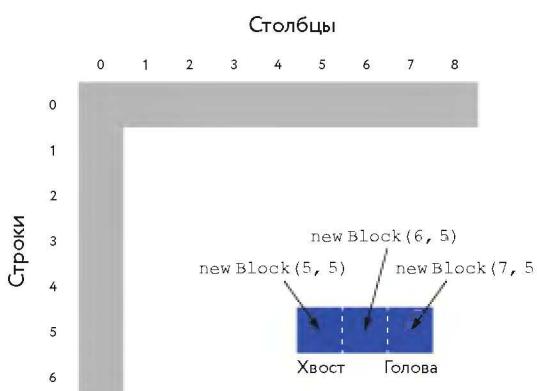
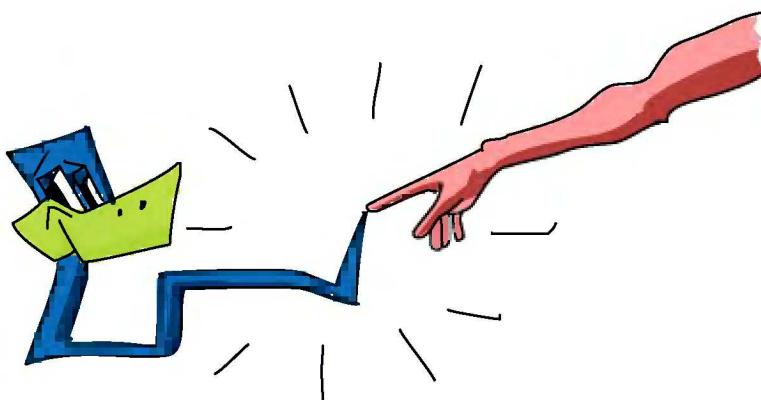


Рис. 17.4. Ячейки, из которых состоит змейка в начале игры



### Задаем направление движения

Next —  
следующий

Свойство `direction` в строке ② хранит текущее направление движения змейки. Также в строке ③ наш конструктор добавляет объекту свойство `nextDirection`, где хранится направление, в котором змейка будет двигаться на следующем шаге анимации. Это свойство будет изменяться в обработчике событий клавиатуры при нажатии на одну из клавиш-стрелок (см. «Обработчик события `keydown`» на с. 264). Пока что конструктор присваивает обоим свойствам значение `"right"`, чтобы в начале игры змейка двигалась вправо.

### Рисуем змейку

Чтобы нарисовать змейку, нам нужно перебрать в цикле все объекты-ячейки из массива `segments`, вызывая методы `drawSquare`. Таким образом, для каждого сегмента змейки в соответствующей ячейке будет нарисован квадрат.

---

```
Snake.prototype.draw = function () {
    for (var i = 0; i < this.segments.length; i++) {
        this.segments[i].drawSquare("Blue");
    }
};
```

---

Метод `draw` использует цикл `for` для обращения к каждому из объектов-ячеек в массиве `segments`. При каждом повторе цикла этот код выбирает текущий сегмент (`this.segments[i]`) и вызывает его метод `drawSquare("Blue")`, в результате чего в соответствующей ячейке игрового поля появится синий квадрат.

Чтобы проверить этот метод, вы можете запустить следующий код, создающий новый объект с помощью конструктора `Snake` и вызывающий его метод `draw`:

```
var snake = new Snake();
snake.draw();
```

## Перемещаем змейку

Для перемещения змейки на одну ячейку в текущем направлении мы создадим метод `move`. Чтобы змейка передвинулась, мы добавим ей новый сегмент головы (то есть добавим новый объект-ячейку в начало массива `segments`), а затем удалим из массива `segments` последний элемент — сегмент хвоста.

Метод `move` также будет вызывать метод под названием `checkCollision`, проверяющий, не столкнулась ли новая голова со змейным хвостом или со стеной (рамкой), а также не съела ли новая голова яблоко. Если голова столкнулась со стеной или хвостом, мы завершим игру вызовом функции `gameOver`, созданной в предыдущей главе. Если же змейка съела яблоко, мы увеличим счет игры и переместим яблоко в новое место.



**Collision** —  
столкновение

## Добавляем метод move

Код метода `move` выглядит так:

```
Snake.prototype.move = function () {
  ①  var head = this.segments[0];
  ②  var newHead;

  ③  this.direction = this.nextDirection;

  ④  if (this.direction === "right") {
      newHead = new Block(head.col + 1, head.row);
    } else if (this.direction === "down") {
      newHead = new Block(head.col, head.row + 1);
    } else if (this.direction === "left") {
      newHead = new Block(head.col - 1, head.row);
    } else if (this.direction === "up") {
      newHead = new Block(head.col, head.row - 1);
    }

  ⑤  if (this.checkCollision(newHead)) {
      gameOver();
      return;
    }
}
```

```
⑥    this.segments.unshift(newHead);

⑦    if (newHead.equal(apple.position)) {
        score++;
        apple.move();
    } else {
        this.segments.pop();
    }
};
```

---

Давайте разберем этот метод строчка за строчкой.

#### Создание новой головы

В строке ① мы сохраняем первый элемент массива `this.segments` (это голова змейки) в переменной `head`. Мы будем неоднократно обращаться к первому сегменту в коде этого метода, поэтому использование такой переменной сделает код короче и чуть проще для понимания. Теперь вместо того, чтобы снова и снова писать `this.segments[0]`, нам достаточно ввести `head`.

В строке ② мы создали переменную `newHead`, чтобы использовать ее для хранения объекта-ячейки, представляющего собой новую змеиную голову (которую мы как раз собираемся создать).

В строке ③ мы присваиваем `this.direction` значение `this.nextDirection` — после этого направление движения змейки будет соответствовать последней нажатой клавише-стрелке (мы в подробностях разберем, как это работает, когда займемся обработчиком событий клавиатуры).

#### DIRECTION И NEXTDIRECTION

Свойство объекта-змейки `direction` будет обновляться один раз за один шаг анимации, поскольку метод `move` вызывается один раз для каждого шага. С другой стороны, свойство `nextDirection` будет менять значение в любой момент, когда игрок нажмет одну из клавиш-стрелок (и если он будет жать на клавиши очень быстро, теоретически это свойство может поменяться несколько раз за один шаг анимации). Используя эти два свойства таким способом, мы можем не бояться, что змейка мгновенно развернется в противоположном направлении и уткнется сама в себя, если игрок очень быстро нажмет две кнопки в промежутке между двумя кадрами анимации.



Начиная со строки ④ мы используем цепочку конструкций `if... else` для обработки разных направлений. В каждом случае мы создаем новую змейную голову и сохраняем ее в переменной `newHead`. В зависимости от направления мы добавляем или вычитаем единицу из значения строки или столбца нынешней головы, чтобы поместить новую голову вплотную рядом с ней (справа, слева, снизу или сверху, в зависимости от направления движения). Например, на рис. 17.5 показано, где окажется новая голова, если в `this.nextDirection` будет направление "down" (вниз).

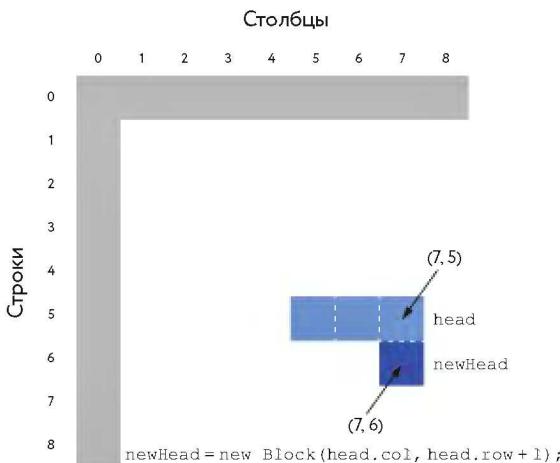


Рис. 17.5. Создание новой головы,  
если в `this.nextDirection` направление "down"

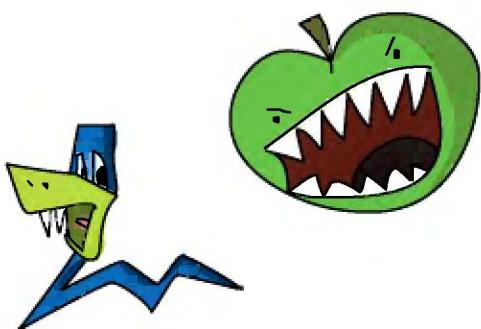
#### Проверка столкновения и добавление головы

В строке ⑤ мы вызываем метод `checkCollision`, чтобы выяснить, не врезалась ли змейка в свой хвост или в стену. Скоро вы увидите код этого метода, но вы, наверное, уже догадались, что он возвращает `true`, если змейка на что-то наткнулась. Если это произойдет, в теле `if` будет вызвана функция `gameOver` и игра завершится, а на экране появится надпись «Конец игры».

Команда `return` в следующей за вызовом `gameOver` строке приведет к раннему возврату из метода, так что остальной код будет пропущен. Но `return` сработает, только если `checkCollision` перед этим вернет `true`, поэтому, если змейка ни с чем не столкнется, будет выполнен остальной код метода.

До тех пор, пока змейка не столкнется с препятствием, в строке ⑥ мы будем добавлять ей новую голову, с помощью `unshift` добавляя

`newHead` в начало массива `segments`. Чтобы вспомнить, как работает `unshift`, загляните в раздел «Добавление элементов в массив» на с. 56.



### Съедение яблока

В строке ⑦ мы используем метод `equal`, сравнивая `newHead` и `apple.position`. Если позиция этих объектов-ячеек одинакова, метод `equal` вернет `true` и это будет означать, что змейка съела яблоко.

Если яблоко съедено, мы увеличиваем счет игры на 1 и затем вызываем для объекта `apple` метод `move`, перемещая яблоко в новую позицию. В противном случае (если змейка не съела яблоко) мы вызываем `pop` для массива `this.segments`.

Эта команда удаляет сегмент змейки, общая же длина змейки остается прежней (поскольку ранее мы добавили новый сегмент головы). Таким образом, когда змейка съедает яблоко, она становится длиннее на один сегмент, поскольку мы добавляем ей головной сегмент, но не убираем сегмент хвоста.

Мы еще не писали код для яблока, поэтому данный метод работать пока не будет. Однако если вам хочется его протестировать, замените конструкцию `if... else` начиная со строки ⑦ на следующую строку:

---

```
this.segments.pop();
```

---

И тогда нам останется только добавить метод `checkCollision`, чем мы сейчас и займемся.

### Добавляем метод `checkCollision`

Каждый раз, когда мы задаем новую позицию змейки головы, нужно выполнять проверку на столкновения. Такие проверки характерны для многих компьютерных игр, и нередко они представляют одну из наиболее сложных при создании игры задач. К счастью, в нашей «Змейке» этот вопрос решается весьма просто.

Нас волнуют только два типа столкновений: столкновения со стенами и столкновения с собственным телом. Змейка может столкнуться сама с собой, если развернуть ее голову так, чтобы она уперлась в тело. В начале игры змейка слишком коротка для этого, однако после поедания нескольких яблок такое столкновение становится вполне вероятным.

Вот код метода checkCollision:

```
Snake.prototype.checkCollision = function (head) {  
    ① var leftCollision = (head.col === 0);  
    var topCollision = (head.row === 0);  
    var rightCollision = (head.col === widthInBlocks - 1);  
    var bottomCollision = (head.row === heightInBlocks - 1);  
  
    ② var wallCollision = leftCollision || topCollision || ←  
        rightCollision || bottomCollision;  
  
    ③ var selfCollision = false;  
  
    ④ for (var i = 0; i < this.segments.length; i++) {  
        ⑤ if (head.equal(this.segments[i])) {  
            selfCollision = true;  
        }  
    }  
  
    ⑥ return wallCollision || selfCollision;  
};
```

Left — слева  
Top — вверху  
Right — справа  
Bottom — внизу  
Wall — стена  
Self collision —  
столкновение  
с собой

Проверка столкновений со стенами

В строке ① мы создаем переменную leftCollision и присваиваем ей значение выражения head.col === 0. Таким образом, эта переменная примет значение true, если змейка столкнется с левой стенкой, иначе говоря, если голова окажется в столбце 0. Аналогично переменная topCollision используется для проверки строки, в которой находится змеиная голова, чтобы выяснить, не столкнулась ли она с верхней стенкой.



Далее мы проверяем столкновение с правой стенкой, сравнивая столбец, где находится голова со значением widthInBlocks - 1. Поскольку widthInBlocks равняется 40, по сути, это проверка, не находится ли голова в столбце 39, который занят правой стенкой, как показано на рис. 17.1. Затем мы выполняем похожую проверку для нижней стенки (переменная bottomCollision), проверяя, не равно ли свойство головы row значению heightInBlocks - 1.

В строке ② мы определяем, столкнулась ли змейка с какой-нибудь из стенок: с помощью операции || (ИЛИ) мы проверяем, не соответствует ли переменная leftCollision, или topCollision, или rightCollision, или bottomCollision значению true. Полученный булев результат мы сохраняем в переменной wallCollision.

## Проверка столкновения с собственным телом

Чтобы проверить, не столкнулась ли змейка с собственным телом, в строке ❸ мы создаем переменную `selfCollision`, сначала присвоив ей значение `false`. Затем в строке ❹ мы используем цикл `for` для перебора всех сегментов змейки и проверки, не находится ли какой-нибудь из сегментов в той же позиции, что и новая голова змейки, — для этого служит выражение `head.equal(this.segments[i])`. И голова, и другие сегменты тела — это объекты-ячейки, поэтому мы можем использовать для проверки совпадения позиций метод объектов-ячеек `equal`. Если обнаружится, что какой-либо из сегментов тела находится там же, где голова, значит змейка столкнулась сама с собой, и тогда мы присваиваем `selfCollision` значение `true` ❺.

И наконец, в строке ❻ мы возвращаем из метода `wallCollision || selfCollision` — это выражение даст `true`, если змейка столкнулась либо со стенкой, либо сама с собой.

## Управляем змейкой с клавиатуры

Теперь мы напишем код, позволяющий игроку задавать направление движения змейки с клавиатуры. Мы добавим в программу обработчик событий клавиатуры, который будет определять нажатия клавиш-стрелок и менять направление движения в соответствии с нажатой клавишей.

## Обработчик события `keydown`

Этот код обрабатывает события клавиатуры:

---

```
❶ var directions = {  
 37: "left",  
 38: "up",  
 39: "right",  
 40: "down"  
};  
  
❷ $("body").keydown(function (event) {  
 ❸   var newDirection = directions[event.keyCode];  
   if (newDirection !== undefined) {  
     snake.setDirection(newDirection);  
   }  
});
```

---

В строке ❶ мы создаем объект для преобразования кодов клавиш-стрелок в строки, обозначающие направления движения (этот объект аналогичен объекту `keyActions`, который мы использовали

в разделе «Реакция на нажатия клавиш» на с. 231). В строке ❷ мы задаем обработчик для событий keydown внутри элемента body. Этот обработчик будет вызываться всякий раз, когда пользователь нажмет клавишу (если сначала он кликнет мышкой по странице браузера).

Первым делом этот обработчик преобразовывает полученный из объекта-события код клавиши в строку с названием направления и сохраняет эту строку в переменной newDirection. Если же полученный код клавиши не равен 37, 38, 39 или 40 (это коды клавиш-стрелок), выражение directions[event.keyCode] вернет undefined.

В строке ❸ мы сравниваем переменную newDirection с undefined и, если она не равна undefined, вызываем метод объекта-змейки setDirection, передавая ему строку newDirection. (Поскольку в этой конструкции if нет ветви else, в случае если newDirection равняется undefined, мы просто проигнорируем нажатие клавиши.)

Сейчас этот код не будет работать, поскольку мы еще не определили метод объекта-змейки setDirection. Давайте это исправим.

### Создаем метод setDirection

Метод setDirection принимает от обработчика клавиатуры, код которого мы только что разбирали, строку с направлением и использует ее, чтобы установить новое направление движения змейки. Кроме того, этот метод не дает игроку менять направление такими способами, после которых змейка сразу же врезается сама в себя. Например, если змейка, двинувшись вправо, тут же начнет двигаться влево, не повернув перед этим вверх или вниз, чтобы освободить себе путь, она мгновенно врезается сама в себя. Мы будем называть такие смены направления недопустимыми, поскольку мы не хотим давать игроку возможность их совершать. Например, на рис. 17.6 для движения змейки вправо показано два допустимых направления и одно недопустимое.

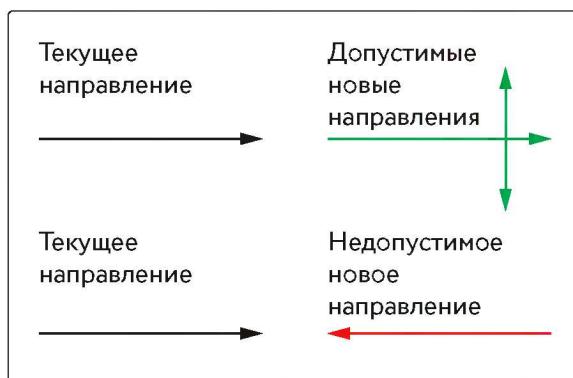


Рис. 17.6. Допустимые направления относительно текущего

Метод `setDirection` проверяет, не пытается ли игрок выбрать недопустимое направление. Когда это так, метод использует команду `return` для досрочного выхода. В противном случае он обновляет свойство объекта-змейки `nextDirection`.

Вот код метода `setDirection`:

```
Snake.prototype.setDirection = function (newDirection) {
 ❶  if (this.direction === "up" && newDirection === "down") {
    return;
  } else if (this.direction === "right" && newDirection === "left") {
    return;
  } else if (this.direction === "down" && newDirection === "up") {
    return;
  } else if (this.direction === "left" && newDirection === "right") {
    return;
  }

 ❷  this.nextDirection = newDirection;
};
```

Конструкция `if... else` в строке ❶ состоит из четырех частей — для обработки четырех недопустимых смен направления, которые мы хотим предотвратить. В первой части говорится, что, если змейка движется вверх (`this.direction` равняется `"up"`), а игрок нажал стрелку вниз (`newDirection` равняется `"down"`), мы должны совершить досрочный выход из метода с помощью `return`. Остальные части конструкции обрабатывают другие недопустимые смены направления тем же образом.

Последняя строка метода `setDirection` будет выполнена, только если в `newDirection` находится допустимое новое направление — иначе до нее дело не дойдет, потому что одна из команд `return` в предыдущих строках завершит выполнение метода.

Если направление `newDirection` является допустимым, мы присваиваем его свойству `nextDirection` в строке ❷.

## Создаем яблоко

В этой игре яблоко представляет собой объект из трех компонентов: это свойство `position`, которое хранит позицию яблока в виде объекта-ячейки, метод `draw`, с помощью которого мы будем рисовать яблоко на экране, и метод `move`, который нужен, чтобы задать яблоку новую позицию после того, как змейка его съела.

## Пишем конструктор `Apple`

Все, что делает конструктор, — это задает свойство `position`, присваивая ему новый объект-ячейку.

---

```
var Apple = function () {
    this.position = new Block(10, 10);
};
```

---

Этот код создает новый объект-ячейку в столбце 10, строке 10 и присваивает его свойству объекта-яблока `position`. Мы будем использовать этот конструктор для создания объекта-яблока в самом начале игры.

## Рисуем яблоко

Чтобы нарисовать яблоко, создадим метод `draw`:

---

```
Apple.prototype.draw = function () {
    this.position.drawCircle("LimeGreen");
};
```

---

Этот метод очень прост, всю работу за него выполняет метод `drawCircle` (созданный в разделе «Добавляем метод `drawCircle`» на с. 255). Чтобы нарисовать яблоко, мы просто вызываем для хранящего объект-ячейку свойства `position` метод `drawCircle`, передавая ему название цвета "LimeGreen" (темно-зеленый) для отображения в заданной ячейке зеленого кружка.

Чтобы проверить работу этого метода, выполните такой код:

---

```
var apple = new Apple();
apple.draw();
```

---

## Перемещаем яблоко

Метод `move` перемещает яблоко в случайную новую позицию на игровом поле (то есть в любую ячейку на «холсте», кроме области рамки). Мы будем вызывать этот метод всякий раз, когда змейка съест яблоко, чтобы оно появилось снова в другой позиции.

---

```
Apple.prototype.move = function () {
    ① var randomCol = Math.floor(Math.random() * (widthInBlocks - 2)) + 1;
    ② var randomRow = Math.floor(Math.random() * (heightInBlocks - 2)) + 1;
    ③ this.position = new Block(randomCol, randomRow);
};
```

---

**Random col** —  
случайный  
столбец

**Random row** —  
случайная  
строка

В строке ① и следующей строке мы создаем переменные `randomCol` и `randomRow`, которые примут значения, соответствующие случайному столбцу и случайной строке на игровом поле. Как показано на рис. 17.1 на с. 253, столбцы и строки игрового поля имеют номера от 1 до 38, следовательно, нам нужно выбрать случайные числа из этого диапазона.

Для получения этих чисел мы воспользуемся выражением `Math.floor(Math.random() * 38)`, которое возвращает случайное число от 0 до 37, и затем прибавим к нему 1, чтобы получить число от 1 до 38 (как работают методы `Math.floor` и `Math.random`, см. в разделе «Случайный выбор» на с. 65).

Именно это мы и делаем в строке ❶ для получения случайного столбца, однако вместо 38 мы пишем `(widthInBlocks - 2)`. Это нужно для того, чтобы у нас была возможность изменить размер игрового поля без необходимости редактировать код программы. И таким же образом мы получаем случайное значение строки с помощью `Math.floor(Math.random() * (heightInBlocks - 2)) + 1`.

Наконец, в строке ❷ мы создаем новый объект-ячейку, позиция которого соответствует нашим случайным значениям для столбца и строки, сохраняя его в `this.position`. Таким образом, позиция яблока изменится на новую случайную позицию в пределах игрового поля.

Вы можете проверить работу метода `move` таким образом:

---

```
var apple = new Apple();
apple.move();
apple.draw();
```

---

## Код игры

Наша игра состоит примерно из 200 строк JavaScript-кода! Если соединить все рассмотренные ранее фрагменты воедино, программа будет выглядеть так:

---

```
// Настройка «холста»
❶ var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");

// Получаем ширину и высоту элемента canvas
var width = canvas.width;
var height = canvas.height;

// Вычисляем ширину и высоту в ячейках
var blockSize = 10;
var widthInBlocks = width / blockSize;
var heightInBlocks = height / blockSize;

// Устанавливаем счет 0
var score = 0;

// Рисуем рамку
❷ var drawBorder = function () {
    ctx.fillStyle = "Gray";
```

---

```

ctx.fillRect(0, 0, width, blockSize);
ctx.fillRect(0, height - blockSize, width, blockSize);
ctx.fillRect(0, 0, blockSize, height);
ctx.fillRect(width - blockSize, 0, blockSize, height);
};

// Выводим счет игры в левом верхнем углу
var drawScore = function () {
    ctx.font = "20px Courier";
    ctx.fillStyle = "Black";
    ctx.textAlign = "left";
    ctx.textBaseline = "top";
    ctx.fillText("Счет: " + score, blockSize, blockSize);
};

// Отменяем действие setInterval и печатаем сообщение «Конец игры»
var gameOver = function () {
    clearInterval(intervalId);
    ctx.font = "60px Courier";
    ctx.fillStyle = "Black";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("Конец игры", width / 2, height / 2);
};

// Рисуем окружность (используя функцию из главы 14)
var circle = function (x, y, radius, fillCircle) {
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    if (fillCircle) {
        ctx.fill();
    } else {
        ctx.stroke();
    }
};

// Задаем конструктор Block (ячейка)
③ var Block = function (col, row) {
    this.col = col;
    this.row = row;
};

// Рисуем квадрат в позиции ячейки
Block.prototype.drawSquare = function (color) {
    var x = this.col * blockSize;
    var y = this.row * blockSize;
    ctx.fillStyle = color;
    ctx.fillRect(x, y, blockSize, blockSize);
};

// Рисуем круг в позиции ячейки
Block.prototype.drawCircle = function (color) {
    var centerX = this.col * blockSize + blockSize / 2;

```

```

    var centerY = this.row * blockSize + blockSize / 2;
    ctx.fillStyle = color;
    circle(centerX, centerY, blockSize / 2, true);
};

// Проверяем, находится ли эта ячейка в той же позиции, что и ячейка
// otherBlock
Block.prototype.equal = function (otherBlock) {
    return this.col === otherBlock.col && this.row === otherBlock.row;
};

// Задаем конструктор Snake (змейка)
❸ var Snake = function () {
    this.segments = [
        new Block(7, 5),
        new Block(6, 5),
        new Block(5, 5)
    ];

    this.direction = "right";
    this.nextDirection = "right";
};

// Рисуем квадратик для каждого сегмента тела змейки
Snake.prototype.draw = function () {
    for (var i = 0; i < this.segments.length; i++) {
        this.segments[i].drawSquare("Blue");
    }
};

// Создаем новую голову и добавляем ее к началу змейки,
// чтобы передвинуть змейку в текущем направлении
Snake.prototype.move = function () {
    var head = this.segments[0];
    var newHead;

    this.direction = this.nextDirection;

    if (this.direction === "right") {
        newHead = new Block(head.col + 1, head.row);
    } else if (this.direction === "down") {
        newHead = new Block(head.col, head.row + 1);
    } else if (this.direction === "left") {
        newHead = new Block(head.col - 1, head.row);
    } else if (this.direction === "up") {
        newHead = new Block(head.col, head.row - 1);
    }

    if (this.checkCollision(newHead)) {
        gameOver();
        return;
    }
}

```

```

        this.segments.unshift(newHead);
        if (newHead.equal(apple.position)) {
            score++;
            apple.move();
        } else {
            this.segments.pop();
        }
    };

    // Проверяем, не столкнулась ли змейка со стеной или собственным
    // телом
    Snake.prototype.checkCollision = function (head) {
        var leftCollision = (head.col === 0);
        var topCollision = (head.row === 0);
        var rightCollision = (head.col === widthInBlocks - 1);
        var bottomCollision = (head.row === heightInBlocks - 1);

        var wallCollision = leftCollision || topCollision || ←
            rightCollision || bottomCollision;

        var selfCollision = false;

        for (var i = 0; i < this.segments.length; i++) {
            if (head.equal(this.segments[i])) {
                selfCollision = true;
            }
        }

        return wallCollision || selfCollision;
    };

    // Задаем следующее направление движения змейки на основе нажатой
    // клавиши
    Snake.prototype.setDirection = function (newDirection) {
        if (this.direction === "up" && newDirection === "down") {
            return;
        } else if (this.direction === "right" && newDirection === "left") {
            return;
        } else if (this.direction === "down" && newDirection === "up") {
            return;
        } else if (this.direction === "left" && newDirection === "right") {
            return;
        }

        this.nextDirection = newDirection;
    };

    // Задаем конструктор Apple (яблоко)
    ⑤ var Apple = function () {
        this.position = new Block(10, 10);
    };

```

```

// Рисуем кружок в позиции яблока
Apple.prototype.draw = function () {
    this.position.drawCircle("LimeGreen");
};

// Перемещаем яблоко в случайную позицию
Apple.prototype.move = function () {
    var randomCol = Math.floor(Math.random() * (widthInBlocks - 2)) + 1;
    var randomRow = Math.floor(Math.random() * (heightInBlocks - 2)) + 1;
    this.position = new Block(randomCol, randomRow);
};

// Создаем объект-змейку и объект-яблоко
❶ var snake = new Snake();
var apple = new Apple();

// Запускаем функцию анимации через setInterval
var intervalId = setInterval(function () {
    ctx.clearRect(0, 0, width, height);
    drawScore();
    snake.move();
    snake.draw();
    apple.draw();
    drawBorder();
}, 100);

// Преобразуем коды клавиш в направления
❷ var directions = {
    37: "left",
    38: "up",
    39: "right",
    40: "down"
};

// Задаем обработчик события keydown (клавиши-стрелки)
$("body").keydown(function (event) {
    var newDirection = directions[event.keyCode];
    if (newDirection !== undefined) {
        snake.setDirection(newDirection);
    }
});

```

---

Этот код состоит из нескольких частей. Первая часть начинается со строки ❶, в ней создаются и настраиваются необходимые в коде игры переменные, включая «холст», контекст рисования, ширину и высоту «холста» (об этом мы говорили в главе 16). Далее, со строки ❷, идет код отдельных функций: `drawBorder`, `drawScore`, `gameOver` и `circle`.

Со строки ❸ начинается код конструктора `Block`, за которым следуют его методы `drawSquare`, `drawCircle` и `equal`. Затем, со строки ❹, идет код конструктора `Snake` и все его методы. Далее, со строки ❺, следует конструктор `Apple` и его методы `draw` и `move`.

Наконец, со строки ⑥ начинается код, который запускает игру и обеспечивает ее выполнение. Сначала мы создаем объект-змейку `snake` и объект-яблоко `apple`. Затем с помощью `setInterval` мы запускаем анимацию игры. Обратите внимание, что при вызове `setInterval` мы сохранили ID интервала в переменной `intervalId`, чтобы иметь возможность отменить его выполнение в функции `gameOver`.

Функция, переданная `setInterval` в качестве аргумента, вызывается на каждом шаге игры. Она отвечает за отображение на «холсте» всей графики и за обновление состояния игры. Она очищает «холст» и затем рисует змейку, яблоко и рамку. Также она вызывает метод объекта-змейки `move`, передвигающий змейку на один шаг в текущем направлении. После вызова `setInterval` со строки ⑦ идет код для отслеживания событий клавиатуры и установки направления движения змейки.

Как и прежде, вам следует поместить весь этот код внутрь элемента `script` в HTML-документе. Чтобы запустить игру, просто загрузите файл `snake.html` в браузер и управляйте змейкой с помощью клавиш-стрелок. Если при нажатии на клавиши ничего не происходит, может понадобиться кликнуть мышкой внутри окна браузера, чтобы он мог отслеживать события клавиатуры.

Если игра не работает, возможно, в ваш JavaScript-код закралась ошибка. Сведения об ошибках отображаются в консоли — посмотрите, нет ли там каких-нибудь сообщений. В случае, если выяснить, в чем проблема, так и не получилось, внимательно, строка за строкой сверьте ваш код с кодом, приведенным выше.

Ну, как вам игра? Сколько очков сможете набрать?



## Что мы узнали

В этой главе мы создали законченную игру на основе элемента `canvas`. В нашей игре используются различные типы данных, концепции и техники, изученные в процессе чтения книги: это числа, строки, булевые значения, массивы, объекты, управляющие конструкции, функции, объектно-ориентированное программирование, обработчики событий, `setInterval` и рисование на «холсте».

Теперь, когда вы разобрались с игрой «Змейка», вы можете запрограммировать на JavaScript множество других несложных двухмерных видеоигр. Например, вы можете создать свои версии классических Breakout, Asteroids, Space Invaders или Tetris. Или придумать собственную игру!

Разумеется, JavaScript годится не только для игр. Например, зная, как выполнять на языке JavaScript математические расчеты, вы можете использовать его для решения задач по математике. А может, вы хотите создать сайт, чтобы показать всему миру, на что вы способны? Вариантов бесчисленное множество!

## УПРАЖНЕНИЯ

Вот некоторые идеи по усовершенствованию и доработке нашей игры.

### #1. Увеличьте игровое поле

Измените размер игрового поля до квадрата  $500 \times 500$  пикселей. Что в коде нужно поменять, чтобы игра работала с таким размером поля?

### #2. Цветная змейка

Наша змейка выглядит скучновато: все сегменты ее тела одинакового синего цвета. Можно сделать ее больше похожей на настоящую змею, если чередовать цвета сегментов, сделав тело полосатым. Например, пусть голова будет зелено-желтой, а дальше чередуйте через один сегмент синий и желтый цвета либо выберите другой окрас на свой вкус.

### #3. Постепенно увеличивайте скорость игры

Измените программу так, чтобы с каждым съеденным яблоком скорость игры возрастала. Для этого вместо `setInterval` используйте `setTimeout`, ведь `setInterval` вызывает заданную функцию всегда через один и тот же промежуток времени. Однако вы можете снова и снова вызывать функцию через `setTimeout`, меняя значение задержки при каждом вызове:

**Animation time** — время анимации  
**Game loop** — цикл игры

---

```
var animationTime = 100;
var gameLoop = function () {
    // Здесь должен быть код для рисования объектов и обновления
    // состояния игры
    setTimeout(gameLoop, animationTime);
};
gameLoop();
```

---

Вместо использования `setInterval` для периодического вызова функции в теле функции `gameLoop` выполняется вызов `setTimeout(gameLoop, animationTime)`, что означает «вызови `gameLoop` снова через `animationTime` миллисекунд». Так же как и `setInterval`, это способ снова и снова вызывать функцию с небольшой задержкой между вызовами. Разница в том, что вы можете легко изменить время задержки из любой части своего кода, меняя значение переменной `animationTime`, и программа будет использовать это значение при последующих вызовах `setTimeout`.

(Однако имейте в виду — в таком случае вам нужно найти другой способ остановки игрового цикла, когда игра закончится. Как бы вы это сделали?)

#### #4. Исправление метода apple.move

Каждый раз после съедания яблоко перемещается в новую случайную позицию, однако ничто не мешает ему очутиться в ячейке, которая уже занята частью тела змейки. Чтобы предотвратить эту возможность, измените метод move так, чтобы он учитывал положения сегментов змейкиного тела. (Подсказка: используйте цикл while, чтобы вызывать метод move до тех пор, пока он не выберет позицию, не занятую змейкой).

## ПОСЛЕСЛОВИЕ: КУДА ДВИГАТЬСЯ ДАЛЬШЕ

Теперь, изучив основы языка JavaScript, вы готовы к любым приключениям в необъятном мире программирования. Вы можете изучить еще один язык или углубить свои познания в JavaScript, подняв навыки на новый уровень. Что именно делать — выбирать только вам, однако кое-что я могу посоветовать.

### Больше о JavaScript

В этой книге мы рассмотрели немало возможностей JavaScript, но вам еще многое стоит узнать об этом языке. Вот некоторые книги и сайты, которые помогут вам познакомиться с различными аспектами JavaScript:

- Книга: «JavaScript: сильные стороны», Дуглас Крокфорд (издательство «Питер»).
- Книга: Eloquent JavaScript, 2nd Edition, Marijn Haverbeke (No Starch Press, 2014).
- Книга: «JavaScript. Подробное руководство», Дэвид Флэнаган (издательство «Символ-Плюс», 2013).
- Ресурсы по JavaScript на Mozilla Developer Network:  
<https://developer.mozilla.org/ru/docs/Web/JavaScript> (на русском).
- Курсы JavaScript от Codecademy: <http://www.codecademy.com/en/tracks/javascript/> (на английском).

## Веб-программирование

Чтобы создавать сайты, вам помимо JavaScript понадобятся знания HTML и CSS.

### HTML

HTML — это язык разметки, предназначенный для создания веб-страниц. Мы изучили основы HTML в пятой главе, однако это лишь капля в море. Вот некоторые источники, из которых вы можете узнать об HTML больше:

- Введение в HTML от Mozilla Developer Network: <https://developer.mozilla.org/ru/docs/Web/Guide/HTML/Introduction> (на русском).
- Курс HTML и CSS от Codecademy: <http://www.codecademy.com/tracks/web/> (на английском).
- Mozilla Webmaker: <https://webmaker.org/>.

### CSS

CSS, или «каскадные таблицы стилей», — это язык для управления внешним видом веб-страниц. Узнать о CSS больше можно здесь:

- Введение в CSS от Mozilla Developer Network: [https://developer.mozilla.org/ru/docs/Web/Guide/CSS/Getting\\_started](https://developer.mozilla.org/ru/docs/Web/Guide/CSS/Getting_started) (на русском).
- Курс HTML и CSS от Codecademy: <http://www.codecademy.com/tracks/web/> (на английском).

## Серверное программирование на Node.js

Веб-страницы располагаются на *веб-серверах*. Сервер хранит весь HTML, CSS и JavaScript-код страницы, предоставляя к ней доступ из интернета. Также вы можете писать программы для серверов (так называемый *серверный код*), позволяющие серверу генерировать различный HTML-код при каждой загрузке страницы. Например, когда вы заходите в Twitter, серверная программа ищет самые свежие посты для вашей ленты, генерирует HTML-файл с этими постами и передает его вашему браузеру.

Node.js позволяет писать серверные программы на JavaScript. Ознакомьтесь с информацией о Node.js по этим ссылкам:

- Больше о Node можно узнать здесь: <http://nodejs.org/> (на английском).
- Руководство Node Beginner Book: <http://www.nodebeginner.ru> (на русском).

## Графическое программирование

Если вы хотите создавать интерактивную графику на JavaScript, у вас есть два основных пути: это элемент `canvas` и `SVG`.

### `canvas`

Мы изучили основы работы с элементом `canvas` в книге, однако его возможности гораздо шире. Вот некоторые обучающие статьи и игры, которые помогут вам ближе познакомиться с «холстом»:

- Обучающая статья по `canvas` на Mozilla Developer Network: [https://developer.mozilla.org/ru/docs/Web/API/Canvas\\_API/Tutorial](https://developer.mozilla.org/ru/docs/Web/API/Canvas_API/Tutorial) (на русском).
- Игра Code Monster от Crunchzilla: <http://www.crunchzilla.com/code-monster/> (на английском).

### `SVG` с помощью `Raphael`

`SVG` — это графический формат, позволяющий рисовать разные фигуры и анимировать их без необходимости перерисовывать каждый шаг анимации с нуля.

Программирование `SVG` — дело не из простых, но вам будет куда легче, если вы воспользуетесь JavaScript-библиотекой `Raphael`:

- Введение в библиотеку `Raphael`: <http://code.tutsplus.com/tutorials/an-introduction-to-the-raphael-javascript-library--net-7186/> (на английском).

## 3D-программирование

Помните, как в главе 13 мы запрашивали у «холста» `canvas` двухмерный контекст рисования вызовом `canvas.getContext("2d")`? Кроме этого, на «холсте» можно создавать и трехмерные изображения. Это еще одна из тех областей, где проще всего воспользоваться библиотекой, и я рекомендую вам библиотеку `three.js`. Вот некоторые ресурсы для ее изучения:

- Руководство по `three.js`: <http://threejs.org/docs/index.html#Manual> (на английском).
- `three.js` для начинающих: <http://blog.teamtreehouse.com/the-beginners-guide-to-three-js/> (на английском).

## Программирование роботов

При помощи JavaScript можно даже управлять роботами! Например, Parrot AR.Drone. Это маленький вертолет, которым можно управлять с помощью Node.js. Также вы можете обратить внимание на Johnny-Five, JavaScript-библиотеку для управления устройствами наподобие Arduino (это популярный микроконтроллер, который используется во многих любительских устройствах и роботах). Вот некоторые ресурсы о том, как управлять роботами и другими устройствами из JavaScript-кода:

- node-ar-drone: <https://github.com/felixge/node-ar-drone/> (на английском).
- NodeBots: <http://nodebots.io/> (на английском).
- Johnny-Five: <https://github.com/twaldron/johnny-five/> (на английском).

## Программирование звука

JavaScript также позволяет программировать звук в веб-браузере с помощью интерфейса Web Audio API. Используя этот программный интерфейс, вы можете создавать звуковые эффекты и даже писать музыку! Вот некоторые ресурсы по Web Audio API:

- Web Audio API на Mozilla Developer Network:  
[https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Audio\\_API/](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/) (на английском).
- Статья «HTML5 Rocks: Getting Started with Web Audio API»:  
<http://www.html5rocks.com/en/tutorials/webaudio/intro/> (на английском).

## Программирование игр

Если вы хотите продолжить заниматься программированием JavaScript-игр, вам стоит попробовать какой-нибудь *игровой движок*. Игровой движок — это библиотека кода, который обеспечивает низкоуровневые детали реализации игры (такие как ввод с клавиатуры и мышки), позволяя вам сосредоточиться на том, что делает вашу игру особенной, то есть на дизайне. Вот ресурсы, где вы можете больше узнать о программировании игр и игровых движках:

- Игровой движок Crafty: <http://craftyjs.com/> (на английском).

- Pixi Renderer: <https://github.com/GoodBoyDigital/pixi.js> (на английском).
- Игровые движки для HTML5: <http://html5gameengine.com/> (на английском).
- Курс по разработке HTML5-игр от Udacity: <https://www.udacity.com/course/cs255> (на английском).
- Книга: 3D Game Programming for Kids, Chris Strom (Pragmatic Programmers, 2013) (на английском).

## Обмен кодом с помощью JSFiddle

Что если вам захочется поделиться своим замечательным JavaScript-кодом со всем миром? Есть немало способов это сделать. Один из самых простых — воспользоваться JSFiddle (<http://jsfiddle.net/>). Просто введите ваш код в поле для JavaScript, добавьте любой необходимый HTML-код в поле для HTML и нажмите Run для запуска своей программы. Чтобы поделиться кодом с другими, нажмите Save, и вы получите URL (веб-адрес), который сможете передать друзьям.

## ГЛОССАРИЙ

В мире программирования немало специальных терминов и обозначений. Чтобы привыкнуть к ним, может понадобиться некоторое время. Здесь вы найдете определения терминов, использованных в книге. Если, читая книгу, вы встретите термин, значение которого вам не вполне ясно, загляните сюда за его кратким определением.

**Аргумент** — значение, которое передается в функцию.

**Атрибут** — пара «ключ-значение» в составе HTML-элемента. Атрибуты можно использовать для управления различными свойствами элемента, например адресом, на который он ссылается, или размерами элемента.

**Бесконечный цикл** — цикл, который никогда не прекращается сам по себе (что нередко приводит к сбою интерпретатора). Может быть следствием неверно заданных условий цикла.

**Библиотека** — содержащий функции и методы пакет JavaScript-кода, который можно подгрузить к веб-странице и использовать в своей программе. В этой книге мы пользуемся библиотекой jQuery, предоставляющей функции и методы для удобной работы с DOM.

**Булево значение** — значение, которое может быть либо истинным (`true`), либо ложным (`false`).

**Верблюжья запись** — общепринятый способ наименования переменных, когда отдельные слова, из которых состоит имя переменной

(кроме самого первого слова), пишутся с заглавной буквы; например: myCamelCaseVariable.

**Возврат** — выход из функции и возвращение в точку программы, откуда она была вызвана. Возврат происходит при достижении конца тела функции либо при выполнении оператора `return` (с помощью которого можно выйти из функции досрочно). При этом функция возвращает назад значение (если конкретное возвращаемое значение в функции не указано, будет возвращено пустое значение `undefined`).

**Вызов** — выполнение функции. Чтобы вызвать функцию в JavaScript, нужно ввести ее имя, а затем пару круглых скобок (внутри которых указываются аргументы, если они есть).

**Выполнение** — запуск некоторого кода, например программы или функции.

**Данные** — хранимая в компьютерных программах информация, с которой выполняются некие действия.

**Декремент** — уменьшение значения переменной (обычно на 1).

**Диалог** — небольшое всплывающее окошко. Из JavaScript-кода можно открывать в браузере различные диалоги, такие как `alert` (отображение текстового сообщения) или `prompt` (запрос и получение ввода от пользователя).

**Индекс** — число, соответствующее позиции значения в массиве. С помощью индекса можно получить доступ к определенному значению.

**Ипкремент** — увеличение значения переменной (обычно на 1).

**Интерпретатор** — компьютерная программа,читывающая и исполняющая код. В составе веб-браузеров есть интерпретатор JavaScript, который используется в этой книге для запуска программ.

**Ключевое слово** — слово, которое имеет специальное значение в JavaScript (например, `for`, `return` или `function`). Ключевые слова нельзя использовать в качестве имен переменных.

**Комментарий** — фрагмент текста программы, который игнорируется интерпретатором JavaScript. Комментарии нужны лишь для того, чтобы пояснить работу программы тем, кто читает ее код.

**Конструктор** — разновидность функции, используемой для создания объектов с одинаковыми встроенными свойствами.

**Массив** — список значений в JavaScript. Каждому значению соответствует индекс, то есть порядковый номер этого значения в массиве. Индекс первого элемента равен 0, второго — 1 и т. д.

**Метод** — функция, являющаяся свойством объекта.

**Обработчик события** — функция, которая вызывается, когда определенное событие происходит с определенным HTML-элементом. Например, в игре «Найди клад!» из главы 11 мы создаем функцию — обработчик событий для кликов мышкой по изображению карты.

**Объект** — набор пар «ключ-значение». Каждый ключ — это строка, которую можно связать с любым JavaScript-значением. Зная ключ, можно получить из объекта связанное с ним значение.

**Объектно-ориентированное программирование** — стиль программирования, подразумевающий использование объектов и методов для структурирования кода и реализации основных возможностей программы.

**Пара «ключ-значение»** — пара, состоящая из строки (называемой *ключом*) и связанного с ней значения (любого типа). Пары «ключ-значение» содержатся в JavaScript-объектах и позволяют задавать их свойства и методы.

**Переменная** — способ связывания имени со значением. После того как переменной присвоено значение, к нему можно обращаться через имя переменной.

**Пробелы** — неотображаемый на экране символ, например пробел, перенос строки, табуляция.

**Свойство** — имя пары «ключ-значение», принадлежащей объекту.

**Синтаксис** — правила объединения ключевых слов, знаков пунктуации и других символов в работающую JavaScript-программу.

**Событие** — действие в браузере, например клик мышкой или нажатие клавиши. События можно как отслеживать, так и реагировать на них — при помощи обработчиков событий.

**Строка** — последовательность символов, окруженная кавычками; представление текста в компьютерных программах.

**Строка селектора** — строка, обозначающая один или несколько HTML-элементов, которую можно передать jQuery-функции для поиска этих элементов.

**Тег** — сетка, используемая для создания HTML-элементов. Все элементы начинаются с открывающего тега, и большинство из них заканчиваются закрывающим тегом. Теги определяют тип создаваемого элемента. Кроме того, в открывающем теге можно задать атрибуты элемента.

**Текстовый редактор** — компьютерная программа для написания и редактирования простого текста без специального форматирования (например, выбора различных шрифтов и цветов). Для создания программ (которые пишутся простым текстом) желателен хороший текстовый редактор.

**Управляющая конструкция** — способ контроля за тем, когда и сколько раз выполняется фрагмент кода. Например, условные конструкции (по заданному условию определяющие, когда выполнять код) и циклы (выполняющие фрагмент кода повторно определенное количество раз).

**Условная конструкция** — конструкция языка, которая выполняет код в зависимости от проверки условия. Если условие истинно (`true`), выполняется один фрагмент кода, а если условие ложно (`false`) — либо другой фрагмент, либо не делается ничего. Примеры условных конструкций: операторы `if` и `if... else`.

**Функция** — состоящий из одной или более команд фрагмент кода, который можно вызывать (выполнять). С помощью функции можно повторять один и тот же набор действий в разных частях программы. Функция может принимать аргументы и возвращать обратно значение.

**Цикл** — способ многократного выполнения фрагмента кода.

**Элемент** — часть HTML-документа; например, заголовок, параграф или тело (`body`). Элемент обозначается открывающим и закрывающим тегами (которые определяют тип элемента) и включает в себя все, что находится между этими тегами. Дерево DOM состоит из элементов.

**Язык программирования** — язык, с помощью которого программист может объяснить компьютеру, как нужно выполнить некую задачу. JavaScript — один из множества языков программирования.

**DOM (объектная модель документа)** — способ, которым веб-браузеры упорядочивают HTML-элементы на веб-странице. Элементы организованы в виде древовидной структуры, которую называют деревом DOM. В JavaScript и jQuery есть методы для работы с DOM, то есть поиска, создания и изменения элементов.

**jQuery** — JavaScript-библиотека, предоставляющая множество методов для работы с элементами DOM на веб-странице.

**Null** — специальное значение, благодаря которому мы видим, что переменная намеренно оставлена пустой.

**Prototype** — свойство конструктора. Методы, добавленные к свойству `prototype`, становятся доступны всем объектам, созданным через этот конструктор.

**Undefined** — специальное значение, которое JavaScript использует, чтобы показать, что некоему свойству или переменной не было присвоено конкретное значение.

## ОБ АВТОРЕ

Ник Морган — фронтенд-разработчик в компании Twitter. Он любит все языки программирования, но к JavaScript питает особую нежность. Ник живет в Сан-Франциско (в его туманной части) со своей невестой и их пушистой собакой Оладушком. Ник ведет блог по адресу [skilldrick.co.uk](http://skilldrick.co.uk).

### О художнике

Миран Липовача — автор книги «Изучай Haskell во имя добра!» (издательство «ДМК-Пресс», 2012). Он обожает боксировать, играть на бас-гитаре и, разумеется, рисовать. Он неравнодушен к танцующим скелетам и числу 71, а проходя через автоматические двери, делает вид, будто открывает их силой разума.

### О техническом редакторе

Перевод статьи можно почитать здесь: <https://m.habrahabr.ru/post/183838/?mobile=yes>

Ангус Кролл — автор статьи «Если бы Хемингуэй писал на JavaScript», и он в равной мере одержим языком JavaScript и литературой. Ангус работает в Twitter, в команде UI-фреймворков, и является одним из авторов фреймворка Flight. Он ведет авторитетный блог JavaScript, JavaScript и выступает на конференциях по всему миру.

## БЛАГОДАРНОСТИ

Тысяча благодарностей моей невесте Филли за помощь и поддержку в течение последних 18 месяцев. Без нее я бы действительно не спрвился. И спасибо Оладушку, нашему песику, за великодушное разрешение использовать его имя в примерах кода.

Спасибо Ангусу — если бы не он, я не сидел бы здесь, в Сан-Франциско, работая над этой книгой. Ангус порекомендовал меня компании Twitter в 2011 году, а в 2013 году сказал Биллу Поллоку, что идея написать книгу, которую вы держите сейчас в руках, может меня заинтересовать. И в довершение он согласился быть техническим редактором, выловив из текста немалое количество JavaScript-ляпсусов.

Спасибо Биллу Поллоку, Сефу Крамеру, Рили Хоффман, Тайлеру Ортману и остальным сотрудникам издательства No Starch Press, которые терпеливо помогали мне писать эту книгу. Особая благодарность Биллу и Сефу за помощь в приведении первоначального текста к его нынешнему виду.

Спасибо юным рецензентам Ривер Брэдли, Дэмиену Чемпу и Алексу Чу, от которых я получил ценный отклик по ранним версиям книги.

И наконец, спасибо Мирану Липоваче. Я давний его поклонник: книга Мирана «Изучай Haskell во имя добра!» — одна из моих любимых книг по программированию, а его иллюстрации к ней просто великолепны. Когда я узнал, что он будет иллюстратором моей книги, я был на седьмом небе от радости. Картинки, которые Миран нарисовал для этой книги, лучше, чем я даже мог себе вообразить, и я счастлив, что мне выпала удача с ним работать.

*Издание для досуга  
Для широкого круга читателей*

Морган Ник

**JavaScript для детей**  
Самоучитель по программированию

Главный редактор Артем Степанов  
Руководитель направления Анастасия Троян  
Ответственный редактор Анна Дружинец  
Литературный редактор Лев Эйделькинд  
Научный редактор Дарья Абрамова  
Дизайн обложки Сергей Хозин  
Верстка Надежда Кудрякова, Елена Бреге  
Корректоры Юлия Молокова, Надежда Болотина

ОТ ОДНОГО  
ИЗ РАЗРАБОТЧИКОВ  
TWITTER



ОДИН ИЗ САМЫХ  
ВОСТРЕБОВАННЫХ  
ЯЗЫКОВ  
ПРОГРАММИРОВАНИЯ

JavaScript — это язык программирования веб-страниц, тот самый ингредиент, который делает сайты интерактивными. Эта книга позволит вам погрузиться в программирование и с легкостью освоить JavaScript. На каждом шаге вы будете видеть результаты своих трудов в виде работающей программы. А с понятными инструкциями, примерами и забавными иллюстрациями обучение будет только приятным.

Эта книга подойдет детям от 10 лет (и их родителям!), а также всем, кто хочет начать программировать с нуля на любом языке или освоить именно JavaScript.

### Вы узнаете, как:

- использовать основные элементы JavaScript: базовые типы, операторы, массивы и функции;
- создавать веб-страницы с помощью HTML;
- делать веб-страницы интерактивными;
- рисовать при помощи JavaScript;
- написать собственную игру — поиск сокровищ на карте, «Виселицу» и «Змейку».

Автор книги Ник Морган — frontend-разработчик в Twitter. Ник обожает язык JavaScript и делится его секретами с искренней любовью.

МИФ  
ДЕТСТВО

Детские книги на сайте  
[mann-ivanov-ferber.ru](http://mann-ivanov-ferber.ru)

[facebook.com/mifdetstvo](https://facebook.com/mifdetstvo)  
 [vk.com/mifdetstvo](https://vk.com/mifdetstvo)  
 [@instagram.com/mifdetstvo](https://instagram.com/mifdetstvo)



ISBN 978-5-00100-295-6



9 785001 002956 >