Эрик Фримен Элизабет Робсон

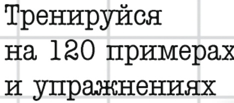
Изучаем программирование на JavaScript

Руководство по программированию на JavaScript

Научись обходить «подводные камни»

Начни карьеру программиста с одной главы

Не ошибайся в преобразованиях типов



Тренируйся на 120 примерах и упражнениях

Узнай, почему твои друзья ничего не понимают в функциях и объектах

O'REILLY® ПИТЕР®

О других книгах серии *Head First*

«Правильно выбранный тон для внутреннего раскрепощенного эксперта-программиста, скрывающегося в каждом из нас. Отличный справочник по практическим стратегиям разработки — мой мозг работает, не отвлекаясь на надоедливый, устаревший академический жаргон».

— Трэвис Каланик, директор Uber

«Замечательная ясность, юмор и изрядная доля интеллекта заставят даже непрограмиста положительно взглянуть на методику решения задач».

— Кори Доктороу, второй редактор Boing Boing, писатель-фантаст

«У меня такое чувство, словно я прочитал сразу полтонны книг».

— Уорд Каннингем, изобретатель Wiki

«Это одна из немногих книг по программированию, которую я считаю незаменимой (а я к этой категории причисляю книг десять, не более)».

— Дэвид Гелернтер, профессор по компьютерным технологиям, Йельский университет

«Я смеялся, я плакал, эта книга тронула меня».

— Дэниел Стейнберг, старший редактор java.net

«Не могу представить себе лучших экскурсоводов, чем Эрик и Элизабет».

— Мико Мацумура, вице-президент отдела маркетинга в Hazelcast, бывший ведущий специалист по Java, Sun Microsystems

«Я буквально влюблен в эту книгу. Я даже поцеловал ее на глазах у жены».

— Сатиш Кумар

«Визуальный подход и последовательное изложение — лучший способ изучения этого материала...»

— Дэнни Гудман, автор книги *Dynamic HTML: The Definitive Guide*

«В отличие от многих невразумительных книг по программированию, насыщенных техническим жаргоном, руководства серии *Head First jQuery* помогают новичкам создавать их первые страницы jQuery на простом и доступном уровне».

— Линдси Скурас, юрист и программист-самоучка

«Очевидно, Эрик и Элизабет знают свое дело. Интернет-технологии становятся все более сложными, и творческое создание веб-страниц начинает играть все более важную роль. Элегантная архитектура занимает центральное место в каждой главе, каждая концепция передается с равной дозой прагматизма и остроумия».

— Кен Голдстейн, бывший директор Shop.com и автор книги *This is Rage: A Novel of Silicon Valley and Other Madness*

«Книга *Изучаем HTML, XHTML и CSS* представляет собой тщательно проработанное современное руководство по дальновидным практикам в области разметки и представления веб-страниц. Авторы предвидят, какие моменты могут вызвать у читателя замешательство, и своевременно разъясняют их. Использованный подход, в основе которого лежат обилие наглядных примеров и последовательность изложения, является оптимальным для читателя: он будет вносить небольшие изменения и наблюдать итоговый эффект в браузере, что позволит разобраться в назначении каждого нового элемента».

— Дэнни Гудмен, автор книги *Динамический HTML: подробное руководство» (Dynamic HTML: The Definitive Guide)*

«Книга *Изучаем HTML, XHTML и CSS* с самого начала создает у читателя ощущение, что весь процесс обучения окажется простым и увлекательным. Освоение HTML при правильном объяснении не сложнее изучения основ родного языка, при этом авторы проделали отличную работу и приводят наглядные примеры по каждой концепции».

— Майк Дэвидсон, президент и исполнительный директор Newsvine, Inc

«Вместо изложения материала в стиле традиционных учебников *Программируем для iPhone и iPad* предлагает читателю живую, увлекательную и даже приятную методику обучения программированию для iOS. Материал подобран умело и качественно: в книге рассматриваются многие ключевые технологии, включая Core Data, и даже такие важные аспекты, как проектирование интерфейса. И где еще можно прочитать, как UIWebView и UITextField беседуют у камина?»

— Шон Мерфи, проектировщик и разработчик приложений для iOS

«Книга *Программируем для iPhone и iPad* объясняет принципы разработки приложений iOS с самого начала. Основные изменения по сравнению с первым изданием относятся к iOS 4, Xcode 4 и написанию приложений для iPad. Благодаря пошаговым описаниям с визуальным стилем изложения материала эта книга становится отличным средством изучения программирования для iPhone и iPad во всех аспектах, от простейших до нетривиальных».

— Рич Розен, программист и соавтор книги *Mac OS X for Unix Geeks*

Head First JavaScript Programming

Eric T. Freeman
Elisabeth Robson

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Tokyo

Изучаем программирование

на JavaScript

Хорошо бы найти книгу по JavaScript, которая была бы приятнее визита к зубному врачу и понятнее налоговой декларации? Как жаль, что это только мечты...

Эрик Фримен
Элизабет Робсон

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2015

ББК 32.988-02-018.1
УДК 004.43
Ф88

Фримен Э., Робсон Э.

Ф88 Изучаем программирование на JavaScript. — СПб.: Питер, 2015. — 640 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-496-01257-7

Вы готовы сделать шаг вперед в веб-программировании и перейти от верстки в HTML и CSS к созданию полноценных динамических страниц? Тогда пришло время познакомиться с самым «горячим» языком программирования — JavaScript!

С помощью этой книги вы узнаете все о языке JavaScript: от переменных до циклов. Вы поймете, почему разные браузеры по-разному реагируют на код и как написать универсальный код, поддерживаемый всеми браузерами. Вам станет ясно, почему с кодом JavaScript никогда не придется беспокоиться о перегруженности страниц и ошибках передачи данных. Не пугайтесь, даже если ранее вы не написали ни одной строчки кода, — благодаря уникальному формату подачи материала эта книга с легкостью проведет вас по всему пути обучения: от написания простейшего скрипта до создания сложных веб-проектов, которые будут работать во всех современных браузерах.

Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

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

ББК 32.988-02-018.1
УДК 004.43

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

ISBN 978-1449340131 англ.
© Authorized Russian translation of the English edition of Head First JavaScript Programming, 1st Edition (ISBN 9781449340131)
© 2014 Eric Freeman, Elisabeth Robson. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same
ISBN 978-5-496-01257-7
© Перевод на русский язык ООО Издательство «Питер», 2015
© Издание на русском языке, оформление ООО Издательство «Питер», 2015

Посвящается JavaScript — ты не родился в благополучной семье, но превзошел все языки, которые пытались конкурировать с тобой в браузерах.

Авторы книги

Элизабет Робсон

Эрик, по словам одного из создателей серии Head First — «одна из редких личностей, хорошо разбирающихся в языке, практике и культуре самых разных областей — технохипстер, вице-президент, инженер, аналитик».

Почти десять лет Эрик отработал на руководящей должности — технического директора Disney Online & Disney.com в компании The Walt Disney Company. Сейчас Эрик отдает свое время WickedlySmart — молодой компании, которую он создал совместно с Элизабет.

Эрик — специалист по компьютерным технологиям, он занимался исследованиями вместе с Дэвидом Гелернтером в Йельском университете. Его диссертация стала фундаментальным трудом в области интерфейсов, реализующих метафору рабочего стола, а также первой реализацией потоков активности — концепции, разработанной им совместно с Гелернтером.

В свободное время Эрик серьезно занимается музыкой; последний проект Эрика — Immersion Station, — созданный вместе со Стивом Роучем, можно найти в iPhone App Store.

Эрик живет с женой и дочерью на острове Бейнбридж. Его дочь часто заглядывает в музыкальную студию Эрика, чтобы поиграть с синтезаторами и генераторами аудиоэффектов.

Пишите Эрику по адресу `eric@wickedlysmart.com` или посетите его сайт `http://ericfreeman.com`.

Элизабет — программист, писатель и преподаватель. Она влюблена в свою работу еще со времени учебы в Йельском университете, где она получила степень магистра в области компьютерных технологий.

Элизабет уже давно занимается Интернетом; она участвовала в создании популярного сайта Ada Project — одного из первых сайтов, помогающих женщинам найти информацию о работе и образовании в области компьютерных технологий.

Она стала одним из учредителей WickedlySmart — фирмы, работающей в области интернет-образования на базе веб-технологий. Здесь она создает книги, статьи, видеокурсы и т. д. На должности директора по специальным проектам в O'Reilly Элизабет разрабатывала семинары и курсы дистанционного обучения. Так проявилась ее страсть к созданию учебных курсов, помогающих людям разобраться в новых технологиях. До прихода в O'Reilly Элизабет работала в The Walt Disney Company, где руководила исследованиями и разработками в сфере цифровых мультимедийных технологий.

Когда Элизабет не сидит за компьютером, она ходит в походы, занимается велоспортом и греблей или готовит вегетарианские блюда.

Вы можете написать Элизабет по адресу `beth@wickedlysmart.com` или посетить ее блог `http://elisabethrobson.com`.

Содержание (сводка)

	Введение	25
1	Первое знакомство с JavaScript. *В незнакомых водах*	37
2	Настоящий код. *Следующий шаг*	79
3	Знакомство с функциями. *Функции для всех*	113
4	Наведение порядка в данных. *Массивы*	157
5	Знакомьтесь: объекты. *Поездка в Объектвиль*	203
6	Взаимодействие с веб-страницей. *Модель DOM*	257
7	Типы, равенство, преобразования и все такое. *Серьезные типы*	291
8	Все вместе. *Построение приложения*	341
9	Асинхронное программирование. *Обработка событий*	403
10	Первоклассные функции. *Функции без ограничений*	449
11	Анонимные функции, область действия и замыкания. *Серьезные функции*	495
12	Нетривиальное создание объектов. *Создание объектов*	539
13	Использование прототипов. *Сильные объекты*	579

Содержание (настоящее)

Введение

Ваш мозг и JavaScript. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему. Ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту очевидно несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. Так как же заставить его изучить JavaScript?

Для кого написана эта книга	24
Мы знаем, о чем вы думаете	25
Эта книга для тех, кто хочет учиться	26
Метапознание: наука о мышлении	27
Вот что сделали МЫ:	28
Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться	29
Примите к сведению	30
Научные редакторы	33
Благодарности	34

1

первое знакомство с javascript

В незнакомых водах

JavaScript открывает фантастические возможности. JavaScript, основной язык программирования Всемирной паутины, позволяет определять расширенное поведение в веб-страницах. Забудьте о сухих, скучных, статичных страницах, которые просто занимают место на экране, — с JavaScript вы будете взаимодействовать с пользователями, реагировать на события, получать и использовать данные из Интернета, выводить графику... и многое, многое другое. При хорошем знании JavaScript вы сможете даже программировать совершенно новое поведение на своих страницах.

И не сомневайтесь — ваши знания будут востребованы. Сейчас JavaScript не только является одним из самых популярных языков программирования, но и поддерживается всеми современными (и многими несовременными) браузерами; более того, появились встроенные реализации JavaScript, существующие отдельно от браузеров. А впрочем, хватит разговоров. Пора браться за дело!

Как работает JavaScript 38
Как пишется код JavaScript 39
Как включить код JavaScript в страницу 40
JavaScript, ты проделал длинный путь, детка... 42
Как создаются команды 46
Переменные и значения 47
Осторожно, ключевые слова! 48
Поаккуратнее с выражениями! 51
Многократное выполнение операций 53
Как работает цикл while 54
Принятие решений в JavaScript 58
А если нужно принять МНОГО решений... 59
Привлекайте пользователя к взаимодействию со страницей 61
Близкое знакомство с console.log 63
Как открыть консоль 64
Пишем серьезное приложение на JavaScript 65
Как добавить код в страницу? (считаем способы) 68
Разметка и код: пути расходятся 69

2

настоящий код

Следующий шаг

Вы уже знаете, что такое переменные, типы, выражения... и так далее. Вы уже кое-что знаете о JavaScript. Более того, знаний достаточно для того, чтобы начать писать настоящие программы, которые делают что-то интересное, которыми кто-то будет пользоваться. Правда, вам не хватает практического опыта написания кода, и мы прямо сейчас начнем решать эту проблему. Как? А просто возьмемся за написание несложной игры, полностью реализованной на JavaScript. Задача масштабная, но мы будем двигаться к цели постепенно, шаг за шагом. Итак, беремся за дело, а если вам вдруг захочется использовать нашу разработку в своих проектах — мы не против, распоряжайтесь кодом, как считаете нужным.

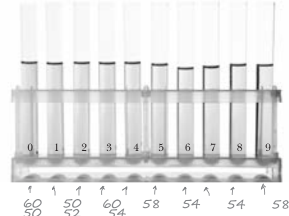
Давайте реализуем игру «Морской бой» 80
Первый заход... 80
Начнем с проектирования 81
Разбираем псевдокод 83
Стоп! Прежде чем идти дальше, вспомните про HTML! 85
Пишем код упрощенной версии «Морского боя» 86
Переходим к реализации логики 87
Как работает функция prompt 89
Проверка на попадание 90
Добавление кода проверки попадания 93
Вывод данных после игры 94
Реализация логики готова! 96
Немного о контроле качества 97
А нельзя ли покороче... 101
Упрощенный «Морской бой» почти готов 102
Как получить случайную позицию 103
Всемирно известный рецепт генерирования случайных чисел 103
Возвращаемся к контролю качества 105
Поздравляем, вы создали свою первую программу на JavaScript! Теперь пара слов о повторном использовании кода 107

знакомство с функциями

3 Функции для всех

В этой главе вы овладеете своей первой суперспособностью. Вы уже кое-что знаете о программировании; пришло время сделать следующий шаг и освоить работу с функциями. Функции позволяют писать код, который может повторно использоваться в разных ситуациях; код, существенно более простой в сопровождении; код, который можно абстрагировать и присвоить ему простое имя, чтобы вы могли забыть о рутинных подробностях и заняться действительно важными делами. Вы увидите, что функции не только открывают путь к мастерству программиста, но и играют ключевую роль в стиле программирования JavaScript. В этой главе мы начнем с основ: механики и всех тонкостей работы функций, а потом в оставшейся части книги будем совершенствовать ваши навыки работы с функциями. Итак, начнем с азов... прямо сейчас.

Так чем плох этот код? 115
Кстати, а вы когда-нибудь слышали о ФУНКЦИЯХ? 117
Хорошо, но как все это работает? 118
Что можно передать функции? 123
В JavaScript используется передача по значению 126
Эксперименты с функциями 128
А еще функции могут возвращать значения 129
Пошаговое выполнение функции с командой return 130
Глобальные и локальные переменные 133
Область действия локальных и глобальных переменных 135
Короткая жизнь переменных 136
Не забывайте объявлять локальные переменные! 137



наведение порядка в данных

4 Массивы

JavaScript может работать не только с числами, строками и логическими значениями. До настоящего момента мы работали исключительно с **примитивами** — простыми строками, числами и логическими значениями (например, «Fido», 23 и true). С примитивными типами можно сделать многое, но в какой-то момент возникнет необходимость в **расширенных данных** для представления всех позиций в корзине покупок, всех песен в плейлисте, группы звезд и их звездных величин или целого каталога продуктов. Подобные задачи требуют более серьезных средств. Типичным инструментом для представления таких однородных данных является **массив** JavaScript. В этой главе вы узнаете, как помещать данные в массив, передавать их и работать с ними. В последующих главах будут рассмотрены другие способы **структурирования данных**, но начнем мы с массивов.

Вы нам поможете? 158
Как представить набор значений в JavaScript 159
Как работают массивы 160
Сколько же элементов в массиве? 162
Генератор Красивых Фраз 164
Тем временем в фирме Bubbles-R-Us... 167
Как перебрать элементы массива 170
Но постойте, существует и более удобный способ перебора! 172
Что, опять?.. Нельзя ли покороче? 178
Доработка цикла for с оператором постфиксного увеличения 179
Создание пустого массива (и добавление элементов) 183
А вот и наши победители... 187
Краткий обзор кода... 189
Работа над функцией printAndGetHighScore 190
Рефакторинг кода с определением функции printAndGetHighScore 191
А теперь все вместе... 193

знакомьтесь: объекты

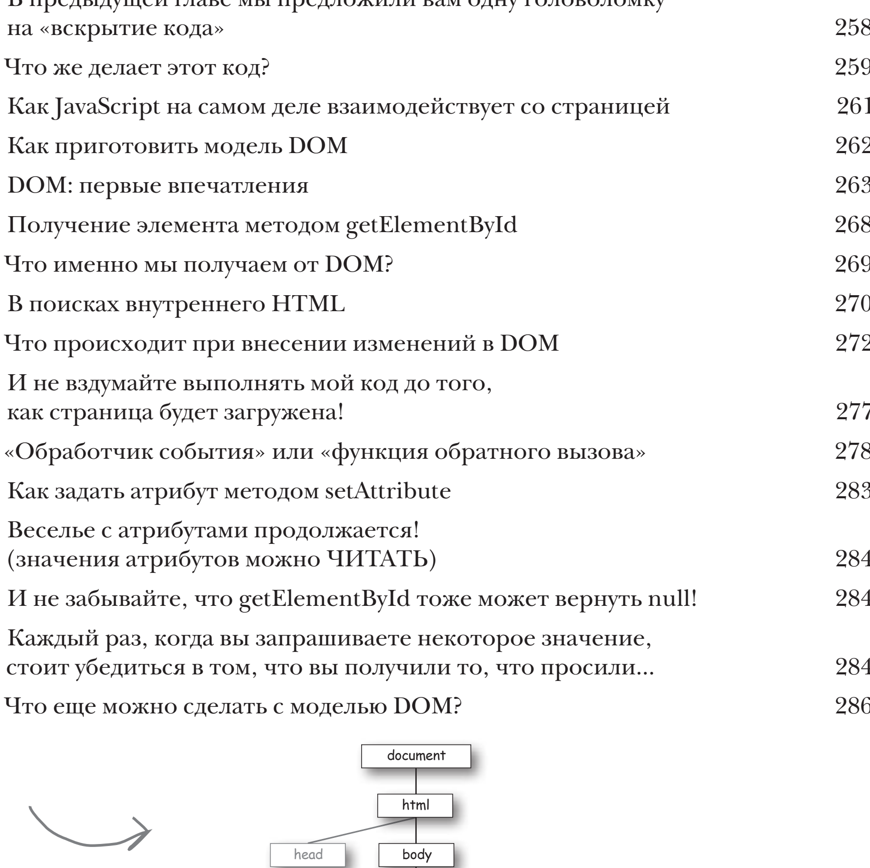
Поездка в Объектвиль

До настоящего момента мы использовали примитивы и массивы. И при этом применялась методология **процедурного программирования с простыми командами, условиями, циклами** for/while и функциями. Такой подход был далек от принципов **объектно-ориентированного программирования**. Собственно, он вообще не имел *ничего общего* с объектно-ориентированным программированием. Мы использовали объекты время от времени (причем вы об этом даже не знали), но еще не написали ни одного собственного объекта. Пришло время покинуть скучный процедурный город и заняться созданием собственных **объектов**. В этой главе вы узнаете, почему объекты сильно улучшают нашу жизнь — во всяком случае в области **программирования**. Так и знайте: привыкнув к объектам, вы уже не захотите возвращаться обратно. Да, и не забудьте прислать открытку, когда обживетесь.

Кто-то сказал «объекты»?! 204
Подробнее о свойствах... 205
Как создать объект 207
Что такое «объектно-ориентированный подход»? 210
Как работают свойства 211
Как объект хранится в переменной?
Любознательные умы интересуются... 216
Сравнение примитивов с объектами 217
Объекты способны на большее... 218
Предварительная проверка 219
Проверка шаг за шагом 220
Еще немного поговорим о передаче объектов функциям 222
Ведите себя прилично! И объекты свои научите... 228
Усовершенствование метода drive 229
Почему метод drive не знает о свойстве started? 232
Как работает this 234
Как поведение влияет на состояние 240
Состояние влияет на поведение 241
Поздравляем с первыми объектами! 243
Представьте, вас окружают сплошные объекты!
(и они упрощают вашу работу) 244

Модель DOM

Вы значительно продвинулись в изучении JavaScript. Фактически вы из новичка в области сценарного программирования превратились в... **программиста**. Впрочем, кое-чего не хватает: для полноценного использования ваших навыков JavaScript необходимо уметь взаимодействовать с веб-страницей, в которой располагается ваш код. Только в этом случае вы сможете писать **динамические** страницы, которые реагируют на действия пользователя и обновляются после загрузки. Как взаимодействовать со страницей? Через объектную модель документа **DOM** (**Document Object Model**). В этой главе мы рассмотрим DOM и общие принципы работы с этой моделью из JavaScript для расширения возможностей страницы.



В предыдущей главе мы предложили вам одну головоломку на «вскрытие кода» 258

Что же делает этот код? 259

Как JavaScript на самом деле взаимодействует со страницей 261

Как приготовить модель DOM 262

DOM: первые впечатления 263

Получение элемента методом getElementById 268

Что именно мы получаем от DOM? 269

В поисках внутреннего HTML 270

Что происходит при внесении изменений в DOM 272

И не вздумайте выполнять мой код до того, как страница будет загружена! 277

«Обработчик события» или «функция обратного вызова» 278

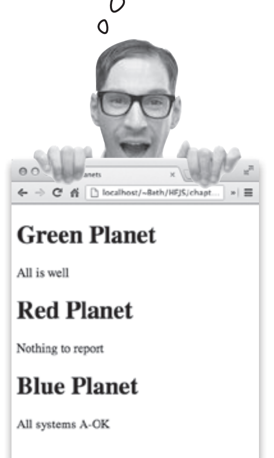
Как задать атрибут методом setAttribute 283

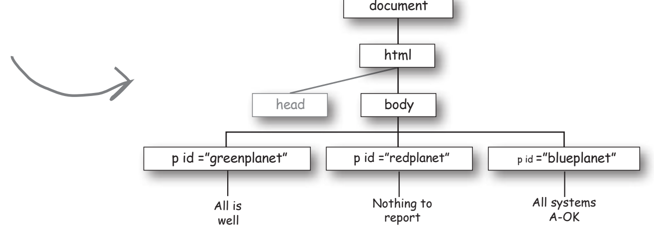
Веселье с атрибутами продолжается! (значения атрибутов можно ЧИТАТЬ) 284

И не забывайте, что getElementById тоже может вернуть null! 284

Каждый раз, когда вы запрашиваете некоторое значение, стоит убедиться в том, что вы получили то, что просили... 284

Что еще можно сделать с моделью DOM? 286



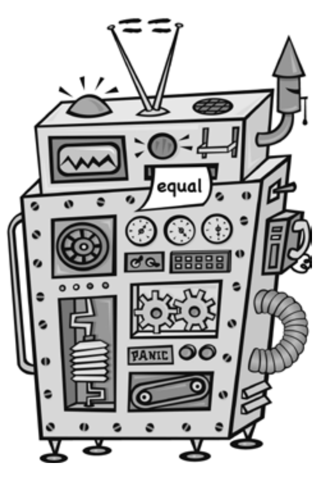


7

типы, равенство, преобразования и все такое

Серьезные типы

Настало время серьезно поговорить о типах. Одна из замечательных особенностей JavaScript заключается в том, что начинающий может достаточно далеко продвинуться, не углубляясь в подробности языка. Но чтобы действительно **овладеть языком**, получить повышение по работе и заняться тем, чем действительно стоит заниматься, нужно хорошо разбираться в **типах**. Помните, что мы говорили о JavaScript, — что у этого языка не было такой роскоши, как академическое определение, прошедшее экспертную оценку? Да, это правда, но отсутствие академической основы не помешало Стиву Джобсу и Биллу Гейтсу; не помешало оно и JavaScript. Это означает, что система типов JavaScript... ну, скажем так — не самая продуманная, и в ней найдется немало **странностей**. Но не беспокойтесь, в этой главе мы все разберем, и вскоре вы научитесь благополучно обходить все эти неприятные моменты с типами.



Истина где-то рядом... 292

Будьте осторожны: undefined иногда появляется совершенно неожиданно... 294

Как использовать null 297

Работа с NaN 299

А дальше еще удивительнее 299

Мы должны вам признаться... 301

Оператор проверки равенства (также известный как ==) 302

Как происходит преобразование операндов (все не так страшно, как может показаться) 303

Как проверить строгое равенство 306

Еще больше преобразований типов... 312

Как проверить два объекта на равенство 315

Псевдоистина где-то рядом... 317

Что JavaScript считает «псевдоложью» 318

Тайная жизнь строк 320

Строка может выглядеть и как примитив, и как объект 321

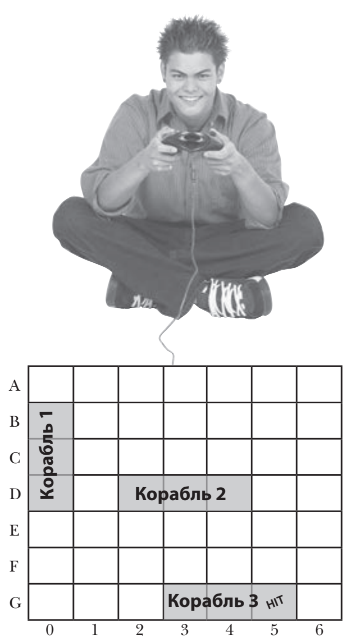
Краткий обзор методов (и свойств) строк 323

Битва за кресло 327

Все вместе

Построение приложения

Подготовьте свой инструментарий к работе. Да, ваш инструментарий — ваши новые навыки программирования, ваше знание DOM и даже некоторое знание HTML и CSS. В этой главе мы объединим все это для создания своего первого полноценного **веб-приложения**. Довольно **примитивных игр** с одним кораблем, который размещается в одной строке. В этой главе мы построим **полную версию**: большое игровое поле, несколько кораблей, ввод данных пользователем прямо на веб-странице. Мы создадим структуру страницы игры в разметке HTML, применим визуальное оформление средствами CSS и напишем код JavaScript, определяющий поведение игры. Приготовьтесь: в этой главе мы займемся полноценным, серьезным программированием и напишем вполне серьезный код.

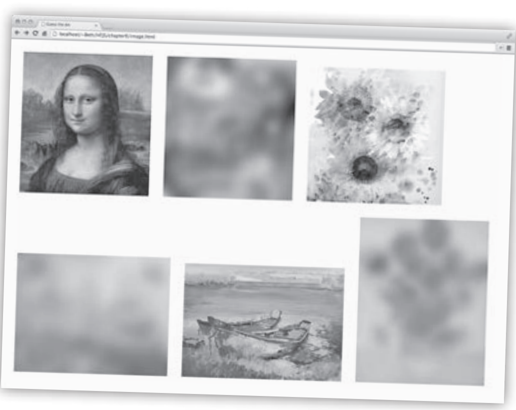


На этот раз мы построим НАСТОЯЩУЮ игру «Морской бой» 342
Возвращаемся к HTML и CSS 343
Создание страницы HTML: общая картина 344
Добавление стилевого оформления 348
Использование классов hit и miss 351
Как спроектировать игру 353
Реализация представления 355
Как работает метод displayMessage 355
Как работают методы displayHit и displayMiss 357
Модель 360
Как мы будем представлять данные кораблей 362
Реализация объекта модели 365
Подготовка метода fire 366
Реализация контроллера 373
Обработка выстрела 374
Планирование кода... 375
Реализация метода parseGuess 376
Подсчет и обработка выстрелов 379
Как связать обработчик событий с кнопкой Fire 383
Передача данных контроллеру 384
Как размещать корабли 388
Метод generateShip 389
Генерирование начальной позиции нового корабля 390
Завершение метода generateShip 391

асинхронное программирование

Обработка событий

В этой главе вам предстоит подняться на принципиально новый уровень. До настоящего момента мы писали код, который обычно выполняется сверху вниз. Конечно, в нем использовались функции, объекты и методы, но выполнение шло по заранее намеченной колее. Жаль, что нам приходится сообщать такие новости во второй половине книги, но **такая структура кода не характерна для JavaScript**. Большая часть кода JavaScript пишется **для обработки событий**. Каких событий? Да любых. Пользователь щелкает на странице, данные поступают из сети, в браузере срабатывает таймер, в DOM происходят изменения... Это далеко не полный список. Более того, в браузере **постоянно** происходят события, которые в основном остаются незамеченными. В этой главе мы пересмотрим свой подход к программированию и узнаем, для чего же нужно писать код, реагирующий на события.



Что такое «событие»? 405

Что такое «обработчик события»? 406

Как создать первый обработчик событий 407

Тест-драйв 408

Как разобраться в событиях? Написать игру, конечно! 410

Реализация игры 411

Тест-драйв 412

Добавим несколько изображений 416

Теперь нужно назначить один обработчик всем свойствам onclick всех изображений 417

Как использовать один обработчик для всех изображений 418

Как работает объект события 421

Работаем с объектом события 423

Тест-драйв: объект события и источник 424

Очереди и события 426

Еще больше событий 429

Как работает setTimeout 430

Завершение кода игры 434

Тест-драйв таймеров 435

первоклассные функции

Функции без ограничений

Изучайте функции и блистайте. В каждом ремесле, искусстве и дисциплине есть ключевой принцип, который отличает игроков «среднего звена» от настоящего профессионала, — и когда речь заходит о JavaScript, признаком профессионализма является хорошее понимание **функций**. Функции играют фундаментальную роль в JavaScript, и многие приемы, применяемые при **проектировании и организации** кода, основаны на хорошем знании функций и умении использовать их. Путь изучения функций на этом уровне интересен и непрост, так что приготовьтесь... Эта глава немного напоминает экскурсию по шоколадной фабрике Вилли Вонка — во время изучения функций JavaScript вы увидите немало странного, безумного и замечательного.

Двойная жизнь ключевого слова function 450
Объявления функций и функциональные выражения 451
Разбор объявления функции 452
Что дальше? Браузер выполняет код 453
Двигаемся вперед... Проверка условия 454
И напоследок... 455
Функции как значения 459
Функции как полноправные граждане JavaScript 462
Полеты первым классом 463
Написание кода для обработки и проверки пассажиров 464
Перебор пассажиров 466
Передача функции другой функции 467
Тест-драйв... вернее, полет 467
Возвращение функций из функций 470
Код заказа напитков 471
Код заказа напитков: другой подход 472
Постойте, одного напитка недостаточно! 473
Заказ напитков с использованием первоклассной функции 474
Тест-драйв-полет 475
«Веб-кола» 477
Как работает метод массивов sort 479
Все вместе 480
Тем временем в «Веб-коле» 481
Тест-драйв сортировки 482

11 анонимные функции, область действия и замыкания

Серьезные функции

Мы узнали много нового о функциях, но это далеко не всё. В этой главе мы пойдем дальше и разберемся в темах, которыми обычно занимаются профессионалы. Вы научитесь **действительно эффективно** работать с функциями. Глава будет не слишком длинной, но с довольно интенсивным изложением материала, так что к концу главы выразительность вашего кода JavaScript превзойдет все ожидания. Вы также будете готовы к тому, чтобы взяться за анализ кода коллеги или заняться изучением библиотеки JavaScript с открытым кодом, потому что мы заодно изучим некоторые распространенные идиомы и соглашения, связанные с использованием функций. А если вы никогда не слышали об **анонимных функциях** и **замыканиях**, то это самое подходящее место для знакомства!

Посмотрим на функции с другой стороны... 496
Как использовать анонимную функцию? 497
Когда определяется функция? Здесь возможны варианты... 503
Что произошло? Почему функция fly не определена? 504
Как создаются вложенные функции 505
Как вложение влияет на область действия 506
В двух словах о лексической области действия 508
Чем интересна лексическая область действия 509
Снова о функциях 511
Вызовы функций (снова) 512
Что такое «замыкание»? 515
Как замкнуть функцию 516
Использование замыканий для реализации счетчика 518
Тест-драйв волшебного счетчика 519
Взгляд за кулисы 519
Создание замыкания с передачей функционального выражения в аргументе 521
Замыкание содержит непосредственное окружение, а не его копию 522
Создание замыкания в обработчике события 523
Программа без замыкания 524
Программа с замыканием 524
Тест-драйв счетчика нажатий 525
Как работает замыкание 526

12

нетривиальное создание объектов

Создание объектов

До настоящего момента мы создавали объекты вручную. Для каждого объекта использовался **объектный литерал,** который задавал все без исключения свойства. Для небольших программ это допустимо, но для серьезного кода потребуется что-то получше, а именно **конструкторы объектов**. Конструкторы упрощают создание объектов, причем вы можете создавать объекты по единому **шаблону** — иначе говоря, конструкторы позволяют создавать серии объектов, обладающих одинаковыми свойствами и содержащих одинаковые методы. Код, написанный с использованием конструкторов, получается гораздо более **компактным** и снижает риск ошибок при создании большого количества объектов. Уверяем, что после изучения этой главы вы будете пользоваться конструкторами так, словно занимались этим всю сознательную жизнь.

Создание объектов с использованием объектных литералов 540
О сходстве и различии между объектами 541
Конструкторы 543
Как создать конструктор 544
Как использовать конструктор 545
Как работают конструкторы 546
В конструкторы также можно добавить методы 548
Опасная зона 551
Техника безопасности 551
Даешь массовое производство! 554
Тест-драйв на новых машинах 556
Не спешите расставаться с объектными литералами 557
Преобразование аргументов в объектный литерал 558
Преобразование конструктора Car 559
Экземпляры 561
Даже сконструированные объекты могут содержать независимые свойства 564
Конструкторы в реальном мире 566
Объект Array 567
Другие встроенные объекты 569

13 использование прототипов

Сильные объекты

Научиться создавать объекты — только начало. Пришло время «накачать мышцы» — изучить расширенные средства определения **отношений** между объектами и организовать **совместное использование кода**. Кроме того, нам понадобятся механизмы расширения существующих объектов. Иначе говоря, нам нужно расширить свой инструментарий работы с объектами. В этой главе вы увидите, что в JavaScript реализована достаточно мощная **объектная модель**, но она немного отличается от модели традиционных объектно-ориентированных языков. Вместо типичных объектно-ориентированных систем на базе классов JavaScript использует модель **прототипов** — объектов, способных наследовать и расширять поведение других объектов. Какая в этом польза для вас? Вскоре узнаете. Итак, за дело...

Object
toString()
hasOwnProperty()
// and more

Dog Prototype
species: "Canine"
bark()
run()
wag()

ShowDog Prototype
league: “Webville”
stack()
bait()
gait()
groom()

ShowDog
name: “Scotty”
breed: “Scottish Terrier”
weight: 15
handler: “Cookie”

Представление объектов на диаграммах 581

Снова о конструкторах: код используется повторно, но насколько эффективно? 582

Дублирование методов действительно создает проблемы? 584

Что такое «прототип»? 585

Наследование через прототип 586

Как работает наследование 587

Переопределение прототипа 589

Как получить прототип 591

Как создать прототип 592

Переопределение унаследованного метода 594

О динамических прототипах 598

Более интересная реализация метода sit 600

Еще раз: как работает свойство sitting 601

С чего начать проектирование объектов 605

Создание цепочки прототипов 607

Как работает наследование в цепочке прототипов 608

Анализ результатов 617

Наводим чистоту 618

Еще немного усилий 619

Вызов Dog.call шаг за шагом 620

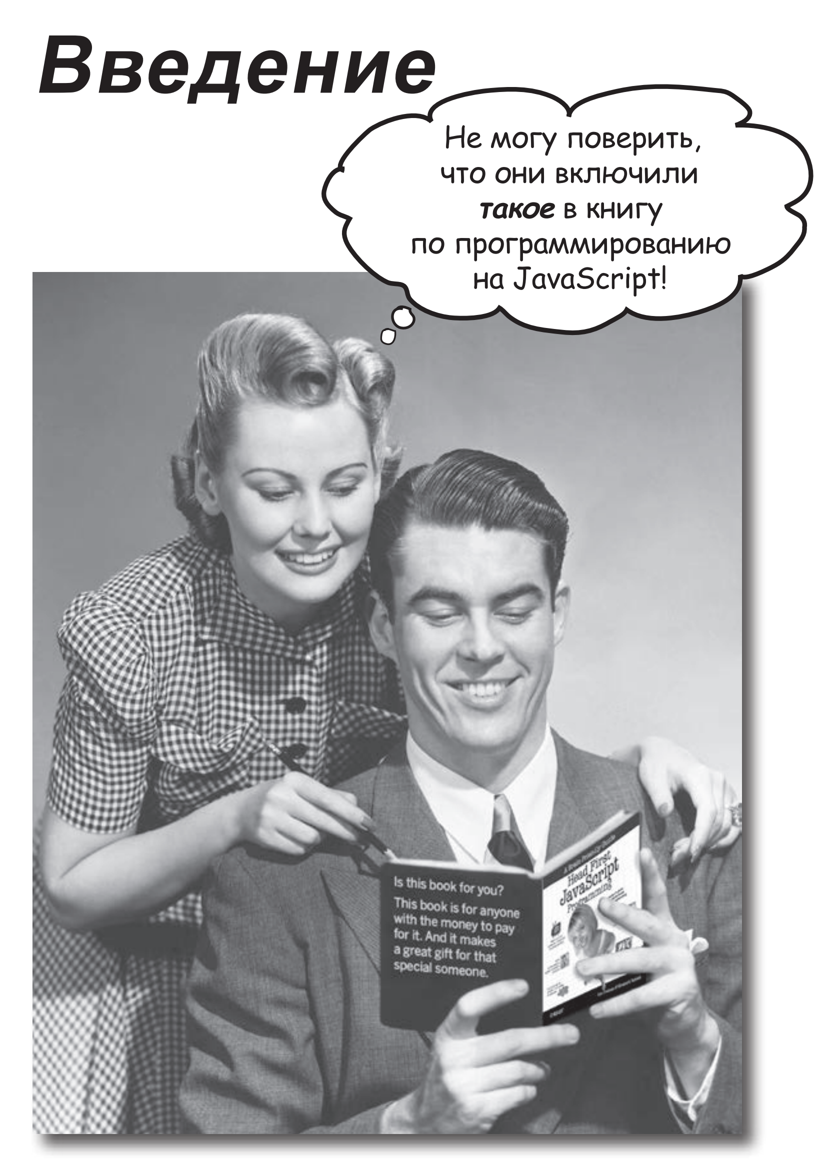
Применяем наследование с пользой... расширяя встроенный объект 626

Теория великого объединения ~~Всего~~ JavaScript 628

Объекты для лучшей жизни 628

Собираем все вместе 629

Введение



В этом разделе мы ответим на насущный вопрос: «Так почему они включили ТАКОЕ в книгу по программированию на JavaScript?»

Для кого написана эта книга?

Если вы ответите «да» на все следующие вопросы:

Под «современным браузером» мы понимаем обновленную версию Safari, Chrome, Firefox или IE версии 9 и выше.

1. У вас есть доступ к компьютеру с **современным браузером** и **текстовым редактором**?

2. Вы хотите **изучить, запомнить, понять и научиться программировать на JavaScript**, используя передовые приемы разработки и самые современные стандарты?

3. Вы предпочитаете **оживленную беседу сухим, скучным академическим лекциям?**

...то эта книга для вас.

[Заметка от отдела продаж: вообще-то эта книга для любого, у кого есть деньги.]

Кому эта книга не подойдет?

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

1. **Вы абсолютно не разбираетесь в веб-программировании?**

   Впервые слышите о HTML и CSS? В таком случае лучше начать с книги *Изучаем HTML, XHTML и CSS* — с ее помощью вы научитесь создавать веб-страницы, прежде чем браться за JavaScript.

2. Вы уже опытный веб-разработчик и ищете ***справочник*?**

3. Вы **боитесь попробовать что-нибудь новое**? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Считаете, что книга, в которой объекты JavaScript изображены в виде человечков, серьезной быть не может?

...эта книга не для вас.

Мы знаем, о чем Вы думаете

«Разве серьезные книги по программированию такие?»

«И почему здесь столько рисунков?»

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

И мы знаем, о чем думает Ваш мозг

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

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

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

Активизируются нейроны. Вспыхивают эмоции. Происходят химические реакции. И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библиотеке — в теплом, уютном месте, где тигры не водятся. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему, на которую вам выделили неделю... максимум десять дней. И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров, например. Или на то, что к огню лучше не прикасаться. Или что на лыжах не стоит кататься в футболке и шортах.

Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я хочу запомнить то, что здесь написано».

Эта книга для тех, кто хочет учиться

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

В отличие от других языков, код JavaScript передается непосредственно браузеру.

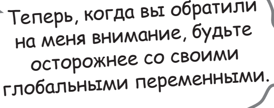
Основные принципы серии Head First:

Наглядность. Графика запоминается лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89% по данным исследований). Кроме того, материал становится более понятным. **Текст размещается на рисунках, к которым он относится**.

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

Я считаю, что код JavaScript следует размещать в элементе <head>.

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

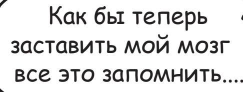


Не торопитесь! Это отразится на быстродействии и времени загрузки страницы!

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

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

Метапознание: наука о мышлении

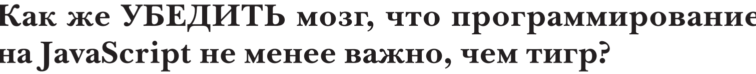


Если вы действительно хотите быстрее и глубже усваивать новые знания — задумайтесь над тем, как вы думаете. Учитесь учиться.

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

Но раз вы читаете эту книгу, то, вероятно, вы хотите научиться писать программы на JavaScript, и по возможности быстрее. Вы хотите *запомнить* прочитанное, а для этого абсолютно необходимо сначала *понять* прочитанное.

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



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

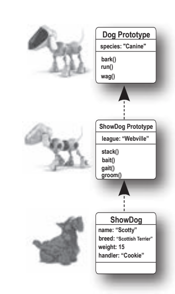
Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию можно запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и то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

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

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

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ:



Мы использовали ***рисунки***, потому что мозг лучше воспринимает графику, чем текст. С точки зрения мозга рисунок стоит 1024 слова. А когда текст комбинируется с графикой, мы внедряем текст прямо в рисунки, так мозг работает эффективнее.

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

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

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

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

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

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

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

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

В историях, примерах, на картинках присутствуют ***люди*** — потому что вы тоже *человек*. И ваш мозг обращает на людей больше внимания, чем на неодушевленные *предметы*.

Мы используем принцип ***80/20***. Если вы собираетесь стать крутым разработчиком JavaScript, только этой книгой вы не обойдетесь. В ней мы не пытаемся рассказывать обо *всем* — только о том, что действительно *необходимо*.

Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться

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

Вырежьте и прикрепите на холодильник.

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

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

2. **Выполняйте упражнения, делайте заметки.**

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

3. **Читайте врезки.**

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

4. **Не читайте других книг, отложив эту перед сном.**

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

5. **Пейте воду. И побольше.**

   Мозг лучше всего работает в условиях высокой влажности. Дегидратация (может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.

6. **Говорите вслух.**

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

7. **Прислушивайтесь к своему мозгу.**

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

8. ***Чувствуйте*!**

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

9. ***Творите*!**

   Попробуйте применить новые знания в своей повседневной работе. Просто сделайте хоть что-нибудь, чтобы приобрести практический опыт за рамками упражнений. Все, что для этого нужно, — это карандаш и подходящая задача... Задача, в которой использование JavaScript может принести пользу.

10. ***Спите.***

    Чтобы научиться программировать, вы должны создать множество новых связей в мозгу. Спите чаще, это полезно.

Примите к сведению

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

Мы описываем ХОРОШИЕ стороны JavaScript и предупреждаем о ПЛОХИХ.

Язык программирования JavaScript не был рожден в академических кругах и не проходил стадию просвещенной экспертной оценки. Он пришел в этот мир из необходимости и вырос в суровой среде ранних браузеров. Предупреждаем: у JavaScript есть масса достоинств, но есть и другие, не столь замечательные стороны. Но в целом JavaScript — прекрасный язык для тех, кто умеет разумно пользоваться им.

В этой книге мы научим вас эффективно использовать хорошие стороны, а также укажем на возможные ловушки и посоветуем, как их лучше обойти.

Мы не пытаемся во всех подробностях описать все аспекты языка.

JavaScript — достаточно обширная тема. Не стоит рассматривать эту книгу как справочник; это учебник, в котором не излагается все, что только можно узнать о JavaScript. Мы хотим научить вас азам JavaScript, чтобы вы могли взять любой старый справочник и сделать на JavaScript все, что захотите.

Эта книга учит использованию JavaScript в браузерах.

Браузер — не только наиболее типичная среда для выполнения кода JavaScript, но и самая удобная (у каждого есть компьютер с текстовым редактором и браузером, а для начала работы на JavaScript ничего большего и не понадобится). Выполнение JavaScript в браузере также означает мгновенный отклик; чтобы увидеть, как работает написанный вами код, достаточно перезагрузить веб-страницу.

Книга поощряет написание хорошо структурированного, удобочитаемого кода.

Код должен быть написан так, чтобы он хорошо читался и был понятен другим людям, а также работал в браузерах, которые выйдут в следующем году. Он должен быть по возможности простым и прямолинейным, чтобы вы справились со своей задачей и могли перейти к другим делам. В этой книге мы научим вас писать простой, хорошо структурированный код, способный адаптироваться к будущим изменениям, — код, которым вы можете гордиться, убрать в рамку и повесить на стену.

Мы рекомендуем использовать разные браузеры.

Мы учим вас писать JavaScript, основанный на стандартах, и все же вы, скорее всего, столкнетесь с незначительными различиями в интерпретации JavaScript разными браузерами. Хотя мы проследили за тем, чтобы приводимый код работал во всех современных браузерах, и даже покажем пару приемов, обеспечивающих поддержку кода этими браузерами, советуем выбрать пару браузеров и тестировать в них код JavaScript. Так вы научитесь видеть различия между браузерами и создавать код JavaScript, который будет хорошо работать в разных браузерах со стабильными результатами.

Программирование — дело серьезное. Вам придется работать, иногда весьма напряженно.

Если у вас уже есть опыт практического программирования, то вы знаете, о чем мы говорим. Если вы взялись за эту книгу после *Изучаем HTML, XHTML и CSS*, то учтите, что код, который вам придется писать здесь, выглядит... немного иначе. Программирование требует особого склада ума. Оно логично, порой крайне абстрактно и алгоритмично. Не волнуйтесь, мы постараемся сделать процесс изучения по возможности приятным для вашего мозга. Двигайтесь постепенно, полноценно питайтесь и побольше спите — и новые концепции уложатся у вас голове.

Упражнения ОБЯЗАТЕЛЬНЫ.

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

Повторение применяется намеренно.

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

Примеры были сделаны по возможности компактными.

Нашим читателям не нравится просматривать 200 строк кода в примерах, чтобы найти две действительно важные строки. Большинство примеров книги приводится в минимально возможном контексте, чтобы та часть, которую вы изучаете, была простой и наглядной. Не ждите, что все примеры будут хорошо отлажены или дописаны до конца — они сделаны в учебных целях и не всегда обладают полноценной функциональностью. Все файлы с примерами доступны в Интернете. Вы найдете их по адресу `http://wickedlysmart.com/hfjs`.

Упражнения «Игры разума» не имеют ответов.

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

Мы часто приводим только код без разметки.

После первой пары глав мы часто приводим только код JavaScript, предполагая, что он упакован в разметку HTML. Ниже приведена простая страница HTML, которая может использоваться с большей частью кода книги. Если где-то должна использоваться другая разметка HTML, мы вам об этом скажем:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Your HTML Page</title>
        <script>

                        ← Ваш код JavaScript обычно размещается здесь.

        </script>
    </head>
    <body>
                   ← Все содержимое веб-страницы размещается здесь.
    </body>
</html>
```

↑ Не беспокойтесь; в начале книги мы все объясним.

Примеры кода, поддержка и обсуждения

Все необходимые сопроводительные материалы доступны на сайте `http://wickedlysmart.com/hfjs`. Здесь размещены примеры кода и дополнительные материалы, включая видеоролики.

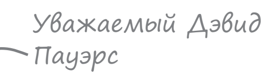
Научные редакторы

Спасибо всем участникам нашей бесподобной группы рецензирования!

Эта книга прошла более тщательное рецензирование, чем любая из наших предыдущих книг. Более 270 человек, присоединившихся к нашей программе WickedlySmart Insiders, участвовали в вычитке и подвергали оперативному критическому анализу материалы книги в процессе написания. Система сработала лучше, чем мы ожидали, и повлияла практически на все аспекты книги. Выражаем свою искреннюю признательность всем участникам; благодаря вам книга стала намного лучше, чем могла бы быть.

Научные редакторы, изображенные выше, представили особенно ценную информацию и внесли значительный вклад в работу над книгой. Кроме них нам помогали: **Галина Орлова, Дж. Патрик Келли, Клаус-Петер Каль, Роб Клири, Ребекка Данн-Кран**, Олаф Шенрих, Джим Купек, Мэтью М. Ханрахан, Рассел Аллин-Виллемс, Кристин Дж. Уилсон, Луи-Филипп Бретон, Тимо Глейзер, Шармен Грей, Ли Бекхэм, Майкл Мерфи, Дэйв Янг, Дон Смоллидж, Алан Русяк, Эрик Р. Лисински, Брент Фазекас, Сью Старр, Эрик (Оранжевые Штаны) Джонсон, Джесс Палмер, Манабу Каваками, Алан Макайвор, Алекс Келли, Ивонна Биксел Труон, Остин Трауп, Тим Уильямс, Дж. Альберт Боуден II, Род Шелтон, Нэнси ДеХейвен Холл, Сью Макги, Франсиско Дебс, Мириам Беркленд, Кристин Грекко, Эльхаджи Барри, Атанасиос Валсамакис, Питер Кейси, Дастин Уоллем и Робб Керри.

Благодарности*



Мы также исключительно благодарны нашему уважаемому научному редактору **Дэвиду Пауэрсу**. По правде говоря, мы вообще не пишем без участия Дэвида — он слишком часто спасал нас от возможных неприятностей. Дэвид помогает нам (а вернее, заставляет нас) сделать книгу более содержательной и технически точной, а его вторая профессия комика-любителя оказывается полезной, когда мы пишем несерьезные части книги. Спасибо еще раз, Дэвид, — ты настоящий профессионал, и мы спокойнее спим ночами, зная, что книга прошла твой технический контроль.

Пусть эта улыбка не вводит вас в заблуждение, этот человек — настоящий профессионал.

К издательству O'Reilly:

Огромное, гигантское спасибо нашему редактору **Меган Бланшетт**, которая расчистила путь для этой книги, устранила все препятствия, терпеливо ждала и жертвовала семейным досугом ради ее завершения. Кроме того, она помогает нам сохранить рассудок в наших отношениях с O'Reilly (а O'Reilly — в их отношениях с нами). Мы обожаем тебя и не дождемся следующего совместного проекта!

Еще один дружеский привет нашему главному редактору **Майку Хендриксону**, который активно продвигал эту книгу с самого начала. Спасибо, Майк; ни одна наша книга не вышла бы без твоего участия. Ты был нашим предводителем больше десяти лет, и мы любим тебя!

Майк Хендриксон

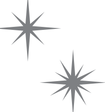
*Большое количество благодарностей объясняется просто: мы проверяем теорию, согласно которой каждый упомянутый в разделе благодарностей купит хотя бы один экземпляр книги (а может, и больше) для родственников и знакомых. Если вы хотите, чтобы мы поблагодарили вас в нашей *следующей* книге, и у вас большая семья — пишите.

Участникам из O'Reilly:

Мы искренне благодарим всю группу O'Reilly: **Мелани Ярбро**, **Боба Пфалера** и **Дэна Фоксмита**, которые придали форму этой книге; **Эду Стивенсону, Хьюджетт Бэррие**р и **Лесли Крэнделлу,** которые руководили маркетингом, — мы оценили их нестандартный подход к делу. Спасибо **Элли Волкхакузен**, **Рэнди Камеру** и **Карен Монтгомери** за стильный дизайн обложки, который продолжает служить нам. Как обычно, спасибо **Рэчел Монахан** за бескомпромиссное редактирование (и за то, как она нас подбадривала), а **Берту Бейтсу** — за исключительно полезную обратную связь.

1 первое знакомство с javascript

В незнакомых водах



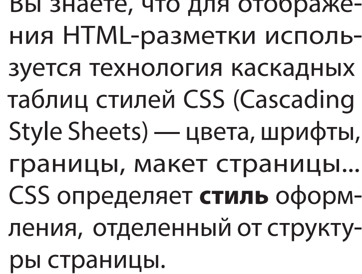
JavaScript открывает фантастические возможности. JavaScript, **основной язык программирования** Всемирной паутины, позволяет определять расширенное поведение в веб-страницах. Забудьте о сухих, скучных, статичных страницах, которые просто занимают место на экране, — с JavaScript вы будете взаимодействовать с пользователями, реагировать на события, получать и использовать данные из Интернета, выводить графику... и многое, многое другое. При хорошем знании JavaScript вы сможете даже программировать **совершенно новое** поведение на своих страницах.

И не сомневайтесь — ваши знания будут востребованы. Сейчас JavaScript не только является одним из самых **популярных языков** программирования, но и **поддерживается** всеми современными (и многими несовременными) браузерами; более того, появились встроенные реализации JavaScript, существующие отдельно от браузеров. А впрочем, хватит разговоров. Пора браться за дело!

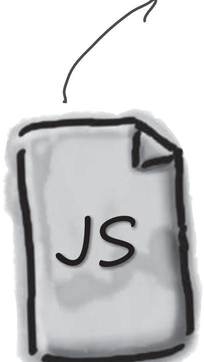
Как работает JavaScript

Вы освоили создание структуры, контента, макета и стиля веб-страниц. Не пора ли добавить к ним поведение? В наше время страница, на которую можно только смотреть, никому не интересна. Хорошие страницы должны быть динамическими и интерактивными, и они должны по-новому взаимодействовать с пользователями. Именно для этого и нужен JavaScript. Для начала давайте посмотрим, какое место JavaScript занимает в экосистеме веб-страниц:

Вы уже знаете, что язык HTML (Hypertext Markup Language) используется для определения **контента** страниц, а также их **структуры**: абзацев, заголовков и разделов.



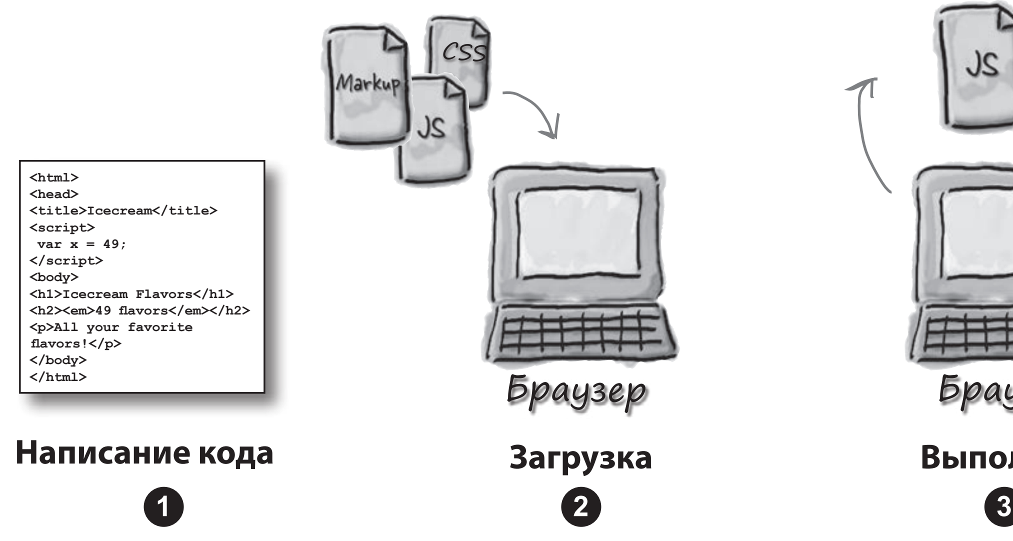
Вы знаете, что для отображения HTML-разметки используется технология каскадных таблиц стилей CSS (Cascading Style Sheets) — цвета, шрифты, границы, макет страницы... CSS определяет **стиль** оформления, отделенный от структуры страницы.



Итак, знакомьтесь: JavaScript, родственник HTML и CSS. JavaScript умеет определять **поведение** веб-страниц. Нужно среагировать, когда пользователь нажимает кнопку «Распродажа заканчивается через 30 секунд!»? Быстро проверить содержимое текстового поля с важными данными? Загрузить сообщения из Твиттера и вывести их на странице? А может, хотите поиграть? На помощь приходит JavaScript. Вы сможете добавить к странице программный код, который будет выполнять вычисления, реагировать на события, рисовать, передавать данные, сигнализировать, обновлять информацию, изменять... Словом, любые динамические операции — всё это умеет делать JavaScript.

Как пишется код JavaScript

JavaScript занимает особое место в мире программирования. Как появляется типичная классическая программа? Вы пишете код, компилируете его, проводите компоновку и устанавливаете на компьютер. Язык JavaScript куда более гибок и динамичен. Программист включает код JavaScript прямо в страницу, а потом загружает ее в браузер. Далее браузер сам сделает все необходимое для выполнения написанного кода. Давайте повнимательнее разберемся с тем, как работает эта схема:



Написание кода

1

Страница создается как обычно: с контентом HTML и стилевым оформлением CSS. На страницу добавляется код JavaScript. Как вы вскоре увидите, по аналогии с HTML и CSS все компоненты можно разместить в одном файле или же выделить код JavaScript в отдельный файл, который включается в страницу.

Вскоре мы разберемся, какой способ лучше...

Загрузка

2

Откройте страницу в браузере. Обнаружив в странице код JavaScript, браузер сразу начинает разбирать его и готовить к выполнению. Как и в случае с HTML и CSS, если браузер обнаруживает ошибки в JavaScript, он старается продолжить чтение JavaScript, HTML и CSS. Браузер старается избежать ситуации, в которой пользователь не сможет увидеть запрошенную страницу.

На будущее: браузер строит «объектную модель» страницы HTML, которая может использоваться кодом JavaScript. Просто запомните этот факт, мы еще к нему вернемся...

Выполнение

3

Браузер начинает выполнять код сразу, как только встречает его, и продолжает выполнять на всем протяжении жизненного цикла страницы. В отличие от ранних версий, современный JavaScript отличается высокой эффективностью, а благодаря изощренным методам компиляции код почти не уступает по скорости традиционным языкам программирования.

Как включить код JavaScript в страницу

Начнем с начала: чтобы продвинуться в изучении JavaScript, необходимо знать, как включить код в страницу. Как же это делается? Конечно, при помощи элемента `<script>`!

Давайте возьмем скучную старую веб-страницу и определим для нее динамическое поведение в элементе `<script>`. Пока не нужно задумываться над смыслом того, что мы включаем в элемент `<script>`, — сейчас важнее понять, как вообще работает JavaScript.

Стандартный заголовок HTML5 doctype, элементы <html> и <head>.

Элемент <body> выглядит вполне традиционно.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Just a Generic Page</title>
    <script>
      setTimeout(wakeUpUser, 5000);
      function wakeUpUser() {
         alert("Are you going to stare at this boring page forever?");
      }
    </script>
  </head>
  <body>
       <h1>Just a generic heading</h1>
       <p>Not a lot to read about here. I'm just an obligatory paragraph living in
an example in a JavaScript book. I'm looking for something to make my life more
exciting.</p>
  </body>
</html>
```

В раздел <head> страницы добавляется элемент script.

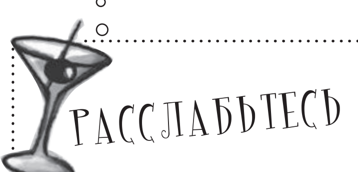
В нем записывается фрагмент кода JavaScript.

Еще раз: сейчас для нас не важно, как работает этот код. И все же... Посмотрите на него и предположите, что происходит в каждой из строк.

Тест-драйв

Введите код страницы и сохраните его в файле с именем “behavior.html”. Теперь загрузите страницу в браузере (перетащите файл в окно браузера или воспользуйтесь командой Файл > Открыть). Что делает наш код?

Подсказка: чтобы это понять, нужно подождать пять секунд.



Только не волнуйтесь. Никто не ждет, что вы сразу начнете понимать JavaScript так, словно знаете его с детства. Сейчас достаточно представлять, на что похож JavaScript.

Расслабляться тоже не стоит: нужно, чтобы ваш мозг заработал в полную силу. Помните код с предыдущей страницы? Давайте попробуем предположить, что происходит в каждой строке:

```
setTimeout(wakeUpUser, 5000);
function wakeUpUser() {
   alert("Are you going to stare at this boring page forever?");
}
```

Вероятно, мы создаем код, который можно будет использовать в других местах, и присваиваем ему имя «wakeUpUser»?

Здесь каким-то образом отсчитывается пять секунд? Подсказка: 1000 миллисекунд = 1 секунда.

Тут все понятно: выводится сообщение для пользователя.

Часто задаваемые вопросы

В: Я слышал, что JavaScript называют «игрушечным языком». Это правда?

О: На первых порах JavaScript не отличался мощью, но затем его значимость возросла, и на расширение возможностей JavaScript были направлены значительные ресурсы (в том числе и умы лучших специалистов). Но знаете что? Еще до того, как JavaScript стал таким быстрым, он был замечательным. И как вы вскоре убедитесь, с ним можно сделать много интересного.

В: Язык JavaScript как-то связан с Java?

О: Только по названию. Язык JavaScript создавался на пике популярности Java, и разработчики JavaScript удачно воспользовались этим обстоятельством. Оба языка заимствуют некоторые элементы синтаксиса из языков семейства C, но в остальном они имеют мало общего.

В: Значит, JavaScript — лучший способ создания динамических страниц? А как насчет решений на базе Flash?

О: Было время, когда технология Flash считалась предпочтительной для создания интерактивных и более динамичных веб-страниц, но с тех пор отрасль явно начала переходить на стандарт HTML5 с JavaScript. А в HTML5 JavaScript теперь является стандартным языком сценариев для веб-программирования. Сейчас значительные силы и средства тратятся на повышение скорости и эффективности JavaScript, а также на создание JavaScript API, расширяющих функциональность браузера.

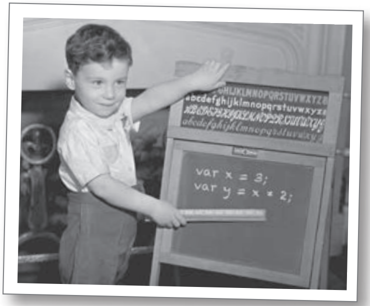
В: Мой друг работает с JavaScript в Photoshop... по крайней мере он так говорит. Это возможно?

О: Да, JavaScript — универсальный язык сценарного программирования и проникает во многие приложения, от графических редакторов до музыкальных программ, и даже в область серверного программирования.

В: Вы говорите, что многие другие языки компилируются. Что это такое и почему этого нет в JavaScript?

О: В традиционных языках программирования — C, C++ или Java — код компилируется перед выполнением. В процессе компиляции код преобразуется в представление, понятное для машины (и обычно оптимизированное по скорости выполнения). Сценарные языки интерпретируются, то есть браузер выполняет каждую строку JavaScript сразу же, как только встретит ее. Для сценарных языков производительность во время выполнения не так важна; они в большей степени ориентированы на построение прототипов, интерактивное программирование с максимальной гибкостью. С ранними версиями JavaScript так и было, по этой причине еще много лет скорость выполнения кода была довольно посредственной. Однако существовал промежуточный вариант: интерпретируемый язык, который компилируется «на ходу». Именно он был выбран разработчиками браузеров для современных реализаций JavaScript. По сути, в JavaScript вы пользуетесь всеми удобствами сценарных языков в сочетании с быстродействием компилируемого языка. Кстати говоря, в этой книге часто встречаются слова «интерпретировать», «вычислять» и «выполнять». Они могут различаться по смыслу в разных контекстах, но для наших целей фактически эквивалентны.

JavaScript, ты проделал длинный путь, детка...



JavaScript 1.0

Возможно, вы не помните Netscape, но он был первым *настоящим* разработчиком браузеров. В середине 1990-х на рынке шла яростная борьба (особенно со стороны Microsoft), и добавление новых интересных возможностей в браузер было исключительно важным делом.

Для этого компания Netscape создала язык сценариев, который позволял любому желающему включить сценарный код в страницу. Так появился LiveScript. Скорее всего, вы никогда не слышали о LiveScript, потому что в то же время Sun Microsystems представила язык Java, и ее акции взлетели до заоблачного уровня. Почему бы не воспользоваться чужим успехом? Так LiveScript стал JavaScript. У этих языков нет ничего общего? Ну и что...

А что же Microsoft? Вскоре за Netscape она создала собственный язык сценариев, который назвали... JScript. Он был как-то подозрительно похож на JavaScript. Так начались войны браузеров.

JavaScript 1.3

Между 1996 и 2000 годом язык JavaScript продолжал развиваться. Netscape передала JavaScript для стандартизации; так родился ECMAScript. Не слышали про ECMAScript? Просто знайте, что он является стандартным определением языка для всех реализаций JavaScript (в браузерах и без них).

В это время разработчики продолжали бороться с JavaScript (войны браузеров были в самом разгаре), хотя использование JavaScript становилось все более распространенным. И хотя тонкие различия между JavaScript и JScript продолжали отравлять жизнь разработчиков, два языка со временем все больше походили друг на друга.

JavaScript все еще пользовался репутацией «языка для дилетантов», но скоро все изменилось...

JavaScript 1.8.5

Наконец, JavaScript достиг зрелости и заслужил признание профессиональных разработчиков! Хотя кто-то скажет, что все дело в появлении надежного стандарта (такого, как ECMAScript 5), который сейчас реализован во всех современных браузерах, на самом деле выходу JavaScript на профессиональную арену в немалой степени способствовала компания Google. В 2005 году Google Maps показал всему миру, на что способен JavaScript при создании динамических веб-страниц.

Лучшие умы из области программирования трудились над совершенствованием интерпретаторов JavaScript и повышением скорости выполнения кода. Синтаксис на протяжении эволюции JavaScript в целом изменился незначительно. Несмотря на всю суету, сопровождавшую его рождение, JavaScript оказался мощным и выразительным языком.

1995 — 2000 — 2012

Возьми в руку карандаш

Посмотрите, как легко пишется код JavaScript

Вы еще не знаете JavaScript, но наверняка сможете хотя бы в общих чертах предположить, как работает его код. Взгляните на каждую строку и попробуйте догадаться, что она делает. Запишите свои ответы. Мы приводим один ответ, чтобы вам было проще взяться за дело. А если задача окажется непосильной, ответы приведены на следующей странице.

```
var price = 28.99;
var discount = 10;
var total =
      price - (price * (discount / 100));
if (total > 25) {
   freeShipping();
}

var count = 10;
while (count > 0) {
   juggle();
   count = count - 1;
}

var dog = {name: "Rover", weight: 35};
if (dog.weight > 30) {
    alert("WOOF WOOF");
} else {
    alert("woof woof");
}

var circleRadius = 20;
var circleArea =
   Math.PI * (circleRadius * circleRadius);
```

Создать переменную price и присвоить ей значение 28.99.

Возьми в руку карандаш Решение

Посмотрите, как легко пишется код JavaScript

Вы еще не знаете JavaScript, но наверняка сможете хотя бы в общих чертах предположить, как работает его код. Взгляните на каждую строку и попробуйте догадаться, что она делает. Запишите свои ответы. Мы привели один ответ, чтобы вам было проще взяться за дело. Итак, ответы.

Код	Ответ
`var price = 28.99;`	*Создать переменную price и присвоить ей значение 28.99.*
`var discount = 10;`	*Создать переменную discount и присвоить ей значение 10.*
`var total = price - (price * (discount / 100));`	*Вычислить цену со скидкой и присвоить переменной total.*
`if (total > 25) {`	*Сравнить переменную total и 25. Если переменная больше...*
`freeShipping();`	*...выполнить фрагмент кода freeShipping.*
`}`	*Конец команды if*
`var count = 10;`	*Создать переменную count и присвоить ей значение 10.*
`while (count > 0) {`	*Пока значение count остается больше 0...*
`juggle();`	*...что-то сделать, а потом...*
`count = count - 1;`	*...уменьшить count на 1.*
`}`	*Конец цикла while*
`var dog = {name: "Rover", weight: 35};`	*Создать переменную dog с атрибутами name и weight.*
`if (dog.weight > 30) {`	*Если атрибут weight больше 30...*
`alert("WOOF WOOF");`	*...вывести на веб-странице сообщение "WOOF WOOF»*
`} else {`	*В противном случае...*
`alert("woof woof");`	*...вывести на веб-странице сообщение «woof woof»*
`}`	*Конец команды if/else*
`var circleRadius = 20;`	*Создать переменную circleRadius и присвоить ей значение 20.*
`var circleArea =`	*Создать переменную с именем circleArea...*
`Math.PI * (circleRadius * circleRadius);`	*...и присвоить ей результат выражения (1256.6370614359173)*

И это правда.

Страницы, построенные на базе HTML и CSS, могут хорошо выглядеть. Но изучив JavaScript, вы сможете строить принципиально новые разновидности страниц. Более того, их будет правильнее рассматривать не как обычные страницы, а как приложения!

А заодно и подзаработать!

Что вы говорите? «Конечно, я это отлично знаю, иначе зачем бы мне читать эту книгу?» Вообще-то мы хотели воспользоваться возможностью и немного поговорить об изучении JavaScript. Если у вас уже имеется опыт работы на каком-нибудь языке программирования или языке сценариев, то вы примерно представляете, что вас ждет. Если до сегодняшнего дня вы ограничивались HTML и CSS, знайте: при изучении языка программирования вас ждет нечто принципиально новое.

В HTML и CSS в основном выполняются декларативные операции. Допустим, вы объявляете, что некоторый текст абзаца или все элементы класса «sale» должны быть окрашены в красный цвет. JavaScript добавляет в страницу новое поведение, а для этого необходимо описать вычисления. Вам понадобятся средства для описания разнообразных операций: «вычислить счет игрока, сложив количество баллов», или «повторить следующее действие десять раз», или «когда пользователь нажмет эту кнопку, воспроизвести такой-то звуковой сигнал», или даже «зайти в Твиттер, получить последнее сообщение и разместить его на этой странице».

Язык, необходимый для решения таких задач, сильно отличается от HTML и CSS. Давайте посмотрим, чем именно...

Как создаются команды

При создании контента HTML вы обычно размечаете текст, определяя его структуру; для этого в текст добавляются элементы, атрибуты и значения:

```
<h1 class="drink">Mocha Caffe Latte</h1>
<p>Espresso, steamed milk and chocolate syrup,
just the way you like it.</p>
```

При работе с HTML мы размечаем текст для определения структуры: например, «Так, здесь у нас будет крупный заголовок, а за ним следует абзац обычного текста».

С CSS дело обстоит немного иначе. Разработчик пишет набор **правил**; каждое правило выбирает элементы страницы, а затем задает для этих элементов набор стилей:

```
h1.drink {
    color: brown;
}
p {
    font-family: sans-serif;
}
```

Для CSS пишутся правила с селекторами (например, h1.drink и p), определяющими, к каким частям разметки HTML применяется данный стиль.

Допустим, все заголовки drink выводятся коричневым цветом...

...А абзацы выводятся шрифтом без засечек.

Код JavaScript состоит из **команд**. Каждая команда описывает небольшую часть выполняемой операции, а весь набор команд определяет поведение страницы:

Набор команд.

Каждая команда выполняет небольшую часть работы, например объявляет переменные, в которых будут храниться используемые значения.

```
var age = 25;
var name = "Owen";

if (age > 14) {
    alert("Sorry this page is for kids only!");
} else {
    alert("Welcome " + name + "!");
}
```

Мы создаем переменную для хранения возраста (25). Нам понадобится и переменная для имени "Owen".

Значение переменной может использоваться для принятия решений. Возраст пользователя больше 14?

Если больше — сообщаем, что пользователь слишком стар для этой страницы.

А если нет — приветствуем пользователя по имени (впрочем, в нашем примере Оуэну 25 лет, так что сообщение не выводится).

Переменные и значения

Вероятно, вы заметили, что в программах JavaScript обычно используются переменные. Переменные предназначены для хранения значений. Каких? Рассмотрим несколько примеров:

```
var winners = 2;
```

Команда объявляет переменную с именем winners и присваивает ей числовое значение 2.

```
var name = "Duke";
```

Переменной присваивается последовательность символов (такая последовательность называется строкой).

```
var isEligible = false;
```

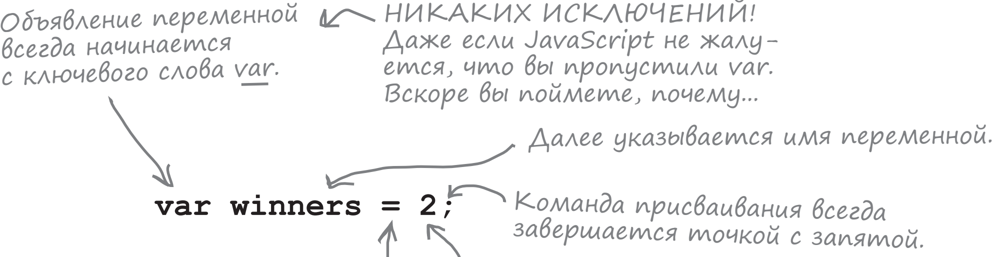
Переменной isEligible присваивается значение false. Переменные, принимающие два значения, true и false (истина/ложь), называются логическими (или булевскими).

В честь математика Джорджа Буля.

2
winners
"Duke"
name
false
isEligible

Обратите внимание: логические значения не заключаются в кавычки.

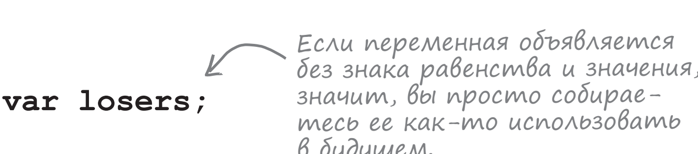
Кроме чисел, строк и логических значений переменные могут хранить и другие данные. Вскоре мы доберемся и до них, но независимо от вида данных все переменные создаются по одним правилам. Присмотримся повнимательнее к объявлению переменной:



Наконец, при желании можно указать начальное значение переменной; для этого ставится знак равенства, за которым следует значение.

Никакого значения? И как теперь жить?! Это просто унизительно.

Мы говорим «при желании», потому что, строго говоря, вы можете создать переменную без начального значения и присвоить его позднее. Для этого достаточно убрать из команды присваивание:



losers

Осторожно, ключевые слова!

У переменной есть имя и у переменной есть значение.

Также вы знаете, что в переменных могут храниться числа, строки и логические значения (и не только). Но как выбрать имя переменной? Подойдет любое имя? Нет, но подобрать допустимое имя несложно. Достаточно убедиться в том, что имя переменной не нарушает два простых правила:

1. **Имя переменной должно начинаться с буквы, подчеркивания или знака доллара.**
2. **Потом могут следовать буквы, цифры, подчеркивания и знаки доллара — в любом количестве.**

Да, и еще один момент: не стоит путать JavaScript и использовать в качестве имен переменных встроенные *ключевые слова*, такие как **var**, **function** или **false**. Эти имена тоже выпадают из списка. В этой книге мы еще рассмотрим некоторые ключевые слова и разберемся, что они означают, а пока ограничимся кратким списком:

break	delete	for	let	super	void
case	do	function	new	switch	while
catch	else	if	package	this	with
class	enum	implements	private	throw	yield
const	export	import	protected	true	
continue	extends	in	public	try	
debugger	false	instanceof	return	typeof	
default	finally	interface	static	var	

Часто задаваемые вопросы

В: Что такое «ключевое слово»?

О: Ключевое слово — одно из зарезервированных слов языка JavaScript. JavaScript использует такие слова для собственных целей. Если вы начнете использовать их как имена своих переменных, то только собьете с толку свой браузер.

В: А если ключевое слово входит в имя переменной? Например, могу ли я создать переменную с именем ifOnly (то есть переменную, в имени которой присутствует ключевое слово if)?

В: Конечно, можете. Запрещены только точные совпадения. Желательно, чтобы ваш код был простым и понятным, поэтому использовать имена вида `elze` тоже нежелательно — их легко спутать с `else`.

В: В JavaScript учитывается регистр символов? Другими словами, myvariable и MyVariable — одно и то же?

О: Если вы работали с разметкой HTML, то, скорее всего, привыкли к тому, что регистр символов не учитывается; в конце концов, для браузера тег <head> ничем не отличается от <HEAD>. Но в JavaScript регистр символов учитывается в именах переменных, ключевых словах, именах функций... Короче, практически везде. Так что будьте внимательны с использованием верхнего и нижнего регистра!

WEBVILLE TIMES

Как избежать досадных ошибок с именами

Выбор имен переменных чрезвычайно широк, поэтому мы дадим несколько рекомендаций, которые спасут вас от многих неприятностей:

Выбирайте осмысленные имена.

Может, имена _m, $, r и foo что-то означают для вас, но окружающие вас не поймут. Имена angle, currentPressure и passedExam не только не забудутся со временем, но и сделают ваш код намного более понятным.

Используйте «горбатый регистр» в именах из нескольких слов.

Допустим, в какой-то момент вам нужно выбрать имя переменной, для двуглавого огнедышащего дракона. Как? Просто начинайте с прописной буквы каждое слово, кроме первого: twoHeadedDragonWithFire. «Горбатый регистр» очень популярен и обладает достаточной гибкостью для создания имени переменной с любой необходимой точностью. Существуют и другие схемы записи имен, но «горбатый регистр» стал самым распространенным (в том числе и за пределами JavaScript).

Используйте имена, начинающиеся с _ и $, только в особых случаях.

Имена переменных, начинающиеся с $, обычно резервируются для библиотек JavaScript. И хотя некоторые авторы используют имена переменных, начинающиеся с _, мы не рекомендуем так поступать, если только у вас нет для этого очень веских причин (когда будут, тогда сами поймете).

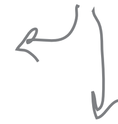
Будьте осторожны.

Будьте осторожны с выбором имен переменных. Позднее мы приведем еще несколько правил осторожного выбора имен, а пока просто запомните: выбирайте понятные имена, держитесь подальше от ключевых слов и всегда указывайте var при объявлении переменной.

О синтаксисе

- Каждая команда завершается символом “;”.
  `x = x + 1;`
- Однострочный комментарий начинается с двух косых черт (//). Комментарии только содержат информацию о коде для вас и других разработчиков. В программе они не выполняются.
  `// Это комментарий`
- Лишние пробелы разрешены (почти везде).
  `x     =     2233;`
- Строки должны заключаться в двойные кавычки (или одиночные, но выберите что-то одно — будьте последовательны).
  `"You rule!"`
  `'And so do you!'`
- Логические значения true и false записываются без кавычек.
  `rockin = true;`
- Переменным не обязательно присваивать значение при объявлении:
  `var width;`
- Язык JavaScript, в отличие от разметки HTML, учитывает регистр символов. Другими словами, Counter и counter — разные переменные.

СТАНЬ браузером



Ниже приведен код JavaScript, содержащий ошибки. Представьте себя на месте браузера и попробуйте отыскать ошибки в коде. А когда это будет сделано, загляните в ответы в конце главы и посмотрите, не упустили ли вы чего-нибудь.

A

```
// Test for jokes
var joke = "JavaScript walked into a bar....';
var toldJoke = "false";
var $punchline =
  "Better watch out for those semi-colons."
var %entage = 20;
var result

if (toldJoke == true) {
    Alert($punchline);
} else
    alert(joke);
}
```

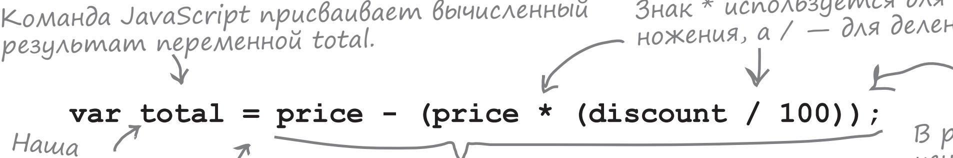
Пока не задумывайтесь над тем, что делает этот фрагмент JavaScript; просто постарайтесь отыскать ошибки в переменных и синтаксисе.

B

```
\\ Movie Night
var zip code = 98104;
var joe'sFavoriteMovie = Forbidden Planet;
var movieTicket$    =    9;

if (movieTicket$ >= 9) {
    alert("Too much!");
} else {
    alert("We're going to see " + joe'sFavoriteMovie);
}
```

Поаккуратнее с выражениями!

Чтобы выразить намерения на языке JavaScript, вам понадобятся выражения. Каждое выражение вычисляется, а результатом является значение. Мы уже встречались с выражениями в примерах кода. Давайте рассмотрим выражение в следующей команде:



В результате вычисления цена (price) уменьшается на размер скидки (discount), заданный в процентах от цены. Если переменная price содержит значение 10, а discount — 20, результат будет равен 8.

Если вы хоть раз посещали уроки математики, подводили баланс или платили налоги, наверняка числовые выражения покажутся вам знакомыми.

Существуют и строковые выражения:

```
"Dear " + "Reader" + ","
```

Эти строки «сцепляются» в новую строку «Dear Reader» (такая операция называется конкатенацией).

```
"super" + "cali" + youKnowTheRest
```

То же самое, но в выражение входит переменная, содержащая строку. При вычислении результата получается строка "supercalifrag ilisticexpialidocious". *

```
phoneNumber.substring(0,3)
```

Еще один пример выражения, возвращающего строку. О том, как оно работает, будет рассказано позднее, а пока достаточно сказать, что выражение возвращает код зоны из телефонного номера.

Результат вычисления также может быть истиной или ложью (true или false); данные выражения называются булевскими (или логическими). Определите, какой результат (true или false) получится при вычислении каждого из следующих выражений:

```
age < 14
```

Если возраст (age) меньше 14, то результат равен true; в противном случае он равен false. Команда проверяет возраст пользователя.

```
cost >= 3.99
```

Если цена товара больше или равна 3.99, выражение истинно. В противном случае оно ложно. Не пропустите распродажу!

```
animal == "bear"
```

Истинно, если в названии животного присутствует слово "bear".

Результат выражения может быть значением другого типа; мы еще вернемся к этой теме. А пока важно знать, что при вычислении любого выражения будет получен некий результат – число, строка или логическое выражение. Что же нам это дает?

* Предполагается, что переменная youKnowTheRest содержит строку "fragilisticexpialidocious".

Возьми в руку карандаш

Возьмите карандаш. Для каждого из приведенных выражений вычислите его значение и запишите в выделенном месте. Именно ЗАПИШИТЕ… забудьте слова мамы о том, что нельзя писать в книгах, и запишите ответ на странице! Свериться с ответами можно в конце главы.

Калькулятор пересчета температуры по Цельсию в Фаренгейты?

```
(9 / 5) * temp + 32
```

Что получится, если переменная temp равна 10? __________

Логическое выражение. Оператор == проверяет, равны ли два значения.

```
color == "orange"
```

Истинным или ложным будет это выражение, если переменная color содержит значение “pink”? __________

А если значение “orange”? __________

```
name + ", " + "you've won!"
```

Какое значение будет, если переменная name содержит строку “Martha”? __________________________________

Проверка: «первое значение больше второго?» Можно использовать оператор >= для условия «первое значение больше либо равно второму?»

```
yourLevel > 5
```

Что получится, если переменная yourLevel = 2? ________

Что получится, если переменная yourLevel = 5? ________

Что получится, если переменная yourLevel = 7? ________

```
(level * points) + bonus
```

Переменная level = 5, points = 30000, а bonus = 3300. Что получится при вычислении? ________

```
color != "orange"
```

Оператор != проверяет, что два значения НЕ равны.

Если переменная color содержит строку “pink”, будет это выражение истинным или ложным? __________

Вопрос на повышенную оценку!

```
1000 + "108"
```

Существует несколько ответов. Из них правилен только один. Какой вы выберете? ______________________

Для любознательных

Вы заметили, что в присваивании используется оператор =, а при проверке ==? Когда вы присваиваете значение переменной, ставьте один знак равенства =, а когда проверяете, совпадают ли два значения, — два знака ==. Начинающие программисты часто путают эти операторы.

```
while (juggling) {
    keepBallsInAir();
}
```

Многократное выполнение операций

Нам часто приходится повторно выполнять одни и те же операции:

Прополоскать, сплюнуть, повторить.

Скушать еще ложечку. А потом еще.

Брать конфеты из коробки, пока они не кончатся.

Конечно, в программах операции тоже приходится выполнять повторно. JavaScript предоставляет несколько синтаксических конструкций для повторного выполнения кода: **while**, **for**, **for in** и **forEach**. Со временем мы рассмотрим все эти разновидности циклов, а пока сосредоточимся на **while**.

Мы недавно рассматривали выражения, вычисление которых дает логический результат (например, `scoops > 0`). Такие выражения играют ключевую роль в командах while:

```
while (scoops > 0) {
    document.write("Another scoop!");
    scoops = scoops - 1;
}
```

Цикл while начинается с ключевого слова while.

Цикл while содержит логическое выражение, которое называется условием продолжения (или просто условием).

Если условие цикла истинно, то весь программный блок выполняется.

Что такое «программный блок»? Все, что заключено в фигурные скобки { }.

Если условие истинно, то после выполнения программного блока происходит возврат к началу цикла, и все повторяется заново. Если же условие ложно, цикл завершается.

И так до победного конца!

Как работает цикл while

Раз уж это ваш первый цикл while, давайте подробно рассмотрим его выполнение и досконально разберемся, как же он работает. Обратите внимание: мы добавили объявление переменной scoops и инициализировали ее значением 5.

Выполнение кода начинается. Сначала переменной scoops (шарики мороженого) присваивается значение 5.

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

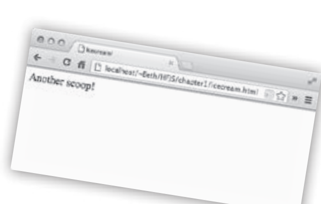
Затем программа переходит к команде while. Когда команда while выполняется в первый раз, сначала программа проверяет ее условие — истинно оно или ложно?

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Значение scoops больше нуля? Несомненно!

**Так как условие истинно, начинается выполнение программного блока. Первая команда в теле цикла выводит в браузер строку “Another scoop!
”.**

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



Следующая команда уменьшает текущее значение scoops на 1 и присваивает результат (4) все той же переменной scoops.

1 шарик съели, 4 осталось!

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Это последняя команда в блоке. Цикл возвращается к началу, и все повторяется заново.

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

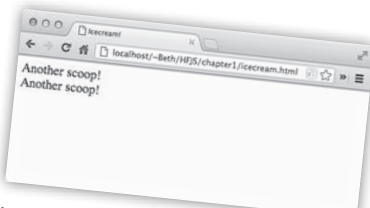
Условие проверяется снова; на этот раз переменная scoops равна 4. Но хотя значение scoops уменьшилось, оно по-прежнему больше нуля.

Еще много осталось!

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

**И снова в браузер выводится строка “Another scoop!
”.**

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



2 шарика ушло, 3 осталось!

Следующая команда уменьшает текущее значение scoops на 1 и присваивает результат (3) той же переменной scoops.

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Это последняя команда блока; программа возвращается к условию, и все повторяется заново.

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Условие проверяется снова, переменная scoops равна 3. Она все еще больше нуля.

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

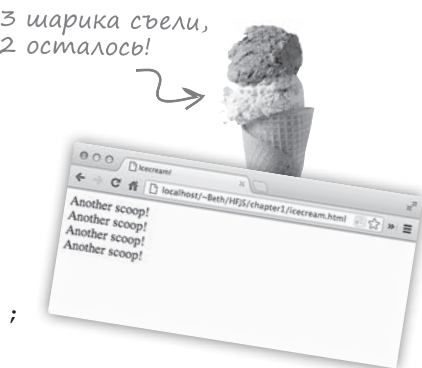
Шарики еще остались!

**Браузеру снова передается строка “Another scoop!
”.**

```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Another scoop!
Another scoop!
Another scoop!

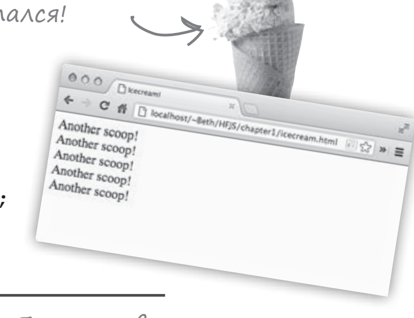
Все продолжается снова и снова: при каждом выполнении цикла scoops уменьшается на 1, в браузер выводится очередная строка, а программа заходит на очередной круг.




```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

И продолжается...

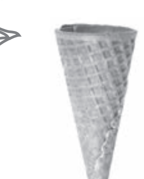
4 шарика съели, 1 остался!




```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

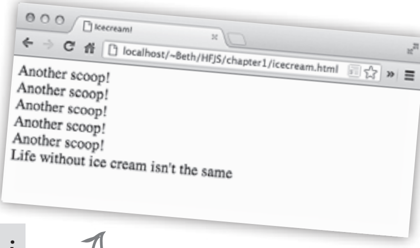
До последнего раза... но теперь кое-что изменилось. Переменная scoops равна нулю, а условие оказывается ложным. На этом все кончается; цикл больше выполняться не должен. На этот раз программа обходит блок и выполняет команду, следующую за ним.

5 шариков съели, осталось 0!



```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Очередная команда выполняет document.write и выводит строку "Life without ice cream isn't the same". Готово!




```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Принятие решений в JavaScript

Вы уже видели, как условное выражение используется для принятия решения о том, должно ли продолжаться выполнение цикла `while`. Логические выражения также могут использоваться для принятия решений в командах JavaScript, использующих `if`. Команда `if` выполняет свой программный блок только в том случае, если проверяемое условие истинно. Пример:

Ключевое слово if, за которым следует условие и программный блок.

Это условие проверяет, осталось ли меньше трех шариков мороженого.

```
if (scoops < 3) {
    alert("Ice cream is running low!");
}
```

И если осталось меньше трех, выполняется программный блок команды if.

Функция alert получает строку и выводит ее во всплывающем окне браузера. Попробуйте!

Команда `if` позволяет последовательно выполнить несколько проверок; для этого добавляется одна или несколько секций `else if`:

Сначала проверяется одно условие, если оно не выполняется, то проверяется другое, указанное в if/else:

```
if (scoops >= 5) {
    alert("Eat faster, the ice cream is going to melt!");
} else if (scoops < 3) {
    alert("Ice cream is running low!");
}
```

Добавьте столько дополнительных проверок "else if", сколько вам нужно; с каждым условием связывается свой программный блок, выполняющийся в случае его истинности.

А если нужно принять МНОГО решений...

Вы можете объединить сколько угодно команд `if/else`. Также можно добавить завершающую секцию `else`; если ни одно условие не выполняется, вы сможете обработать и эту ситуацию, как в следующем примере:

```
if (scoops >= 5) {
   alert("Eat faster, the ice cream is going to melt!");
} else if (scoops == 3) {
   alert("Ice cream is running low!");
} else if (scoops == 2) {
   alert("Going once!");
} else if (scoops == 1) {
   alert("Going twice!");
} else if (scoops == 0) {
   alert("Gone!");
} else {
   alert("Still lots of ice cream left, come and get it.");
}
```

Сначала мы проверяем, осталось ли пять и более шариков мороженого...

...Или если осталось ровно три шарика...

...Или если осталось 2, 1 или 0 шариков... В каждом случае выводится соответствующий сигнал.

Если ни одно из предыдущих условий не является истинным, то будет выполнен этот код.

Часто задаваемые вопросы

В: Что такое «программный блок»?

О: На уровне синтаксиса программный блок (который обычно называется просто блоком) представляет собой набор команд — одну или несколько, заключенных в фигурные скобки. Все команды блока образуют группу, которая выполняется как единое целое. Например, если условие while истинно, то будут выполнены все команды в блоке. Это относится и к блокам if или else if.

В: Я видел код, в котором условие состоит из одной переменной, и эта переменная содержит даже не логическое значение, а строку. Как он работает?

О: Эта тема рассматривается чуть позже, но в двух словах — язык JavaScript достаточно гибок в отношении того, что он считает истинными или ложными значениями. Например, любая переменная, которая содержит (непустую) строку, считается истинной, а переменная, которой еще не было присвоено значение, считается ложной. Подождите, вскоре мы рассмотрим этот вопрос будет подробно.

В: Вы сказали, что результатом выражения могут быть не только числа, строки или логические значения, но и что-то еще. А что именно?

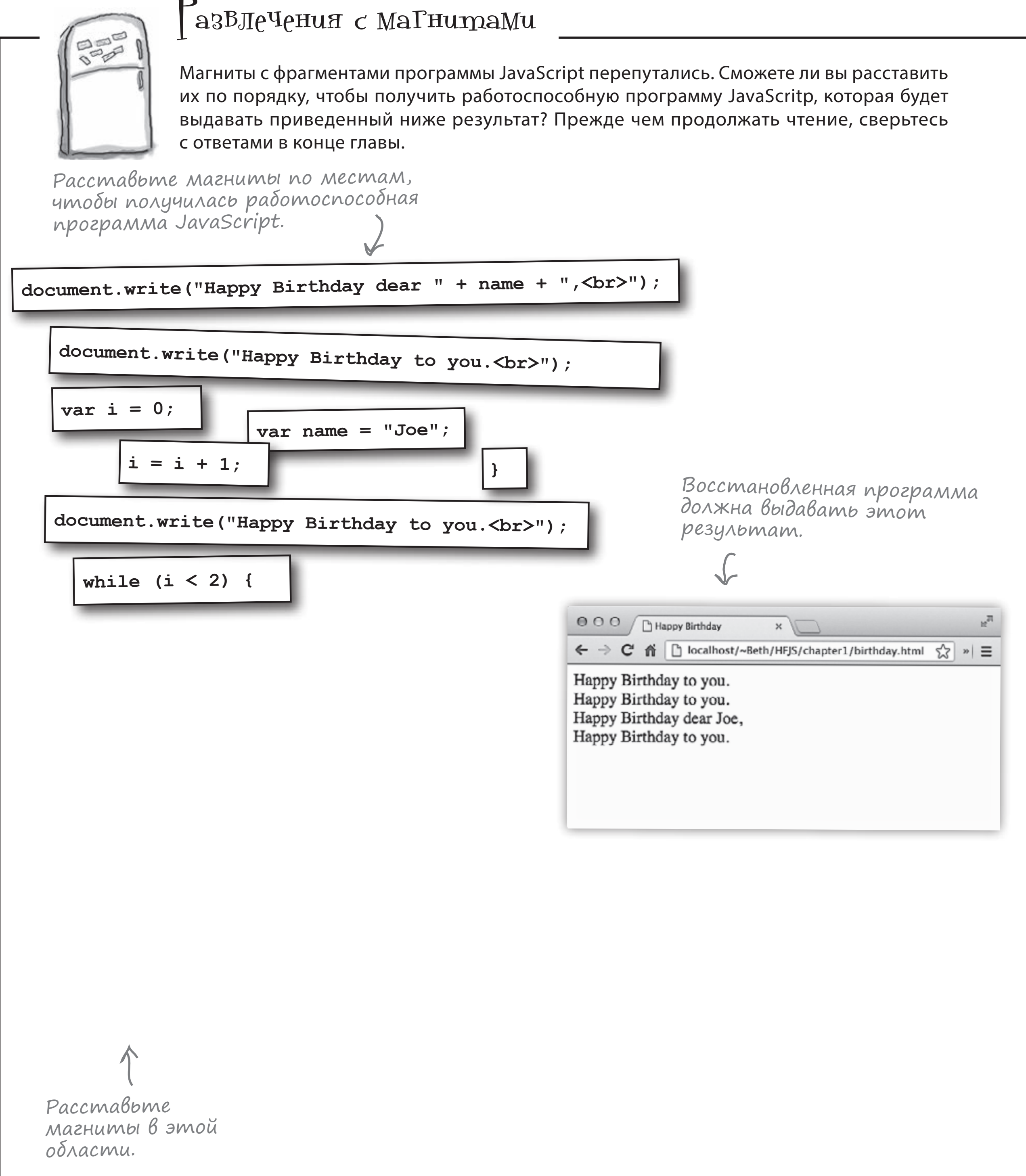
О: Сейчас мы ограничимся примитивными типами: числами, строками и логическими значениями. Но со временем мы займемся более сложными типами: массивами (наборами значений), объектами и функциями.

В: Откуда взялось название «булевские значения» (другое название логических значений)?

О: Оно происходит от имени Джорджа Буля — английского математика, создателя булевой логики.

Развлечения с магнитами

Магниты с фрагментами программы JavaScript перепутались. Сможете ли вы расставить их по порядку, чтобы получить работоспособную программу JavaScritp, которая будет выдавать приведенный ниже результат? Прежде чем продолжать чтение, сверьтесь с ответами в конце главы.



Привлекайте пользователя к взаимодействию со страницей

Мы говорили о том, как важно повысить уровень интерактивности страниц и что для этого необходимо взаимодействие с пользователем. Это взаимодействие можно организовать несколькими способами; некоторые из них уже встречались вам. Ниже приведена краткая сводка таких взаимодействий, а потом мы рассмотрим их более подробно.

Создание сигнала

Браузер поддерживает простейший механизм оповещения пользователей при помощи функции `alert`. Вызовите `alert` со строкой, содержащей сообщение, и браузер выведет ее в симпатичном диалоговом окне. Честно говоря, мы немного злоупотребляли этой функцией, она очень проста и удобна; но ее следует применять только тогда, когда вы действительно хотите, чтобы пользователь отложил все дела и немедленно ознакомился с вашим сообщением.

Эти три способа встречаются в данной главе.

Прямая запись в документ

Веб-страницу можно рассматривать как документ (именно так ее называет браузер). Функция `document.write` позволяет вывести произвольную разметку HTML и контент в произвольной точке страницы. Так поступать не рекомендуется, хотя время от времени этот способ все же применяется. Мы воспользовались, потому что он достаточно прост для изучения JavaScript.

Вывод на консоль

В каждой среде JavaScript существует консоль, на которую можно выводить сообщения из программного кода. Чтобы вывести сообщение на консоль, вызовите функцию `console.log` и передайте ей строку (вскоре мы рассмотрим работу с консолью более подробно). Функция `console.log` — отличный инструмент диагностики и отладки. Впрочем, в нормальной программе вывод на консоль спрятан от пользователя, так что данный механизм вряд ли можно назвать эффективным.

Консоль — удобный инструмент для поиска ошибок в коде! Если вы ошиблись при вводе (например, пропустили кавычку), JavaScript обычно выводит на консоль описание ошибки, которое упростит ее поиск.

Непосредственная модификация документа

Основной способ взаимодействия со страницей и пользователями — средства JavaScript, позволяющие обращаться к разметке веб-страницы, читать и изменять ее содержимое, даже изменять структуру и стиль! Все эти операции выполняются через *объектную модель документа* (см. далее). Как вы вскоре убедитесь, это лучший вариант взаимодействия с пользователем. Однако для работы с объектной моделью документа необходимо знать структуру страницы и программный интерфейс, используемый при чтении и записи страницы. Вскоре мы доберемся до этой темы, но сначала нужно освоить еще немного JavaScript.

Заветная цель, к которой мы стремимся. Достигнув ее, вы сможете читать страницу, изменять ее и выполнять любые манипуляции с ее содержимым.

Все средства взаимодействия явились на маскарад. Удастся ли вам узнать их под масками? Соедините описания справа с именами в левом столбце. Мы провели одну линию за вас.

document.write

console.log

alert

объектная модель документа

Я немедленно останавливаю то, чем занимается пользователь, и выдаю короткое сообщение. Чтобы двигаться дальше, пользователь должен нажать «OK».

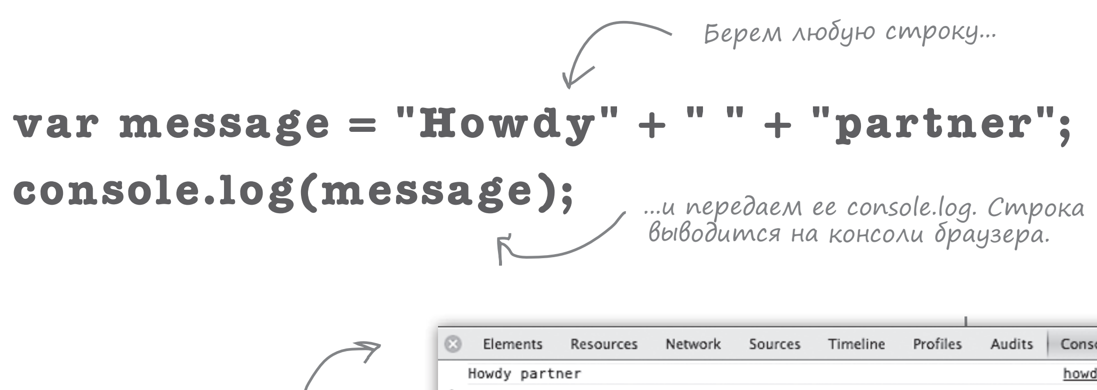
Я умею вставлять небольшие фрагменты разметки HTML и текста в документ. Может, я и не самый элегантный способ передачи информации пользователю, но зато я работаю во всех браузерах.

С моей помощью вы сможете полностью управлять веб-страницей: получать данные, введенные пользователем, изменять разметку HTML и стили, обновлять содержимое страницы.

Я существую исключительно для решения простых задач отладки, и позволяю выводить информацию на специальную консоль, предназначенную для разработчиков.

Близкое знакомство с console.log

Давайте разберемся, как работает функция `console.log`. Мы воспользуемся ею в этой главе для просмотра результатов выполнения кода, в других главах книги — для анализа и отладки выходных данных. Но не забывайте, что консоль остается скрытой от большинства рядовых веб-пользователей, поэтому использовать ее в окончательной версии веб-страницы нежелательно. Вывод на консоль обычно используется для диагностики в процессе разработки страницы. Кроме того, консоль помогает увидеть, что происходит в вашем коде, в процессе изучения основных конструкций JavaScript. Вот как это происходит:



Консоль содержит все данные, которые выводятся вызовами console.log в вашем приложении.

Часто Задаваемые Вопросы

В: Я понимаю, что console.log может использоваться для вывода строк, но что это вообще такое? Почему “console” и “log” разделены точкой?

О: Хороший вопрос. Мы немного забегаем вперед, но рассматривайте консоль как объект, выполняющий различные операции (консольные). Одна из таких задач — вывод данных на консоль. Чтобы приказать консоли выполнить эту операцию, мы используем синтаксис “console.log” и передаем выводимое значение в скобках. Помните об этом; объекты будут намного подробнее рассматриваться в этой книге. А пока достаточно просто уметь использовать console.log.

В: А консоль умеет делать что-нибудь еще, кроме обычного вывода?

О: Да, но чаще она используется для обычного вывода. Существует еще несколько способов использования вывода (и консоли), но они относятся к специфике конкретных браузеров. Консоль поддерживается всеми современными браузерами, но не определяется ни в одной формальной спецификации.

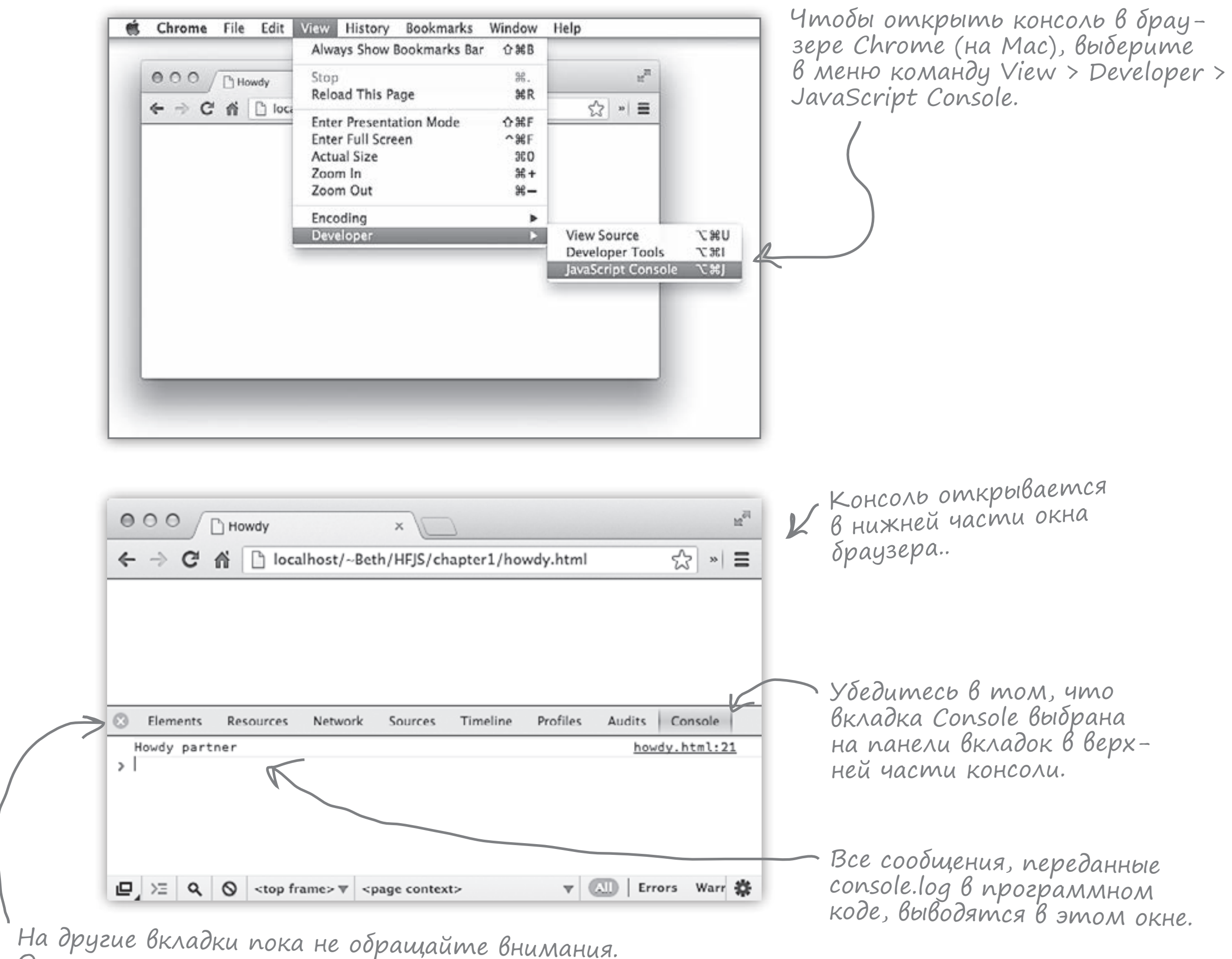
В: Все это, конечно, хорошо, но как найти консоль? Я использую ее в своем коде, но не вижу выводимых данных!

О: Во многих браузерах окно консоли открывается отдельной командой. За подробностями переходите к следующей странице.

Как открыть консоль

В разных браузерах используются разные реализации консоли. А если вам этого недостаточно, то знайте, что реализация консоли браузером достаточно часто меняется — может быть и незначительно, но все же может оказаться, что консоль вашего браузера будет отличаться от изображенной на иллюстрации.

Поэтому здесь мы покажем, как открыть консоль в браузере Chrome (версия 25) на Mac, а инструкции по обращению к консоли во всех основных браузерах доступны в Интернете по адресу http://wickedlysmart.com/hfjsconsole. Вероятно, освоив работу с консолью в одном браузере, вы сможете без особых проблем разобраться и с другими. Мы рекомендуем опробовать операции с консолью как минимум в двух браузерах, пока вы не начнете достаточно уверенно чувствовать себя при работе с ней.



Пишем серьезное приложение на JavaScript

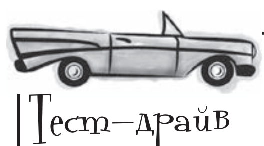
Давайте применим новые знания JavaScript и `console.log` на практике и сделаем что-нибудь полезное. Нам понадобятся переменные, команда `while`, несколько команд `if` и `else`. Немного усилий — и у вас получится серьезное приложение коммерческого уровня. Но прежде чем рассматривать код, подумайте, а как бы вы запрограммировали классическую песенку про «99 бутылок пива»?

```
var word = "bottles";

var count = 99;

while (count > 0) {

    console.log(count + " " + word + " of beer on the wall");

    console.log(count + " " + word + " of beer,");

    console.log("Take one down, pass it around,");

    count = count - 1;

    if (count > 0) {

        console.log(count + " " + word + " of beer on the wall.");

    } else {

        console.log("No more " + word + " of beer on the wall.");

    }

}
```

У этого кода есть небольшой недостаток. Он работает правильно, но вывод нельзя назвать абсолютно идеальным. Удастся ли вам найти этот недостаток и исправить его?

Верно! Но всему свое время. Просто мы хотели дать вам достаточно знаний JavaScript, чтобы задача была более интересной. Впрочем, в начале главы уже было показано, что код JavaScript включается в HTML точно так же, как и CSS; иначе говоря, он просто встраивается прямо в документ в тегах `<script>`.

Как и CSS, код JavaScript может храниться в файлах, внешних по отношению к HTML. Так что сначала мы встроим код своего «серьезного бизнес-приложения» в страницу, а после тщательного тестирования переместим JavaScript во внешний файл.

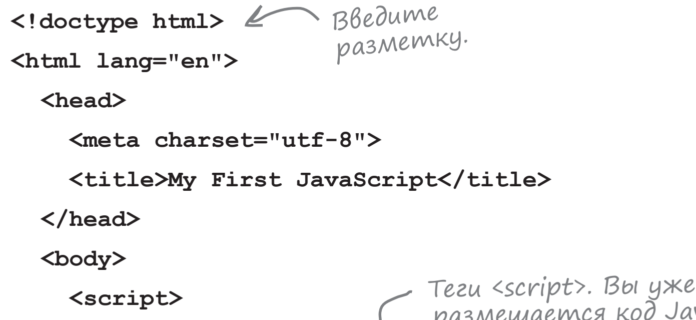


Тест-драйв

Итак, введем код в браузере... Выполните приведенные ниже инструкции и запустите мегаприложение! Результат тоже показан внизу:

Чтобы загрузить весь код и файлы примеров, посетите страницу http://wickedlysmart.com/hfjs.

1 Просмотрите на разметку HTML; в нее будет вставлен код JavaScript. Наберите HTML-код и включите фрагмент JavaScript между тегами <script>, приведенный пару страниц назад. Для этого подойдет Блокнот (Windows) или TextEdit (Mac), работающие в режиме простого текста. Или, если у вас имеется любимый редактор HTML — Dreamweaver, Coda или WebStorm, вы можете работать в нем.



```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>

    </script>
  </body>
</html>
```

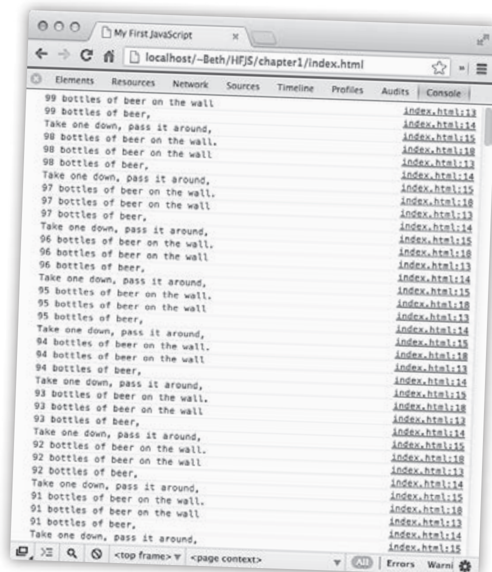
Введите разметку.

Теги <script>. Вы уже знаете, что в них размещается код JavaScript.

2 Сохраните файл под именем “index.html”.

3 Загрузите файл в браузере. Либо перетащите файл прямо в окно браузера, либо выберите команду Файл > Открыть (или Файл > Открыть файл) в своем браузере.

4 На самой веб-странице никакие данные не выводятся, потому что весь вывод направляется на консоль вызовом console.log. Откройте консоль в браузере и поздравьте себя с серьезным успехом.



Результат тестового запуска нашего кода. Программа генерирует полный текст песенки «99 бутылок пива» и выводит текст на консоль.

Как добавить код в страницу? (считаем способы)

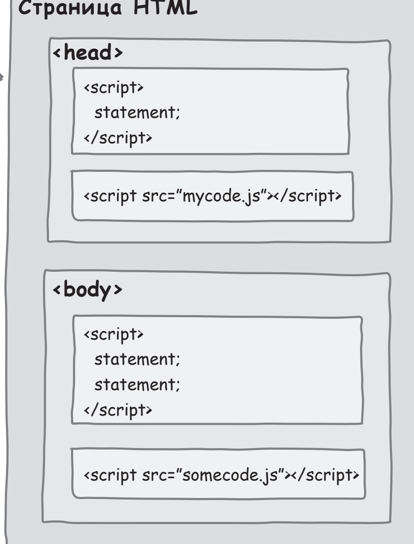
Как вы уже знаете, элемент `<script>` с кодом JavaScript можно добавить в разделы `<head>` или `<body>` вашей страницы, но существует пара других способов добавления кода в страницу. Рассмотрим все места, в которых можно разместить код JavaScript (и почему его лучше размещать в том или ином месте).

Код можно встроить в элемент <head>. Размещение кода в элементе <script> секции <head> — самый распространенный способ включения кода в страницы. Конечно, в этом случае ваш код будет легко найти, да и выбор места выглядит логично, но это не всегда лучший вариант размещения. Почему? Читайте дальше…

Также код можно встроить в тело документа. Для этого заключите код JavaScript в элемент <script> и разместите его в разделе <body> страницы (как правило, в конце тела).

Этот вариант лучше предыдущего. Почему? Потому что во время загрузки браузер загружает все содержимое раздела <head> страницы и только потом переходит к разделу <body>. Если ваш код расположен в <head>, пользователю придется немного подождать, чтобы увидеть страницу. Если же код загружается после разметки HTML в <body>, пользователь будет видеть содержимое страницы во время загрузки.

А может, есть другой способ — еще лучше? Продолжайте читать…



Третий вариант — разместить код в отдельном файле и включить ссылку на него в элемент <head>. Это делается так же, как ссылка на файл CSS. Единственное различие в том, что в атрибуте src тега <script> указывается URL-адрес файла JavaScript.

Хранение кода во внешнем файле упрощает сопровождение (отдельно от разметки HTML), его можно использовать в нескольких страницах. Но и у этого метода имеется недостаток: код должен быть загружен до тела страницы. Как же с ним справиться? Продолжайте читать…

Наконец, ссылку на внешний файл можно включить в тело страницы. Ага, все лучшее с обеих сторон... Мы имеем удобный, простой в сопровождении файл JavaScript, который можно включить в любую страницу, и ссылка располагается в конце тела страницы, так что код будет загружаться только после загрузки тела страницы. Отлично!

Что ни говорите, а я считаю, что <head> — лучшее место для размещения кода.

Разметка и код: пути расходятся

Расставаться неприятно, но другого выхода не было. Пришло время взять код JavaScript и поселить его в отдельном файле. Вот как это делается...

Откройте index.html и выделите весь код (то есть весь текст между тегами <script>). Выделение должно выглядеть примерно так:

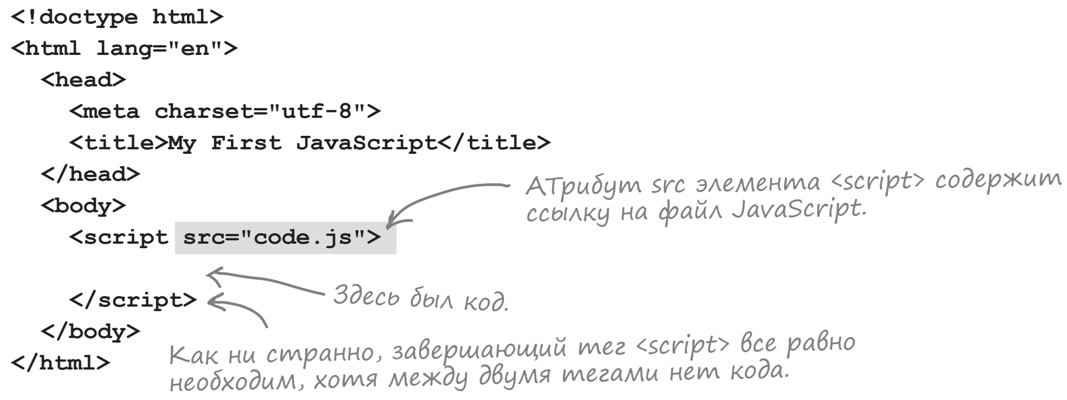
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>
      var word = "bottles";
      var count = 99;
      while (count > 0) {
         console.log(count + " " + word + " of beer on the wall");
         console.log(count + " " + word + " of beer,");
         console.log("Take one down, pass it around,");
         count = count - 1;
         if (count > 0) {
            console.log(count + " " + word + " of beer on the wall.");
         } else {
            console.log("No more " + word + " of beer on the wall.");
         }
      }
    </script>
  </body>
</html>
```

Выделите только код без тегов <script>; во внешнем файле они не понадобятся...

2. Создайте в редакторе новый файл с именем “code.js” и вставьте в него скопированный код. Сохраните файл “code.js”.

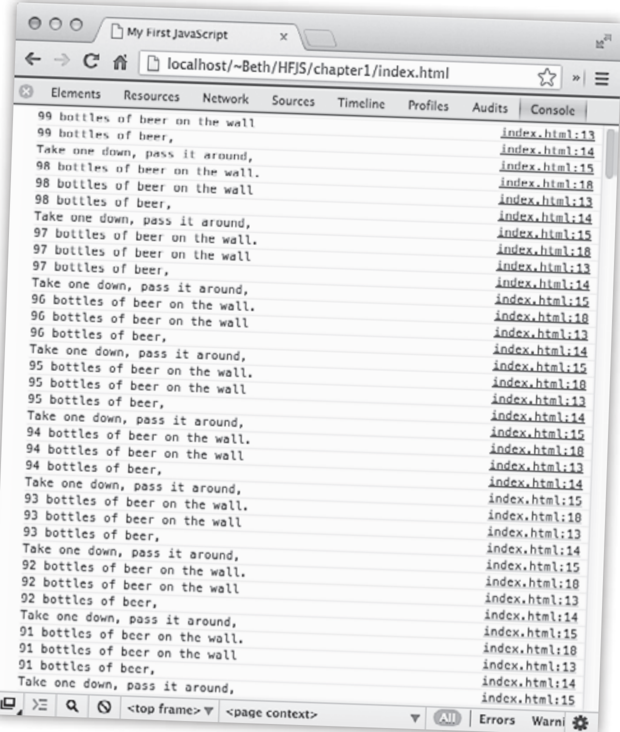
code.js

3 Теперь нужно включить ссылку на файл "code.js" в файл "index.html", чтобы файл загружался при загрузке страницы. Для этого удалите код JavaScript из "index.html", но оставьте теги <script>. Затем добавьте в открывающий тег <script> атрибут src со ссылкой на "code.js".



```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script src="code.js">

    </script>
  </body>
</html>
```


4 Вот и все, косметическая операция завершена. Давайте посмотрим, успешно ли она прошла. Перезагрузите страницу "index.html"; результат должен выглядеть точно так же, как прежде. Учтите, что при использовании ссылки вида src="code.js" предполагается, что файл кода расположен в одном каталоге с файлом HTML.



Результат выглядит так же, как прежде, но теперь разметка HTML и код JavaScript хранятся в разных файлах. Согласитесь, такая структура выглядит более стройной и удобной, и создает меньше проблем с сопровождением.

Анатомия элемента Script

Вы научились использовать элемент <script> для включения кода в страницу. Чтобы окончательно снять все вопросы, давайте рассмотрим элемент <script> в мельчайших подробностях:

```
<script type="text/javascript" >
    alert("Hello world!");
</script>
```

Атрибут type сообщает браузеру, что дальше идет код JavaScript. Впрочем, если он не указан, браузер по умолчанию считает, что это именно JavaScript. По этой причине мы рекомендуем опускать атрибут type — и то же самое рекомендуют разработчики стандартов.

Открывающий тег <script>.

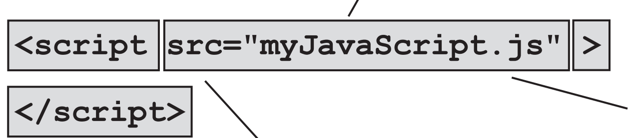
Не забудьте правую угловую скобку в открывающем теге.

Весь текст между тегами <script> должен быть действительным кодом JavaScript.

Сценарный код всегда должен завершаться тегом </script>, всегда!

А для ссылки на отдельный файл JavaScript из разметки HTML используется элемент <script> следующего вида:

Добавьте атрибут src с URL-адресом файла JavaScript.



Файлы JavaScript имеют расширение ".js".

Напоминаем: не забудьте закрывающий тег </script>! Он необходим даже в том случае, если в разметку включается ссылка на внешний файл.

Если вы включаете ссылку на отдельный файл JavaScript, элемент <script> не содержит кода JavaScript.

Будьте осторожны!

Встроенный код JavaScript не может объединяться с внешним.

Если вы попытаетесь добавить код между тегами <script>, в которых уже используется атрибут src, у вас ничего не выйдет. Для этого нужны два отдельных элемента <script>.

```
<script src="goodies.js">
   var = "quick hack";
</script>
```

ОШИБКА

Откровенно о JavaScript

Интервью недели:
Знакомьтесь — JavaScript

Head First: Добро пожаловать, JavaScript. Вы постоянно заняты, работаете в множестве веб-страниц... Спасибо, что нашли время пообщаться с нами.

JavaScript: Нет проблем. У меня *действительно* хватает дел; JavaScript используется едва ли не на каждой современной веб-странице для самых разных целей, от простых эффектов меню до полноценных игр. Ни единой свободной минуты!

Head First: Удивительно. Всего несколько лет назад говорили, что вы «недоделанный, игрушечный язык сценариев», а сегодня вы повсюду.

JavaScript: Не напоминайте мне о тех временах. Утекло много воды, и многие выдающиеся умы трудились над тем, чтобы меня улучшить.

Head First: В чем улучшить? Основные возможности почти не изменились...

JavaScript: Ну, в нескольких отношениях. Во-первых, сейчас я работаю с молниеносной быстротой. Хотя меня по-прежнему считают языком сценариев, теперь по быстродействию я почти не уступаю традиционным компилируемым языкам.

Head First: А во-вторых?

JavaScript: Мои умения выполнять разные операции сильно расширились. Библиотеки JavaScript, доступные в современных браузерах, определяют географическое местоположение, воспроизводят аудио- и видеоролики, выводят графику на веб-странице и делают многое другое. Но чтобы все это было возможно, необходимо знать JavaScript.

Head First: Вернемся к критике. Мы слышали и не столь любезные отзывы... Кажется, вас называли «языком для поделок».

JavaScript: Ну и что? Я один из самых распространенных языков... А может, и *самый* распространенный. Я сражался с конкурентами и победил. Помните Java в браузере? Жалкое зрелище. VBScript? Ха-ха. JScript? Flash?! Silverlight? Я мог бы продолжать. Разве я выжил бы, если бы был плохим?

Head First: Вас критиковали за некоторую... «упрощенность».

JavaScript: Честно говоря, это одна из моих сильных сторон. Вы запускаете браузер, вводите несколько строк JavaScript, и программа работает. Кроме того, я отлично подхожу для новичков. Говорят, для начинающего программиста нет более подходящего языка, чем JavaScript.

Head First: Но за простоту приходится платить?

JavaScript: Да, я прост в том смысле, что со мной можно сходу взяться за работу. Но при этом я достаточно глубок, и во мне реализованы все современные программные конструкции.

Head First: Например?

JavaScript: Что вы скажете о динамических типах, полноценных функциях и замыканиях?

Head First: Откровенно говоря, ничего. Я не знаю, что это такое.

JavaScript: Понятно... Ничего страшного. Продолжайте читать, и вы все узнаете.

Head First: Ну хотя бы в общих чертах?

JavaScript: JavaScript проектировался для работы в динамичной веб-среде, где пользователь взаимодействует со страницами, данные меняются, происходят события, а язык отражает данный стиль программирования. Мы вернемся к этой теме позднее, когда вы больше узнаете о JavaScript.

Head First: Послушать вас, так вы идеальный язык. Это правда?

JavaScript стискивает зубы...

JavaScript: Знаете, я вырос не в стенах академических учреждений, как большинство языков. Я родился в реальном мире, и очень быстро оказался перед выбором — тонуть или выплывать. Я не идеален; безусловно, у меня есть свои «плохие стороны».

Head First с улыбкой ведущего: Сегодня мы увидели вас с новой стороны. У нас есть тема для будущего разговора. Что-нибудь скажете на прощание?

JavaScript: Не судите меня по тому, что несовершенно. Изучайте лучшее и используйте в работе!

КЛЮЧЕВЫЕ МОМЕНТЫ

- JavaScript используется для определения **поведения** в веб-страницах.
- Современные браузеры выполняют JavaScript намного быстрее, чем несколько лет назад.
- Браузер начинает выполнять код JavaScript сразу же, как только он обнаружит этот код на странице.
- Для добавления JavaScript на страницу используется элемент **<script>**.
- Код JavaScript можно встроить в веб-страницу или же включить в разметку HTML ссылку на отдельный файл JavaScript.
- Для определения ссылки на отдельный файл JavaScript используется атрибут **src** тега <script>.
- HTML **объявляет** структуру и содержимое страницы; JavaScript занимается **вычислениями** и определяет поведение страницы.
- Программа JavaScript состоит из серии **команд**.
- Одна из самых распространенных команд JavaScript — объявление переменной. Ключевое слово **var** используется для объявления новой переменной, а оператор присваивания **=** задает ее значение.
- Имена переменных JavaScript строятся по определенным правилам, которые необходимо соблюдать.
- Помните, что ключевые слова JavaScript не могут быть именами переменных.
- Выражения JavaScript используются для вычисления значений.
- Три основных типа выражений — **числовые**, **строковые** и **логические**.
- Команды **if/else** используются для принятия решений в коде.
- Команды **while/for** используются для многократного выполнения команд в цикле.
- Используйте функцию **console.log** (вместо **alert**) для вывода сообщений на консоль.
- Консольные сообщения используются в основном для диагностики и отладки. Скорее всего, пользователи никогда их не увидят.
- Язык JavaScript чаще всего используется для определения поведения веб-страниц, но он также может использоваться для программирования сценариев в таких приложениях, как Adobe Photoshop, OpenOffice и Google Apps, и даже для программирования на стороне сервера.

СТАНЬ браузером. Решение

Ниже приведен код JavaScript, содержащий ошибки. Представьте себя на месте браузера и попробуйте отыскать ошибки в коде. А когда это будет сделано, загляните в ответы в конце главы и посмотрите, не упустили ли вы чего-нибудь. Ниже приведено наше решение.

A

```
// Test for jokes
var joke = "JavaScript walked into a bar....';
var toldJoke = "false";
var $punchline =
  "Better watch out for those semi-colons."
var %entage = 20;
var result

if (toldJoke == true) {
    Alert($punchline);
} else
    alert(joke);
}
```

Строки должны заключаться либо в двойные кавычки ("), либо в одиночные кавычки ('). Смешивать разные кавычки нельзя!

Логические значения не заключаются в кавычки, если только вы не собираетесь работать с ними как со строками.

Имя переменной может начинаться с $, хотя лучше так не делать.

Команды должны завершаться «;»!

Символ % не может использоваться в именах переменных.

Снова пропущен завершающий символ «;».

Должно быть alert, а не Alert. Язык JavaScript учитывает регистр символов.

Пропущена открывающая скобка.

B

```
\\ Movie Night
var zip code = 98104;
var joe'sFavoriteMovie = Forbidden Planet;
var movieTicket$    =    9;

if (movieTicket$ >= 9) {
    alert("Too much!");
} else {
    alert("We're going to see " + joe'sFavoriteMovie);
}
```

Комментарии должны начинаться с //, а не с \\.

В именах переменных не может быть пробелов.

Имена переменных не могут содержать кавычки.

Но строка "Forbidden Planet" должна быть заключена в кавычки.

Команда if/else не работает из-за недействительного имени переменной.

Возьми в руку карандаш
Решение

Возьмите карандаш. Для каждого из приведенных ниже выражений вычислите его значение и запишите в выделенном месте. Да, ЗАПИШИТЕ… забудьте слова мамы о том, что нельзя писать в книгах, и запишите ответ на странице! Посмотрите наше решение.

Калькулятор пересчета температуры по Цельсию в шкалу Фаренгейта?

```
(9 / 5) * temp + 32
```

Что получится, если переменная temp равна 10? *50*

Логическое выражение. Оператор == проверяет, равны ли два значения.

```
color == "orange"
```

Истинным или ложным будет это выражение, если переменная color содержит значение “pink”? *false*
А если значение “orange”? *true*

```
name + ", " + "you've won!"
```

Какое значение будет, если переменная name содержит строку “Martha”? *“Martha, you’ve won!”*

```
yourLevel > 5
```

Проверка: «первое значение больше второго?» Можно использовать оператор >= для условия «первое значение больше либо равно второму?»

Что получится, если переменная yourLevel = 2? *false*
Что получится, если переменная yourLevel = 5? *false*
Что получится, если переменная yourLevel = 7? *true*

```
(level * points) + bonus
```

Переменная level = 5, points = 30 000, a bonus = 3300. Что получится при вычислении? *153300*

```
color != "orange"
```

Если переменная color содержит строку “pink”, это выражение будет истинным или ложным? *true*

Оператор != проверяет, что два значения НЕ равны.

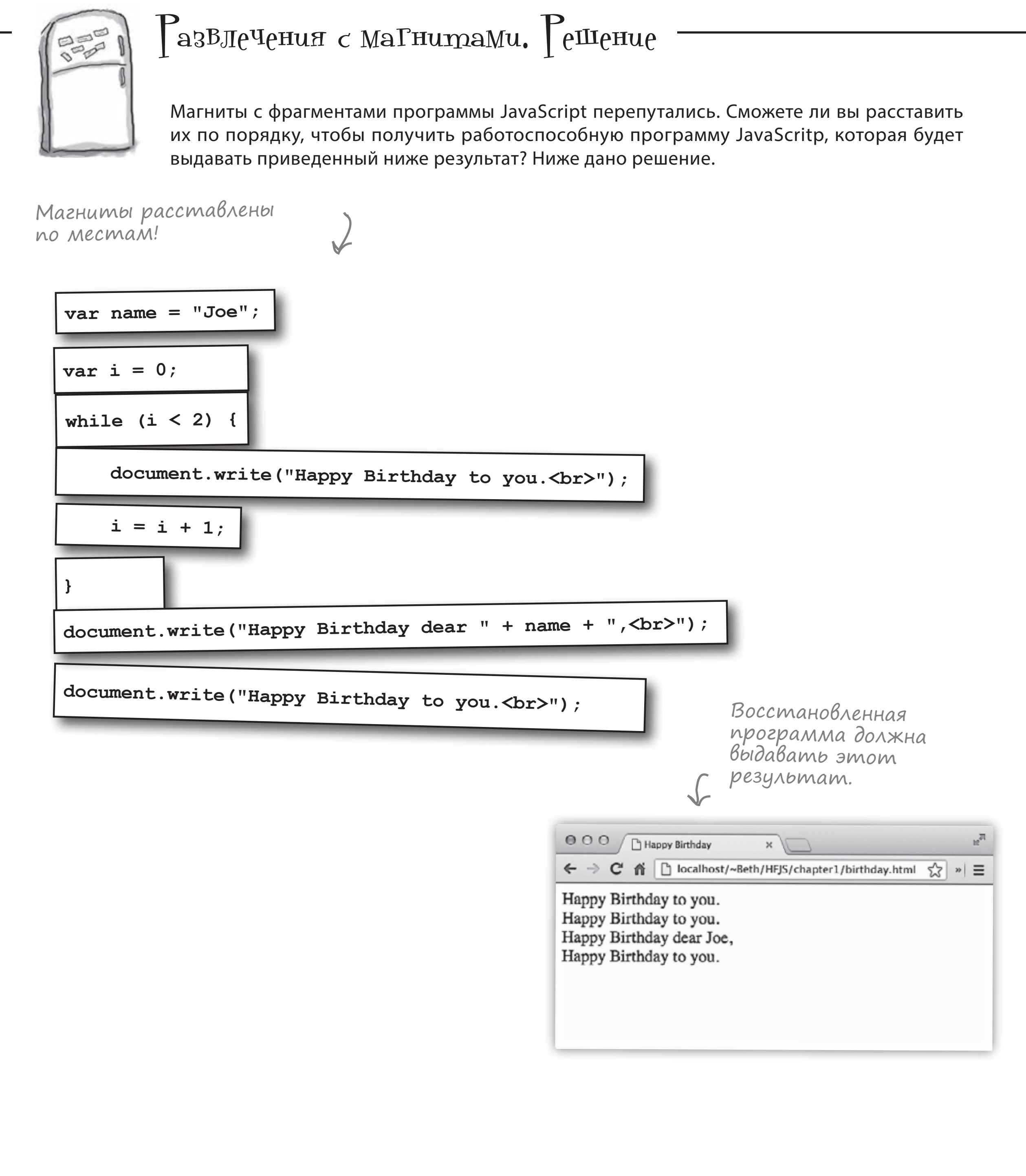
Вопрос на повышенную оценку!

```
1000 + "108"
```

Существует несколько ответов. Из них правилен только один. Какой вы выберете? *“1000108”*

Для любознательных

Вы заметили, что в присваивании используется оператор =, а при проверке ==? Когда вы присваиваете значение переменной, ставьте один знак равенства =, а когда проверяете, совпадают ли два значения, — два знака ==. Начинающие программисты часто путают эти операторы.



Развлечения с магнитами. Решение

Магниты с фрагментами программы JavaScript перепутались. Сможете ли вы расставить их по порядку, чтобы получить работоспособную программу JavaScritp, которая будет выдавать приведенный ниже результат? Ниже дано решение.

Магниты расставлены по местам!

```
var name = "Joe";
var i = 0;
while (i < 2) {
    document.write("Happy Birthday to you.<br>");
    i = i + 1;
}
document.write("Happy Birthday dear " + name + ",<br>");
document.write("Happy Birthday to you.<br>");
```

Восстановленная программа должна выдавать этот результат.

Все средства взаимодействия явились на маскарад. Удастся ли вам узнать их под масками? Соедините описания справа с именами в левом столбце. Наше решение:

document.write	Я умею вставлять небольшие фрагменты разметки HTML и текста в документ. Может, я и не самый элегантный способ передачи информации пользователю, но зато я работаю во всех браузерах.
console.log	Я существую исключительно для решения простых задач отладки и позволяю выводить информацию на специальную консоль, предназначенную для разработчиков.
alert	Я немедленно останавливаю то, чем занимается пользователь, и выдаю короткое сообщение. Чтобы двигаться дальше, пользователь должен нажать «OK».
document object model	С моей помощью вы сможете полностью управлять веб-страницей: получать данные, введенные пользователем, изменять разметку HTML и стили, обновлять содержимое страницы.

2 настоящий код

Следующий шаг

Вы уже знаете, что такое переменные, типы, выражения... и так далее. Вы уже кое-что знаете о JavaScript. Более того, знаний достаточно, чтобы начать писать настоящие программы, которые делают что-то интересное, которыми кто-то будет пользоваться. Правда, вам не хватает практического опыта написания кода, и мы прямо сейчас начнем решать эту проблему. Как? А просто возьмемся за написание несложной игры, полностью реализованной на JavaScript. Задача масштабная, но мы будем двигаться к цели постепенно, шаг за шагом. Итак, беремся за дело, а если вам вдруг захочется использовать нашу разработку в своих проектах — мы не против; распоряжайтесь кодом, как считаете нужным.

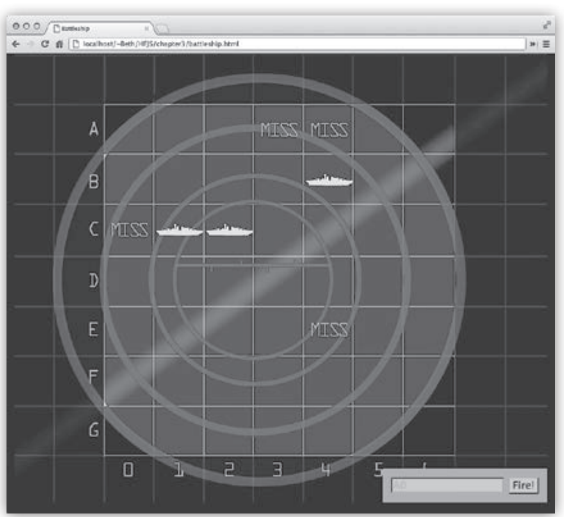
Давайте реализуем игру «Морской бой»

Вы играете против браузера: браузер прячет свои корабли, а вы пытаетесь найти и уничтожить их. В отличие от настоящего «Морского боя», здесь вы не будете расставлять корабли и соревноваться с браузером. Ваша задача — потопить корабли браузера за минимальное количество ходов.

Цель: Потопить корабли за минимальное количество ходов. В зависимости от того, насколько хорошо вам это удалось, вы получаете оценку.

Подготовка: При запуске игровой программы компьютер расставляет корабли на виртуальной «сетке». Когда это сделано, игра запрашивает координаты первого выстрела.

Как играть: по запросу браузера игрок вводит координаты выстрела. Браузер проверяет клетку и выводит результат: «Попал», «Ранил» или «Убил». После того как все корабли будут потоплены, игра заканчивается с выводом оценки.



Так будет выглядеть поле боя: таблица 7x7, на которой размещены три корабля. Сейчас мы упростим задачу, но когда вы будете лучше знать JavaScript, доработаем программу, и результат будет выглядеть именно так — с графикой и всем прочим... А звуковое сопровождение можете реализовать самостоятельно.

Первый заход...

...упрощенный вариант

Вместо того чтобы сразу браться за полноценную графическую версию 7×7 с тремя кораблями, мы начнем с упрощенного варианта — простой одномерной таблицы с семью клетками и одним кораблем. Да, примитивно, но мы занимаемся написанием кода приложения, а не его визуальным оформлением (по крайней мере пока).

Не беспокойтесь, упрощенная версия игры сэкономит немало времени и сил при построении полной реализации. Кроме того, серьезно уменьшен объем работы над первой полноценной программой JavaScript (мы не учитываем Невероятно Серьезное Бизнес-Приложение из главы 1). Итак, в этой главе мы построим простую версию игры, а когда вы будете больше знать о JavaScript, перейдем к полноценной версии.

Вместо таблицы 7x7, как на верхнем рисунке, мы начнем с таблицы 1x7. Да, и корабль пока будет только один.

Каждый корабль занимает три клетки.

Начнем с проектирования

Нам понадобятся переменные, а еще числа и строки, команды if, условные проверки и циклы... но где и сколько? И как все это собрать воедино? Чтобы ответить на эти вопросы, мы должны лучше представлять себе, что должно происходить в игре.

Сначала нужно определить общую структуру игры. Общая схема выглядит так:

1 Пользователь запускает игру

A Игра размещает корабль в случайной позиции таблицы.

2 Начинается бой

Повторять следующие действия, пока корабль не будет потоплен:

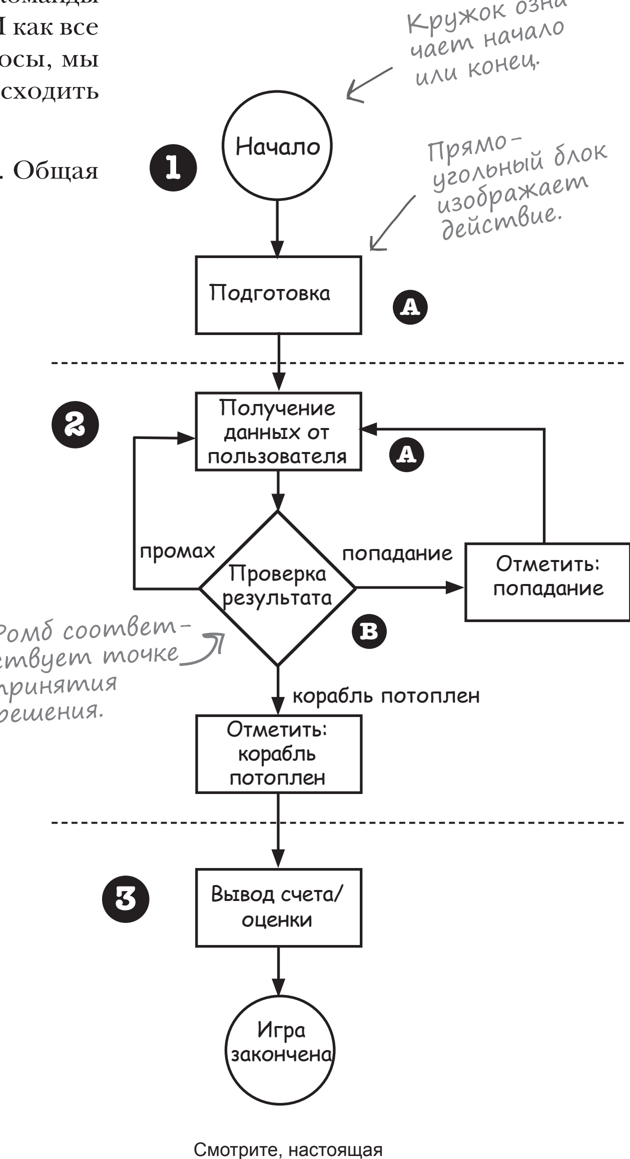
A Запросить у пользователя координаты выстрела («2», «0» и т. д.)

B Сравнить координаты с положением корабля и определить результат: попадание, промах, корабль потоплен.

3 Игра закончена

Вывести оценку, которая определяется по количеству попыток.

Итак, мы в общих чертах представляем, что должна делать программа. Теперь нужно более подробно проработать отдельные шаги.



Смотрите, настоящая блок-схема!

Чуть подробнее...

По высокоуровневому описанию и блок-схеме вполне профессионального вида мы достаточно хорошо представляем, как должна работать игра. Но прежде чем переходить к написанию кода, необходимо разобраться с некоторыми подробностями.

Представление кораблей

Для начала нужно определиться с тем, как будут представляться корабли на игровом поле. Виртуальная сетка игрового поля не зря называется «виртуальной». Иначе говоря, в программе она не существует. Если и игра, и пользователь знают, что корабль располагается в трех смежных клетках из семи (начиная с нуля), то заводить представление для строки таблицы не обязательно. Возможно, вам хочется определить структуру данных, в которой хранится содержимое всех семи ячеек, и попытаться разместить корабль в ней. Но в этом просто нет необходимости. Достаточно знать, в каких ячейках находится корабль (предположим, в ячейках 1, 2 и 3).

Получение данных от пользователя

Для получения данных от пользователя можно использовать функцию `prompt`. Когда пользователь должен ввести новую позицию, программа вызывает функцию `prompt`, выводящую сообщение, и получает данные — число в диапазоне от 0 до 6.

Вывод результатов

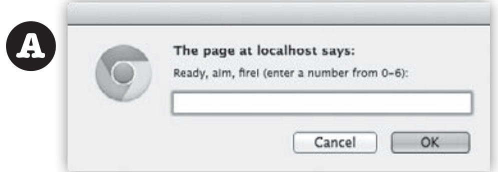
Как организовать вывод? Пока мы продолжим использовать функцию `alert`. Решение топорное, но оно работает. (В полноценной игре мы будем напрямую обновлять веб-страницу, но до этого еще нужно добраться.)

1. **Игра запускается** и создает один корабль, занимающий три клетки в одной строке из семи клеток.

   Позиции клеток обозначаются целыми числами; например, на этом рисунке корабль занимает клетки 1,2,3:

2. **Начинаем!** Запрашиваем координаты выстрела:

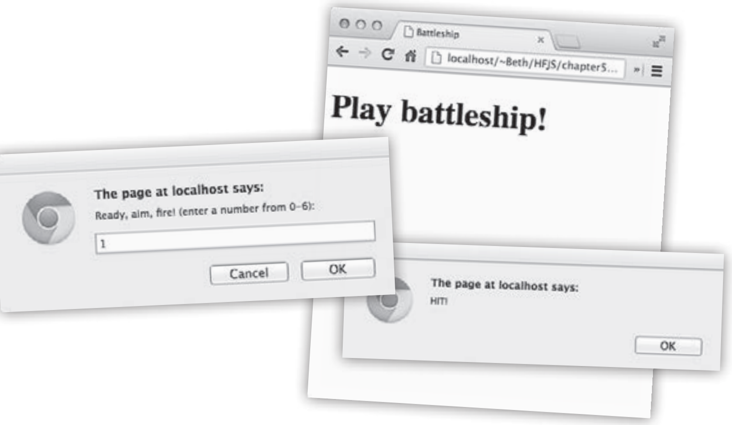
A



B Проверяем, попал ли пользователь в одну из трех клеток корабля. Количество попаданий хранится в переменной.

3. **Игра заканчивается,** когда пользователь попал по всем трем клеткам и количество попаданий в переменной hits достигает трех. Программа сообщает, за сколько выстрелов был потоплен корабль.

Простое взаимодействие с игрой



Разбираем псевдокод

К планированию и написанию кода следует подходить методично. Мы начнем с написания *псевдокода.* Псевдокод представляет собой промежуточную стадию между кодом JavaScript и описанием программы на разговорном языке. Как вы вскоре увидите, он помогает понять логику работы программы еще до того, как вы займетесь разработкой *настоящего кода.*

В псевдокоде упрощенного варианта «Морского боя» имеется раздел с описанием переменных, и раздел с описанием логики программы. Раздел переменных сообщает, какие данные необходимо хранить в коде, а раздел логики — какой код нужно будет написать для создания игры.

ОБЪЯВИТЬ три *переменные* для хранения позиции каждой клетки корабля. Присвоить им имена `location1`, `location2` и `location3`.

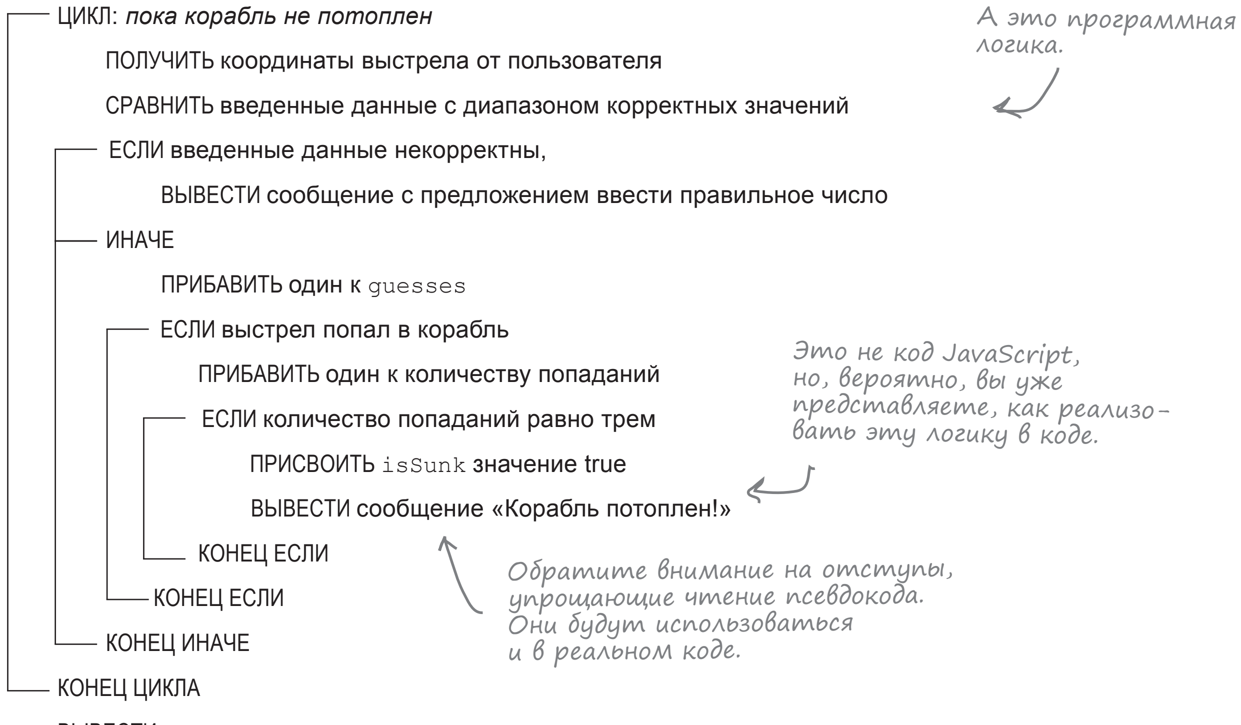
ОБЪЯВИТЬ *переменную* для номера текущей попытки. Присвоить ей имя `guess`.

ОБЪЯВИТЬ *переменную* для количества попаданий. Присвоить ей имя `hits` и *инициализировать* значением 0.

ОБЪЯВИТЬ *переменную* для количества попыток. Присвоить ей имя `guesses` и *инициализировать* значением 0.

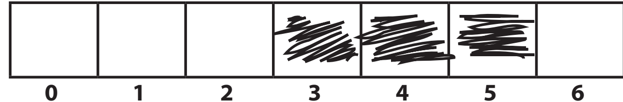
ОБЪЯВИТЬ *переменную* для хранения информации о том, потоплен корабль или нет. Присвоить ей имя `isSunk` и *инициализировать* значением `false`.

Переменные, которые нам понадобятся.



Возьми в руку карандаш

Допустим, виртуальное игровое поле выглядит так:



А для представления позиций клеток корабля используются переменные location:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Для тестирования будет использоваться следующая серия входных значений:

```
1, 4, 2, 3, 5
```

Теперь по псевдокоду с предыдущей страницы повторите каждый шаг программы и посмотрите, как она работает с введенным значением. Запишите результаты. Мы начали заполнять табличку, чтобы вам было проще. Если вы впервые имеете дело с псевдокодом, не торопитесь и внимательно разберитесь, как он работает.

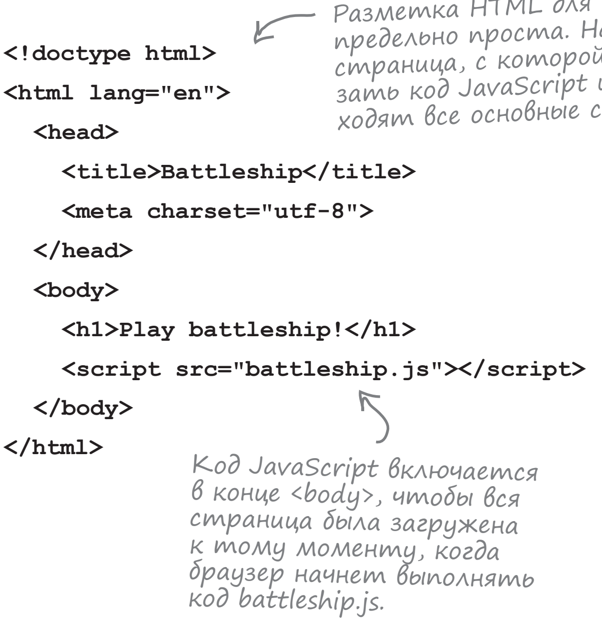
Если вам понадобится подсказка, загляните в конец главы.

location1	location2	location3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false

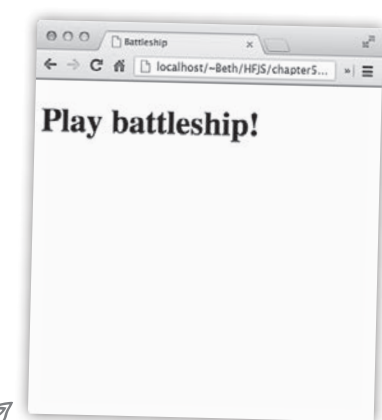
В первой строке приведены начальные значения переменных до того, как пользователь введет позицию первого выстрела. Переменная guess не инициализирована, поэтому ее значение не определено.

Стоп! Прежде чем идти дальше, вспомните про HTML!

Наш проект далеко не уйдет без разметки HTML, которую можно было бы связать с кодом. Введите приведенную ниже разметку и сохраните ее в новом файле с именем battleship.html. А когда все будет готово, можно вернуться к написанию кода.



```
<!doctype html>
<html lang="en">
  <head>
    <title>Battleship</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Play battleship!</h1>
    <script src="battleship.js"></script>
  </body>
</html>
```

Именно это вы увидите при загрузке страницы. Чтобы игра заработала, придется написать дополнительный код!

Напрягите извилины!

Возможно, мы немного забегаем вперед, но какой код вы бы написали для генерации случайной позиции корабля при каждой загрузке страницы? Какие факторы необходимо учесть для правильного размещения корабля? Запишите свои мысли по этому поводу.

Пишем код упрощенной версии «Морского боя»

Псевдокод послужит нам «эскизом» для написания настоящего кода JavaScript. Для начала определимся с переменными. Чтобы вспомнить, какие переменные нужны для программы «Морской бой», взгляните на псевдокод:

Нам понадобятся три переменные для хранения текущей позиции корабля.

ОБЪЯВИТЬ три *переменные* для хранения позиции каждой клетки корабля. Присвоить им имена `location1`, `location2` и `location3`.

ОБЪЯВИТЬ *переменную* для номера текущей попытки. Присвоить ей имя `guess`.

ОБЪЯВИТЬ *переменную* для количества попаданий. Присвоить ей имя `hits` и *инициализировать* значением 0.

ОБЪЯВИТЬ *переменную* для количества попыток. Присвоить ей имя `guesses` и *инициализировать* значением 0.

И еще три (guess, hits и guesses) для действий пользователя.

ОБЪЯВИТЬ *переменную* для хранения информации о том, потоплен корабль или нет. Присвоить ей имя `isSunk` и *инициализировать* значением `false`.

И наконец, еще одна переменная будет сообщать, потоплен корабль или нет.

Давайте определим эти переменные в файле JavaScript. Создайте новый файл с именем battleship.js и введите следующие объявления переменных:

```
var location1 = 3;
var location2 = 4;
var location3 = 5;
```

Три переменные location. Мы инициализируем их так, чтобы корабль размещался в клетках 3, 4 и 5 — просто на время.

А потом вернемся и напишем код, генерирующий случайную позицию корабля для усложнения задачи пользователя.

```
var guess;
var hits = 0;
var guesses = 0;
```

Переменная guess не важна, пока пользователь не введет координаты выстрела. До этого момента она будет иметь неопределенное значение undefined.

Переменным hits и guesses присваивается начальное значение 0.

```
var isSunk = false;
```

Наконец, переменной isSunk присваивается значение false. Она перейдет в состояние true, когда корабль пойдет на дно.

Для любознательных

Если вы не указали начальное значение переменной, JavaScript присваивает ей значение по умолчанию `undefined`, сообщая, что «переменная еще не инициализирована». `Undefined` и другие нестандартные значения позднее будут рассмотрены более подробно.

Переходим к реализации логики

С переменными разобрались, теперь пора заняться псевдокодом, описывающим непосредственную реализацию игры. Мы разобьем его на несколько частей. Начнем с цикла: он должен выполняться до тех пор, пока корабль не будет потоплен. Дальше нужно получить координаты выстрела у пользователя и проверить их (убедиться в том, что пользователь ввел число от 0 до 6). После этого нужно написать логику проверки попадания и убедиться в том, что корабль еще не пошел на дно. В завершение работы будет выводиться небольшой отчет с количеством попыток, которые понадобились игроку для того, чтобы потопить корабль.

```
ЦИКЛ: пока корабль не потоплен
    ПОЛУЧИТЬ координаты выстрела у пользователя
    СРАВНИТЬ введенные данные с диапазоном корректных значений
    ЕСЛИ введенные данные некорректны,
        ВЫВЕСТИ сообщение с предложением ввести правильное число
    ИНАЧЕ
        ПРИБАВИТЬ один к guesses
        ЕСЛИ выстрел попал в корабль
            ПРИБАВИТЬ один к количеству попаданий
            ЕСЛИ количество попаданий равно трем
                ПРИСВОИТЬ isSunk значение true
                ВЫВЕСТИ сообщение «Корабль потоплен!»
            КОНЕЦ ЕСЛИ
        КОНЕЦ ЕСЛИ
    КОНЕЦ ИНАЧЕ
КОНЕЦ ЦИКЛА
ВЫВЕСТИ статистику
```

ШАГ 1: Создание цикла, получение данных у пользователя и их проверка на корректность.

ШАГ 2: Проверяем результат выстрела. Попал? Мимо?

ШАГ 3: Проверяем, не затонул ли корабль.

ШАГ 4: Вывод завершающего сообщения для пользователя.

Что делать:

- [ ] Создать цикл и получить данные у пользователя
- [ ] Проверить на попадание
- [ ] Проверить, не был ли корабль потоплен
- [ ] Вывести результат для пользователя

Шаг 1: Создание цикла, получение данных

- ☐ Создать цикл и получить данные у пользователя
- ☐ Проверить на попадание
- ☐ Проверить, не был ли корабль потоплен
- ☐ Вывести результат для пользователя

Сейчас мы преобразуем описание логики игры в реальный код JavaScript. Процесс перевода псевдокода на JavaScript не идеален, там и сям попадаются небольшие поправки. Псевдокод дает хорошее представление о том, *что* должна делать программа; остается написать код JavaScript, который определит, *как* она должна это делать.

Возьмем весь код, который у нас имеется на данный момент, и сосредоточимся на добавляемых частях (чтобы сэкономить бумагу и спасти пару деревьев — или электронов, если вы читаете электронную версию книги).

ОБЪЯВИТЬ **переменные**

```
var location1 = 3;
var location2 = 4;
var location3 = 5;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;
```

Вообще-то эта часть уже рассматривалась, но мы приводим ее для полноты описания.

ЦИКЛ: *пока корабль не будет потоплен*

```
while (isSunk == false) {
```

Здесь начинается цикл. Пока корабль не будет потоплен, игра продолжается, а значит, продолжается и цикл.

Не забудьте, что для проверки продолжения цикла используется условие. В данном случае мы проверяем, что переменная isSunk все еще содержит false. Как только корабль будет потоплен, она станет true.

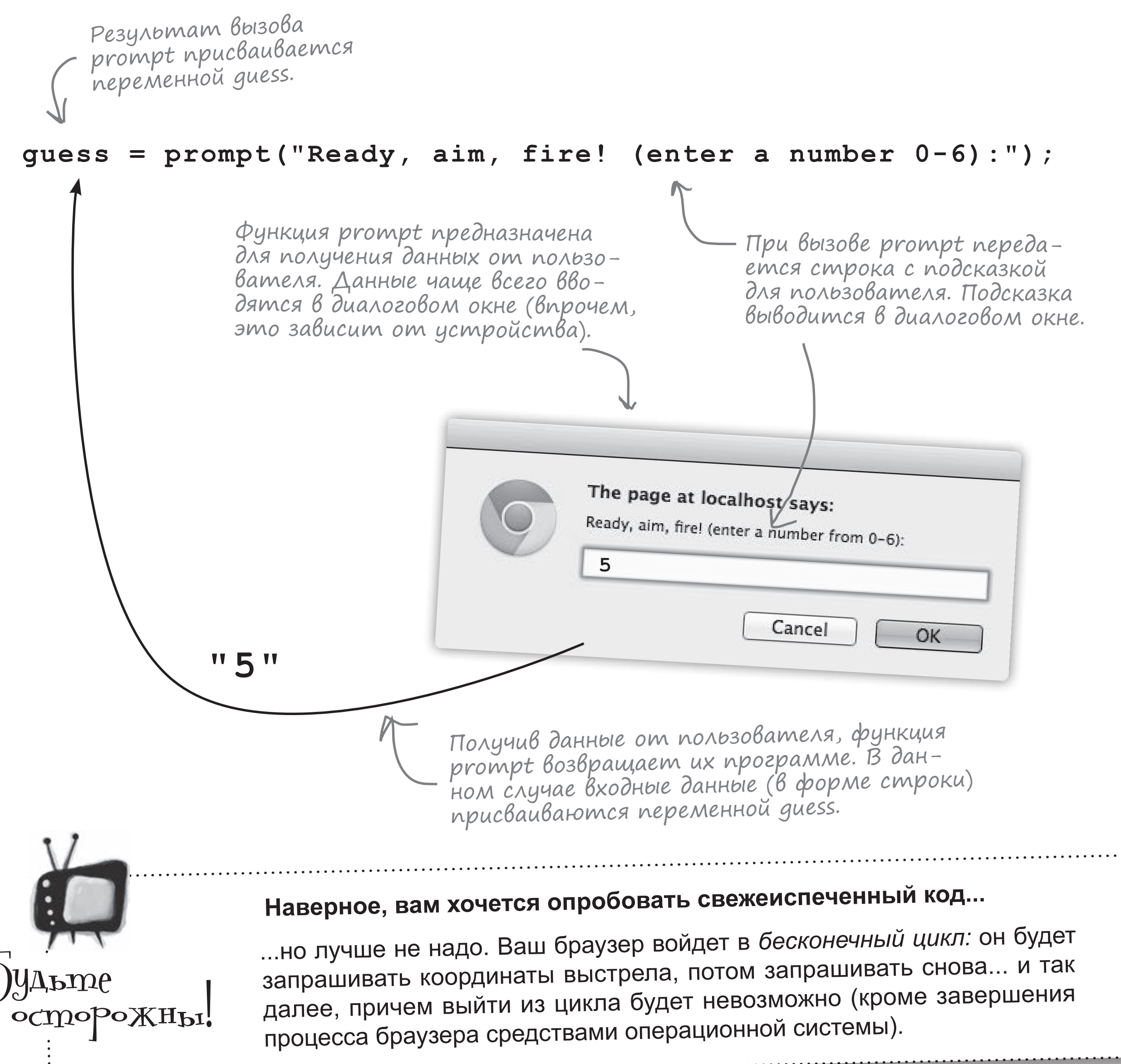
ПОЛУЧИТЬ координаты выстрела

```
    guess = prompt("Ready, aim, fire! (enter a number 0-6):");

}
```

При каждом выполнении цикла while мы запрашиваем у пользователя координаты выстрела. Для ввода данных используется встроенная функция prompt. Подробности на следующей странице...

Как работает функция prompt

Браузер предоставляет встроенную функцию `prompt`, которая предназначена для получения данных от пользователя. Функция `prompt` имеет много общего с уже использовавшейся функцией `alert`. Как и `alert`, `prompt` вызывает на экран диалоговое окно с переданной строкой, но предоставляет место, в котором пользователь может ввести ответ. Этот ответ в форме строки возвращается как результат вызова функции. Если пользователь закрыл диалоговое окно или ничего не ввел, функция возвращает `null`.



Будьте осторожны!

Наверное, вам хочется опробовать свежеиспеченный код...

...но лучше не надо. Ваш браузер войдет в *бесконечный цикл:* он будет запрашивать координаты выстрела, потом запрашивать снова... и так далее, причем выйти из цикла будет невозможно (кроме завершения процесса браузера средствами операционной системы).

Проверка на попадание

- [x] Создать цикл и получить данные у пользователя
- [ ] Проверить на попадание
- [ ] Проверить, не был ли корабль потоплен
- [ ] Вывести результат для пользователя

Из псевдокода видно, что перед проверкой на попадание необходимо убедиться, что пользователь ввел корректное значение. Если это так, можно переходить к следующей проверке — определить, попал ли выстрел в корабль. Нужно проследить за тем, чтобы переменные `guesses` и `hits` были обновлены по результатам проверки. Давайте проверим корректность введенного значения, а если все правильно, увеличим переменную `guesses`. А после этого мы напишем код, проверяющий, попал пользователь в цель или промахнулся.

```
// Объявления переменных


while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
       alert("Please enter a valid cell number!");
    } else {
       guesses = guesses + 1;
    }
}
```

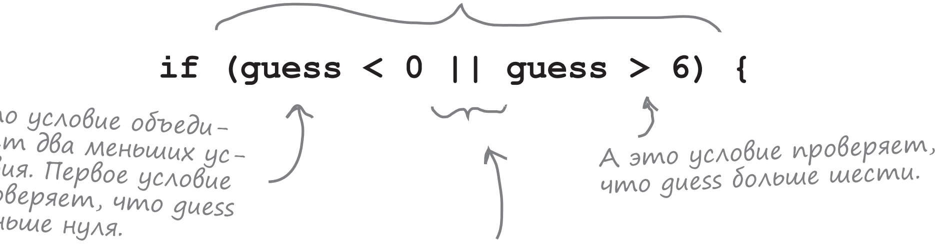
Чтобы проверить корректность ввода, убедимся, что значение в диапазоне от 0 до 6.

Если пользователь ввел недопустимое значение, сообщаем об этом функцией alert.

Если введенное значение корректно, увеличиваем guesses на 1 (в этой переменной хранится количество выстрелов пользователя).

Присмотримся повнимательнее к проверке корректности ввода. Да, мы проверяем, что значение лежит в диапазоне от 0 до 6, но как именно выполняется эта условная проверка? Рассмотрим ее по частям:

Результат будет «истиной» (true), если значение меньше нуля ИЛИ больше шести. Если выполняется хотя бы одно из условий, значит, введенное значение некорректно.



А эта конструкция называется оператором ИЛИ и объединяет две проверки, чтобы в случае истинности одной из них все "большое" условие было истинным. Если же оба меньших условия ложны, то и все выражение ложно, а значение guess лежит в диапазоне от 0 до 6, то есть оно корректно.

Часто задаваемые вопросы

В: В диалоговом окне prompt есть кнопка отмены. Какие данные вернет функция prompt, если пользователь нажмет эту кнопку?

О: Если нажать отмену в диалоговом окне, то вместо строки prompt вернет значение null. Помните, что null означает «нет значения». В данном случае это вполне уместно, потому что запрос был отменен. Мы можем воспользоваться тем, что prompt может возвращать null, и проверить, нажал ли пользователь кнопку отмены. Программа при этом, например, должна останавливать игру. В нашем коде такая проверка не выполняется, однако о данной возможности стоит помнить — мы еще воспользуемся ею позднее.

В: Вы сказали, что prompt всегда возвращает строку. Как же мы сравниваем строки — «0» или «6» — с числами 0 и 6?

О: В этой ситуации JavaScript автоматически преобразует строку введенных данных в число, чтобы сравнить guess < 0 и guess > 6. Если вы ввели только число (допустим, 4), JavaScript сможет преобразовать строку «4» в число 4. Позднее мы поговорим о преобразованиях типов.

В: Что произойдет, если пользователь введет вместо числа что-то другое — например, текст «six» или «quit»?

О: В этом случае JavaScript не сможет преобразовать строку в число. Программа будет сравнивать «six» с 6, или «quit» с 6 и вернет false. Попытка будет засчитана как промах. Полнофункциональная реализация сначала проверит пользовательский ввод и убедится в том, что было введено корректное число.

В: Результат операции ИЛИ будет истинным при истинности одного или другого условия, или истинными могут быть оба условия?

О: Да, истинными могут быть оба условия. Оператор ИЛИ (||) дает истинный результат при истинности хотя бы одного из двух условий. Результат ложен только в том случае, если ложны оба условия.

В: Наверное, существует и парный оператор И?

О: Да! Оператор И (&&) работает аналогичным образом, но дает истинный результат только в случае истинности обоих условий.

В: Что такое «бесконечный цикл»?

О: Резонный вопрос. Бесконечный цикл — одна из многочисленных проблем, преследующих неосторожных программистов. Как вам известно, работой цикла управляет условие, и цикл выполняется до тех пор, пока это условие остается истинным. Если в вашем коде не происходит ничего такого, в результате чего условие в какой-то момент станет ложным, то цикл никогда не завершится... Во всяком случае до того, как вы закроете браузер или перезагрузите систему.

Краткое руководство по булевским операторам

Булевские операторы используются в логических выражениях, результатом которых является значение true или false. Булевские операторы делятся на два вида: операторы сравнения и логические операторы.

Операторы сравнения

Операторы сравнения сопоставляют два значения:

- `<` “меньше”
- `>` “больше”
- `==` “равно”
- `===` “в точности равно” (мы еще вернемся к этому оператору!)
- `<=` “меньше либо равно”
- `>=` “больше либо равно”
- `!=` “не равно”

Логические операторы

Логические операторы объединяют два булевских выражения и создают один результат true или false. Два важнейших логических оператора:

- `||` ИЛИ. Результат будет истинным, если истинно *хотя бы одно* из двух выражений.
- `&&` И. Результат будет истинным, если истинны *оба* выражения.

Еще один логический оператор — НЕ — применяется к одному булевскому выражению (вместо двух):

- `!` НЕ. Результат будет истинным, если выражение ложно.

Ну что, попал?

- [x] Создать цикл и получить данные у пользователя
- [ ] Проверить на попадание
- [ ] Проверить, не был ли корабль потоплен
- [ ] Вывести результат для пользователя

А сейчас начинается самое интересное — пользователь выдал предположение о положении корабля, а программа должна определить, угадал он или нет. Нужно проверить, принадлежит ли заданная клетка числу клеток, занимаемых кораблем. Если выстрел оказался удачным, мы увеличиваем переменную `hits`.

Ниже приведена первая версия кода проверки попаданий; давайте последовательно разберем ее:

```
if (guess == location1) {
    hits = hits + 1;
} else  if (guess == location2) {
    hits = hits + 1;
} else if (guess == location3) {
    hits = hits + 1;
}
```

Если выстрел пришелся в клетку location1, произошло попадание — переменная hits увеличивается на 1.

В противном случае, если выстрел пришелся в клетку location2, делаем то же самое.

Наконец, если выстрел пришелся в клетку location3, также увеличиваем переменную hits.

А если ни одно из условий не выполняется, то переменная hits не увеличивается.

Обратите внимание на отступы в блоках if/else. Они упрощают чтение кода, особенно при большом количестве вложенных блоков.

Возьми в руку карандаш

Что вы думаете о первом варианте кода проверки попадания? Не выглядит ли он сложнее, чем это необходимо? Не кажется ли вам, что он... несколько однообразен? Нельзя ли его упростить? Попробуйте сделать это, используя информацию об операторе || (логический оператор ИЛИ). *Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.*

Добавление кода проверки попадания

- [x] Создать цикл и получить данные у пользователя
- [x] Проверить на попадание
- [ ] Проверить, не был ли корабль потоплен
- [ ] Вывести результат для пользователя

А теперь соберем воедино все, о чем говорилось на паре последних страниц:

ЦИКЛ: *пока корабль не потоплен*

ПОЛУЧИТЬ координаты выстрела от пользователя

ПРИБАВИТЬ один к guesses

ЕСЛИ выстрел попал в корабль

ПРИБАВИТЬ один к количеству попаданий

```
// Объявления переменных

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      alert("Please enter a valid cell number!");
    } else {
      guesses = guesses + 1;

      if (guess == location1 || guess == location2 || guess == location3) {
          hits = hits + 1;
      }
    }
}
```

Проверяем предположение пользователя...

Похоже, введенное значение корректно; увеличиваем переменную guesses на 1.

Если выстрел пришелся в одну из клеток корабля, увеличиваем счетчик hits.

Три проверки были объединены в одну команду if оператором || (ИЛИ). Это условие читается так: «Если значение guess равно location1 ИЛИ значение guess равно location2 ИЛИ значение guess равно location3, увеличить счетчик hits».

Да тебе просто повезло!

- [x] Создать цикл и получить данные у пользователя
- [x] Проверить на попадание
- [x] Проверить, не был ли корабль потоплен
- [ ] Вывести результат для пользователя

Работа почти закончена; логика практически полностью реализована. Снова обращаясь к псевдокоду, мы видим, что теперь нужно проверить, достигло ли количество попаданий трех. В этом случае корабль потоплен. Тогда программа присваивает переменной `isSunk` значение true, а пользователь получает сообщение о том, что корабль уничтожен. Рассмотрим предварительную версию кода, прежде чем добавлять его в программу.

Сначала проверяем, что в корабль было произведено три попадания.

```
if (hits == 3) {
    isSunk = true;
    alert("You sank my battleship!");
}
```

Если так, то переменной isSunk присваивается значение true.

Сообщаем об этом пользователю!

Еще раз взгляните на приведенный выше цикл while. Что произойдет, если переменная isSunk равна true?

Вывод данных после игры

- ☑ Создать цикл и получить данные у пользователя
- ☑ Проверить на попадание
- ☑ Проверить, не был ли корабль потоплен
- ☑ Вывести результат для пользователя

Переменной `isSunk` присваивается значение true, и выполнение цикла while прекращается. Программа, в которую мы вложили столько сил, перестает выполнять тело цикла, а игра подходит к концу. Но нужно еще вывести информацию о том, насколько хорошо пользователь справился со своей задачей:

```
var stats = "You took " + guesses + " guesses to sink the battleship, " +
                "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Здесь создается строка с сообщением, включающим информацию о количестве выстрелов и точности. Обратите внимание: строка разбивается на фрагменты (для добавления переменной guesses и удобства чтения), которые сцепляются оператором конкатенации «+». Пока введите код в приведенном виде, более подробное описание будет дано далее.

Добавьте этот фрагмент и проверку уничтожения корабля в основную программу:

```
// Объявления переменных

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;

            if (hits == 3) {

                isSunk = true;

                alert("You sank my battleship!");
            }
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
                "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

ЦИКЛ: *пока корабль не будет потоплен*

ПОЛУЧИТЬ координаты выстрела

ПРИБАВИТЬ один к guesses

ЕСЛИ выстрел попал в корабль

ПРИБАВИТЬ один к количеству попаданий

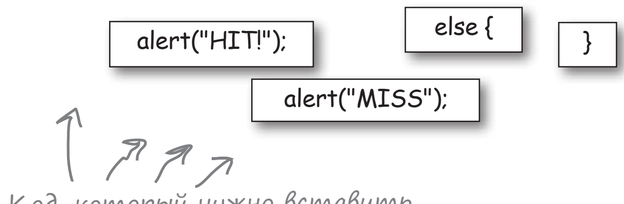
ЕСЛИ количество попаданий равно трем

ПРИСВОИТЬ isSunk значение true

ВЫВЕСТИ сообщение «Корабль потоплен!»

ВЫВЕСТИ результат для пользователя

Помните, мы говорили о том, что псевдокод часто бывает неидеальным? В исходном псевдокоде мы забыли об одной мелочи: мы не сообщаем пользователю, попал он в корабль (HIT) или промахнулся (MISS). Сможете ли вы расставить эти фрагменты кода в правильные места?



Код, который нужно вставить.

```
// Объявления переменных

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;
            if (hits == 3) {
              isSunk = true;
              alert("You sank my battleship!");
            }
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Много, много фигурных скобок. Если вы запутаетесь, какая из них к чему относится, просто проведите вертикальную линию прямо в книге.

Реализация логики готова!

- [x] Создать цикл и получить данные у пользователя
- [x] Проверить на попадание
- [x] Проверить, не был ли корабль потоплен
- [x] Вывести результат для пользователя

Порядок! Наш псевдокод полностью преобразован в код JavaScript. Мы даже обнаружили то, что было упущено в исходном варианте псевдокода, и добавили реализацию этой возможности в программу. Ниже приведен полный код программы. Введите его и сохраните в файле battleship.js:

```
var location1 = 3;
var location2 = 4;
var location3 = 5;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        } else {
           alert("MISS");
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Немного о контроле качества

«Контролем качества» называется процесс тестирования программного продукта с целью поиска дефектов. Сейчас мы проведем небольшой сеанс контроля качества для нашего кода. Загрузите файл battleship.html в браузере и начните игру. Немного поэкспериментируйте. Все работает идеально? Или вам удастся найти какие-то недостатки? Если удастся, запишите их справа. Пример нашего тестового запуска показан ниже на этой странице.

Для заметок

Запишите все, что, по вашему мнению, работает не так, как должно, — или что можно улучшить.

Вот как выглядит наше взаимодействие с игрой.

The page at localhost says:
Please enter a valid cell number!
OK

Сначала мы ввели некорректное число 9.

The page at localhost says:
Please enter a valid cell number!
OK

The page at localhost says:
Ready, aim, fire! (enter a number from 0–6):
0
Cancel OK

Потом ввели 0, чтобы получить промах.

The page at localhost says:
MISS
OK

The page at localhost says:
Ready, aim, fire! (enter a number from 0–6):
3
Cancel OK

А потом — целых три попадания подряд!

The page at localhost says:
HIT!

The page at localhost says:
Ready, aim, fire! (enter a number from 0–6):
4
Cancel OK

The page at localhost says:
HIT!

The page at localhost says:
Ready, aim, fire! (enter a number from 0–6):
5
Cancel OK

The page at localhost says:
HIT!
OK

На третьем (и последнем) попадании корабль был потоплен.

The page at localhost says:
You sank my battleship!
OK

Мы видим, что для победы понадобилось 4 выстрела с общей точностью 0.75.

The page at localhost says:
You took 4 guesses to sink the battleship, which means your shooting accuracy was 0.75
OK

Логика игры в общем-то понятна... кроме булевских операторов. Для чего они нужны? Просто для группировки условий?

Булевские операторы позволяют формулировать более сложные логические утверждения.

Вы уже видели достаточно условных конструкций, чтобы, допустим, проверить, что температура больше 32 градусов. Или что переменная inStock, представляющая наличие товара на складе, истинна. Но иногда требуется проверять более сложные условия — допустим, что значение не только больше 32, но и меньше 100. Или что вместе с признаком наличия товара inStock установлен признак распродажи onSale. Или что товар доступен по льготной цене для пользователей с привилегированной учетной записью. Как видите, условия бывают довольно сложными.

Рассмотрим несколько примеров, чтобы вы лучше поняли, как работают булевские операторы.

Допустим, нужно проверить, что товар имеется в наличии (inStock) И участвует в распродаже (onSale). Это можно сделать так:

```
if (inStock == true) {
    if (onSale == true) {
        // sounds like a bargain!
        alert("buy buy buy!");
    }
}
```

Сначала проверяем, что товар имеется в наличии...

А если есть — что он также участвует в распродаже.

И если оба условия выполняются, тогда совершаем действие — например, покупаем!

Важно: этот код выполняется только в том случае, если истинны оба условия!

Этот код можно упростить объединением двух условий. В отличие от «Морского боя», где мы проверяли, что guess < 0 ИЛИ guess > 6, мы хотим знать, что переменная inStock истинна, И переменная onSale тоже истинна. Как же это делается?

Оператор И: объединенное условие истинно только в том случае, если истинны обе части, первая И вторая.

```
if (inStock == true  &&  onSale == true) {
        // Выгодное предложение!
        alert("buy buy buy!");
}
```

Этот код не только более компактен, но и лучше читается. Сравните с кодом на предыдущей странице и убедитесь сами.

Это еще не все; булевские операторы можно объединять самыми разнообразными способами:

Здесь операторы И и ИЛИ задействованы в одном условном выражении. Это выражение означает следующее: если товар имеется в наличии И участвует в распродаже, ИЛИ цена меньше 60 — покупаем.

```
if (inStock == true  &&  (onSale == true || price < 60)) {
        // Выгодное предложение!!
        alert("buy buy buy!");
}
```

Обратите внимание: круглые скобки группируют условия, чтобы сначала был вычислен результат операции ИЛИ, а потом этот результат был использован для вычисления результата И.

Возьми в руку карандаш

Мы подготовили несколько булевских выражений, значения которых необходимо вычислить. Заполните пропуски и сверьтесь с ответами в конце главы, прежде чем двигаться дальше.

```
var temp = 81;
var willRain = true;
var humid = (temp > 80 && willRain == true);
```

Чему равно значение **humid**? ________________

```
var guess = 6;
var isValid = (guess >= 0 && guess <= 6);
```

Чему равно значение **isValid**? ________________

```
var kB = 1287;
var tooBig = (kB > 1000);
var urgent = true;
var sendFile =
    (urgent == true || tooBig == false);
```

Чему равно значение **sendFile**? ________________

```
var keyPressed = "N";
var points = 142;
var level;
if (keyPressed == "Y" ||
     (points > 100 && points < 200)) {
   level = 2;
} else {
   level = 1;
}
```

Чему равно значение **level**? ______________

Бухгалтеры Боб и Билл работают над новым приложением для веб-сайта своей компании (приложение проверяет цену и выдает рекомендацию для покупки). Оба они написали команды if/else с использованием булевских выражений. Каждый уверен, что его код правилен. Так кто же из них прав? А кому лучше заниматься бухгалтерией, а не программированием? Сверьтесь с ответами в конце главы.

```
if (price < 200 || price > 600) {
    alert("Price is too low or too high! Don't buy the gadget.");
} else {
    alert("Price is right! Buy the gadget.");
}
```

Билл

```
if (price >= 200 || price <= 600) {
    alert("Price is right! Buy the gadget.");
} else {
    alert("Price is too low or too high! Don't buy the gadget.");
}
```

А нельзя ли покороче...

Даже не знаем, как бы это сказать, но... вы излишне многословны в определении своих условий. Что мы имеем в виду? Для примера возьмем хотя бы это условие:

В условиях if булевские переменные часто проверяются на истинность или ложность.

```
if  (inStock == true) {
   ...
}
```

Переменная inStock может содержать одно из двух значений: true или false.

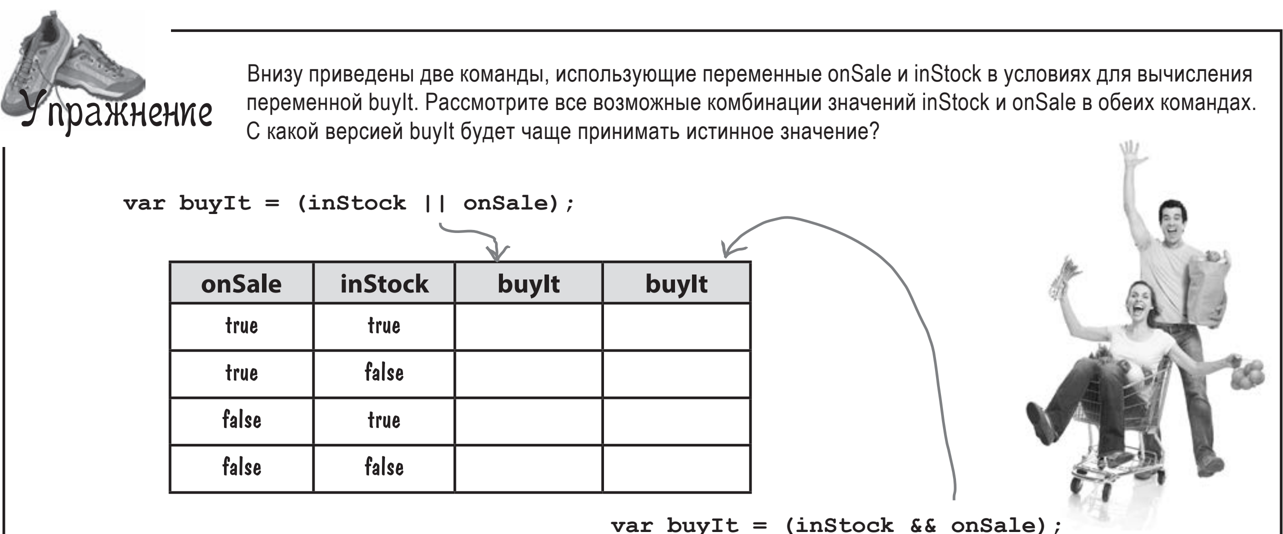
Как выясняется, такое сравнение немного избыточно. Вся суть условия заключается в том, что его результатом является значение true или false, но наша булевская переменная `inStock` уже содержит одно из этих значений. Следовательно, сравнивать ее ни с чем не нужно; достаточно просто указать ее в условии саму по себе. Иначе говоря, можно использовать запись следующего вида:

```
if  (inStock) {
   ...
}
```

Если просто указать логическую переменную саму по себе, то в случае истинности этой переменной условие тоже будет истинным, а блок будет выполнен.

А если переменная inStock ложна, то и условие ложно, и программный блок пропускается.

Возможно, кто-то скажет, что исходная, длинная версия более четко выражала намерения программиста, но на практике чаще применяется более компактная запись. Кроме того, компактная запись еще и проще читается.



Упражнение

Внизу приведены две команды, использующие переменные onSale и inStock в условиях для вычисления переменной buyIt. Рассмотрите все возможные комбинации значений inStock и onSale в обеих командах. С какой версией buyIt будет чаще принимать истинное значение?

```
var buyIt = (inStock || onSale);
```

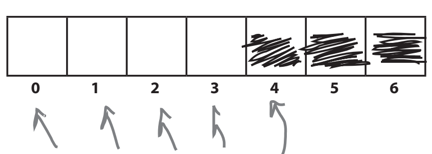
onSale	inStock	buyIt	buyIt
true	true		
true	false		
false	true		
false	false		

```
var buyIt = (inStock && onSale);
```

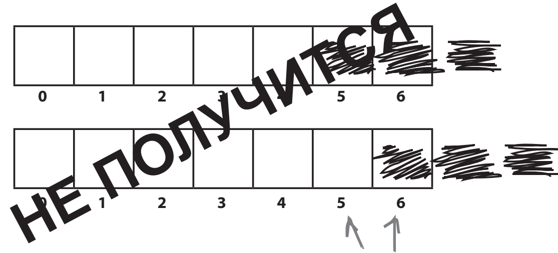
Упрощенный «Морской бой» почти готов

Да, но осталось решить еще одну маленькую проблему — сейчас местонахождение корабля жестко зафиксировано в программе, и сколько бы раз вы ни играли, корабль всегда находится в клетках 3, 4 и 5. Для тестирования такой подход работает неплохо, но чтобы игра была чуть более интересной для пользователя, корабль должен размещаться случайным образом.

Давайте немного отступим и подумаем, как правильно разместить корабль в одномерной таблице из семи ячеек. Нам понадобится начальная позиция, от которой будут отсчитываться три смежные клетки. Это означает, что начальная клетка должна лежать в диапазоне от 0 до 4.



Мы можем начать с клеток 0, 1, 2, 3 и 4, и у нас все еще останется место для размещения корабля в трех клетках.



Но начинаться с клетки 5 или 6 корабль не может.

Как получить случайную позицию

Если у нас уже имеется начальная позиция randomLoc (от 0 до 4), то корабль просто размещается в этой и двух последующих клетках.

```
var location1 = randomLoc;
var location2 = location1 + 1;
var location3 = location2 + 1;
```

Используем случайную позицию с двумя следующими смежными клетками.

Замечательно, но как сгенерировать случайное число? В этом нам поможет JavaScript и встроенные функции. Дело в том, что в JavaScript уже имеется набор готовых функций для выполнения стандартных математических операций, и этот набор включает пару функций для генерирования случайных чисел. Встроенные функции (и функции вообще) будут более подробно описаны позднее. А сейчас мы просто воспользуемся ими для решения конкретной задачи.

Всемирно известный рецепт генерирования случайных чисел

Начнем с функции `Math.random`. Вызывая эту функцию, вы получаете случайное число:

Наша переменная. Мы хотим присвоить ей значение от 0 до 4.

Math.random является частью стандартного инструментария JavaScript и возвращает случайное число.

```
var randomLoc = Math.random();
```

Единственная проблема заключается в том, что функция возвращает числа из диапазона от 0 до 1 (не включая 1) — такие, как 0.128, 0.830, 0.9 или 0.42. Значит, на основании этих чисел нужно будет как-то сгенерировать целые случайные числа от 0 до 4.

Нам нужны целые числа от 0 до 4 — то есть 0, 1, 2, 3 или 4, а не дробные числа вида 0,34. Для начала можно умножить число, возвращаемое `Math.random`, на 5; это позволит нам приблизиться шаг к решению задачи. Вот что мы имеем в виду...

Умножая случайное число на 5, мы получаем число в диапазоне от 0 до 5, не включая 5, — такие числа, как 0.13983, 4.231, 2.3451 или, допустим, 4.999.

```
var randomLoc = Math.random() * 5;
```

Помните: звездочка (*) обозначает умножение.

Уже лучше! Остается отсечь дробную часть, чтобы получить целое число. Для этого можно воспользоваться другой встроенной математической функцией Math.floor:

Функция Math.floor округляет числа, отбрасывая их дробную часть.

```
var randomLoc = Math.floor(Math.random() * 5);
```

Так, например, 0.13983 превращается в 0, 2.34 превращается в 2, а 4.999 превращается в 4.

Часто задаваемые вопросы

В: Если мы хотим сгенерировать число от 0 до 4, то почему в программе встречается число 5: `Math.floor(Math.random() * 5)`**?**

О: Хороший вопрос. Прежде всего, функция Math.random генерирует число от 0 до 1, не включая 1. Максимальное число, которое можно получить от Math.random, равно 0.999... Максимальное число, которое можно получить при умножении его на 5, равно 4.999... Функция Math.floor округляет числа в меньшую сторону, поэтому 1.2 округляется до 1, но точно так же округляется и 1.9999. Если мы генерируем числа от 0 до 4.999... то все результаты будут округляться до чисел от 0 до 4. Это не единственный способ, и в других языках данная задача нередко решается иначе, но в коде JavaScript чаще всего встречается именно такое решение.

В: Выходит, если мне понадобится случайное число от 0 до 100 (включая 100), я должен написать `Math.floor(Math.random() * 101)`**?**

О: Точно! Умножение на 101 и отсечение дробной части вызовом Math.floor гарантирует, что результат не превысит 100.

В: Зачем нужны круглые скобки в Math.random()?

О: Круглые скобки используются при «вызове» функции. Иногда функции нужно передать значение (как в случае с функцией alert, которой нужно передать сообщение), а иногда дополнительная информация не нужна, как в случае с Math.random. Но каждый раз, когда вы вызываете функцию (встроенную или нет), нужно использовать круглые скобки. Пока не беспокойтесь об этом; эта тема рассматривается в следующей главе.

В: Моя игра не работает. На веб-странице не выводится ничего, кроме заголовка «Play battleship». Как узнать, что я сделал не так?

О: В таких случаях может пригодиться консоль. Если вы допустили ошибку (скажем, забыли поставить кавычку в строке), JavaScript обычно пожалуется на недопустимый синтаксис и даже приведет номер строки с ошибкой. Иногда ошибки оказываются более хитрыми. Например, если вы по ошибке написали isSunk = false вместо isSunk == false, сообщение об ошибке не выводится, но программа работает совершенно не так, как предполагалось. Попробуйте воспользоваться функцией console.log для вывода значений переменных в разных точках кода. Возможно, вам удастся обнаружить ошибку.

Для заметок

Возвращаемся к контролю качества

Все необходимое готово. Давайте соберем код воедино (мы уже сделали это ниже) и заменим существующий фрагмент размещения корабля. Когда все будет сделано, проведите несколько пробных запусков и посмотрите, как быстро вам удастся потопить врага.

```
var randomLoc = Math.floor(Math.random() * 5);
var location1 = randomLoc;
var location2 = location1 + 1;
var location3 = location2 + 1;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      // Остальной код....
```

Замените объявления переменных location этими новыми командами.

Один из сеансов тестирования. Теперь, когда позиция корабля неизвестна заранее, игра стала чуть более интересной. Тем не менее результат все равно выглядит довольно впечатляюще...

Battleship

localhost/~Beth/HFJS/chapter5...

Play battleship!

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
3
Cancel OK

Попадание с первого выстрела.

The page at localhost says:
HIT!
OK

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
4
Cancel OK

Второй выстрел оказывается менее успешным.

The page at localhost says:
MISS
OK

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
2
Cancel OK

А потом два попадания подряд.

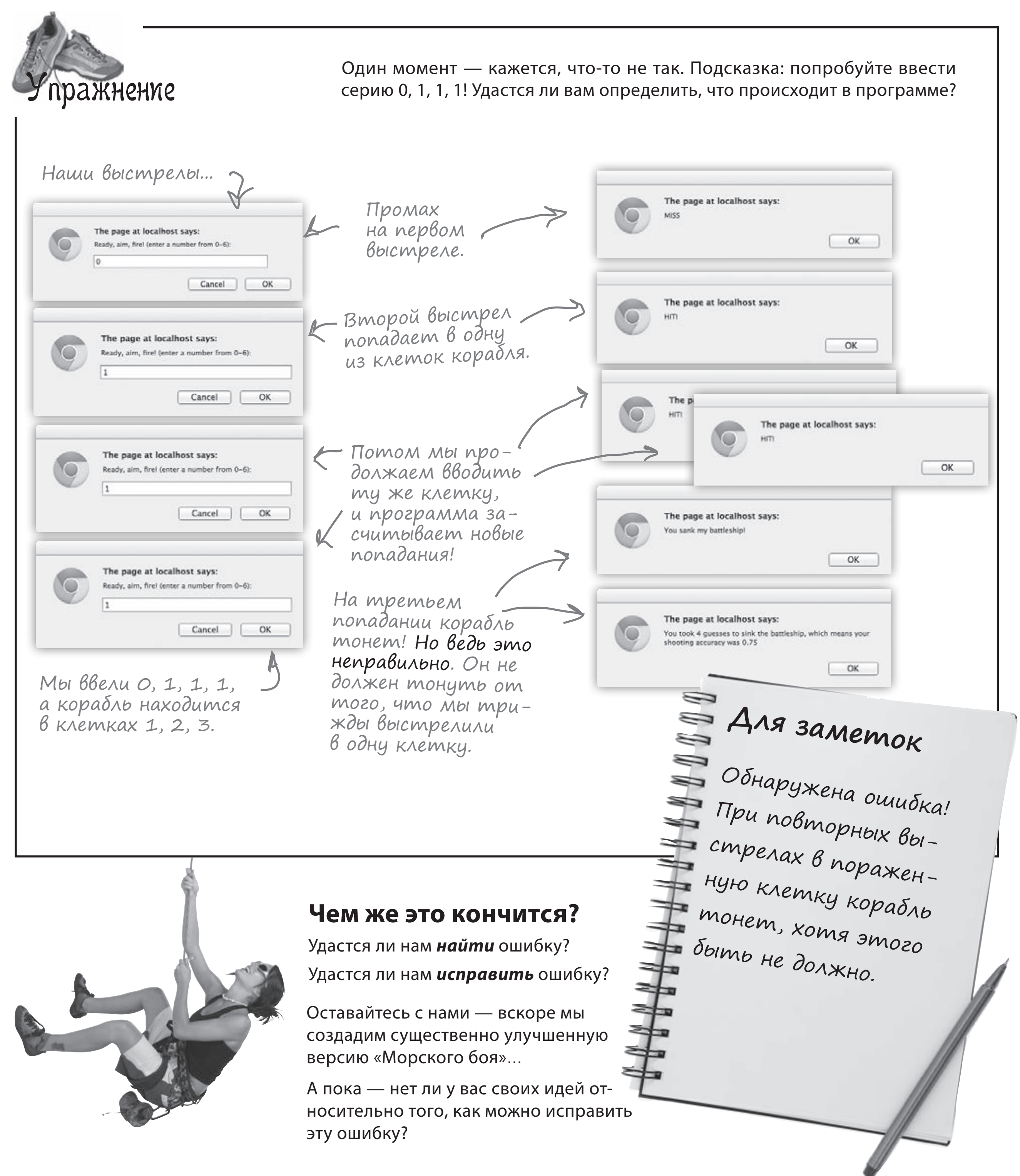
The page at localhost says:
HIT!

The page at localhost says:
You sank my battleship!
OK

The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
1
Cancel OK

Последнее попадание, корабль потоплен!

The page at localhost says:
You took 4 guesses to sink the battleship, which means your shooting accuracy was 0.75
OK



Один момент — кажется, что-то не так. Подсказка: попробуйте ввести серию 0, 1, 1, 1! Удастся ли вам определить, что происходит в программе?

Чем же это кончится?

Удастся ли нам ***найти*** ошибку?

Удастся ли нам ***исправить*** ошибку?

Оставайтесь с нами — вскоре мы создадим существенно улучшенную версию «Морского боя»...

А пока — нет ли у вас своих идей относительно того, как можно исправить эту ошибку?

Поздравляем, вы создали свою первую программу на JavaScript! Теперь пара слов о повторном использовании кода

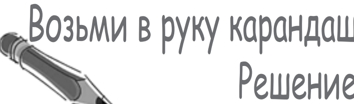
Возможно, вы обратили внимание на использование таких встроенных функций, как `alert`, `prompt`, `console.log` и `Math.random`. При минимальных усилиях с нашей стороны эти функции позволяют, словно по волшебству, открывать диалоговые окна, выводить данные на консоль и генерировать случайные числа. Но встроенные функции представляют собой нечто иное, как готовый код, который был заранее написан за вас. Программа наглядно демонстрирует их мощь, которая проявляется как раз в возможности многократного использования. От вас потребуется совсем немного: вызвать функцию тогда, когда она вам понадобится.

Впрочем, чтобы пользоваться функциями, нужно достаточно много знать о них, о том, как они должны вызываться, какие значения им следует передавать и так далее. Всем этим мы займемся в следующей главе, когда будем учиться создавать собственные функции.

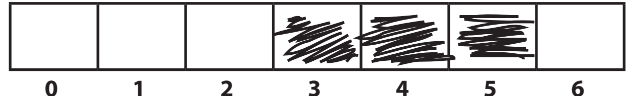
А пока ознакомьтесь со списком ключевых моментов этой главы... и хорошенько выспитесь, чтобы новые знания улеглись у вас в голове.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Для описания логики программы JavaScript можно воспользоваться блок-схемой, на которой представлены точки принятия решений и выполняемые действия.
- Прежде чем переходить к написанию кода, бывает полезно создать эскиз программы на псевдокоде.
- **Псевдокод** представляет собой приближенное представление того, что должен делать ваш код.
- Булевские операторы делятся на две категории: операторы сравнения и логические операторы. При использовании в выражении булевские операторы дают результат true или false.
- Операторы **сравнения** сравнивают две величины и возвращают true или false. Например, оператор сравнения < («меньше») может использоваться в выражении 3 < 6 (результат — true).
- **Логические** операторы сравнивают логические значения. Например, true || false дает true, а true && false дает false.
- Функция **Math.random** генерирует случайное число в диапазоне от 0 до 1 (не включая 1).
- Функция **Math.floor** округляет число в сторону уменьшения до ближайшего целого.
- При использовании функций Math.random и Math.floor имена должны начинаться с прописной буквы M.
- Функция JavaScript **prompt** выводит диалоговое окно с сообщением и полем, в котором пользователь вводит запрашиваемое значение.
- В этой главе метод prompt используется для получения данных у пользователя, а метод alert — для вывода результатов игры в браузере.



Допустим, виртуальное игровое поле выглядит так:



А для представления позиций клеток корабля используются переменные location:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Для тестирования будет использоваться следующая серия входных значений:

```
1, 4, 2, 3, 5
```

Теперь по псевдокоду на предыдущей странице повторите каждый шаг программы и посмотрите, как он работает с введенным значением. Запишите результаты внизу. Мы начали заполнять таблицу, чтобы вам было проще. Ниже приведено наше решение.

location1	location2	location 3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false
3	4	5	4	2	1	false
3	4	5	2	3	1	false
3	4	5	3	4	2	false
3	4	5	5	5	3	true

Упражнение
Решение

Внизу приведены две команды, использующие переменные onSale и inStock в условиях для вычисления переменной buyIt. Рассмотрите все возможные комбинации значений inStock и onSale в обеих командах. С какой версией buyIt будет чаще принимать истинное значение? (С оператором ИЛИ (||)!)

```
var buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

```
var buyIt = (inStock && onSale);
```

Возьми в руку карандаш Решение

Мы подготовили несколько булевских выражений, значения которых необходимо вычислить. Заполните пропуски. Ниже приведено наше решение:

```
var temp = 81;
var willRain = true;
var humid = (temp > 80 && willRain == true);
```

Чему равно значение **humid**? *true*

```
var guess = 6;
var isValid = (guess >= 0 && guess <= 6);
```

Чему равно значение **isValid**? *true*

```
var kB = 1287;
var tooBig = (kB > 1000);
var urgent = true;
var sendFile =
    (urgent == true || tooBig == false);
```

Чему равно значение **sendFile**? *true*

```
var keyPressed = "N";
var points = 142;
var level;
if (keyPressed == "Y" ||
     (points > 100 && points < 200)) {
   level = 2;
} else {
   level = 1;
}
```

Чему равно значение **level**? *2*

Бухгалтеры Боб и Билл работают над приложением для веб-сайта своей компании (приложение проверяет цену и выдает рекомендацию для покупки). Оба они написали команды if/else с использованием булевских выражений. Каждый уверен, что его код правилен. Так кто же из них прав? А кому лучше заниматься бухгалтерией, а не программированием? Ниже приведено наше решение.

Боб

```
if (price < 200 || price > 600) {
   alert("Price is too low or too high! Don't buy the gadget.");
} else {
   alert("Price is right! Buy the gadget.");
}
```

Билл

```
if (price >= 200 || price <= 600) {
   alert("Price is right! Buy the gadget.");
} else {
   alert("Price is too low or too high! Don't buy the gadget.");
}
```

Из Боба получился лучший программист (а может, и бухгалтер). Решение Боба работает, а решение Билла — нет. Чтобы понять, почему, проверим условия Боба и Билла с тремя разными значениями цены (слишком низкое, слишком высокое и правильное), и посмотрим, какие результаты будут получены:

price	Bob's	Bill's
100	true alert: Не покупать!	true alert: Покупать!
700	true alert: Не покупать!	true alert: Покупать!
400	false alert: Buy!	true alert: Покупать!

Допустим, price = 100. Значение 100 меньше 200, поэтому условие Боба истинно (помните: в ИЛИ для истинности всего выражения достаточно истинности одного условия), и программа выводит рекомендацию «Не покупать!»

Но условие Билла окажется истинным, потому что price <= 600! Таким образом, результат всего выражения оказывается истинным, и программа сообщает «Покупать!» несмотря на слишком низкую цену.

Как выясняется, условие Билла всегда истинно независимо от price, поэтому код всегда выдает рекомендацию «Покупать!». Пожалуй, Биллу лучше заниматься бухгалтерией, а не программированием.

Помните, мы говорили о том, что псевдокод часто бывает неидеальным? В исходном псевдокоде мы забыли об одной мелочи: мы не сообщаем пользователю, попал он в корабль (HIT) или промахнулся (MISS). Сможете ли вы расставить эти фрагменты кода в правильные места? Ниже приведено наше решение:

```
// Объявления переменных

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
              isSunk = true;
              alert("You sank my battleship!");
            }
        } else {
            alert("MISS");
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Возьми в руку карандаш
Решение

Что вы думаете о первом варианте кода проверки попадания? Не выглядит ли он сложнее, чем это необходимо? Не кажется ли вам, что он... несколько однообразен? Нельзя ли его упростить? Попробуйте сделать это, используя информацию об операторе || (логический оператор ИЛИ). *Ниже приведено наше решение.*

```
if (guess == location1) {
    hits = hits + 1;
} else  if (guess == location2) {
    hits = hits + 1;
} else if (guess == location3) {
    hits = hits + 1;
}
```

Один и тот же код используется снова и снова.

Если нам когда-нибудь потребуется изменить способ обновления hits, коррекцию придется делать сразу в трех местах. Такие ситуации часто становятся причинами ошибок в коде.

Да, и этот код сложнее, чем необходимо... И он хуже читается, чем хотелось бы... А чтобы его набрать, программисту придется изрядно потрудиться и подумать.

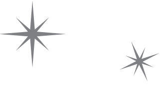
С логическим оператором ИЛИ условия можно объединить. Если выстрел попадает в любую из клеток location1, location2 или location3, то условие будет истинным, и переменная hits будет обновлена.

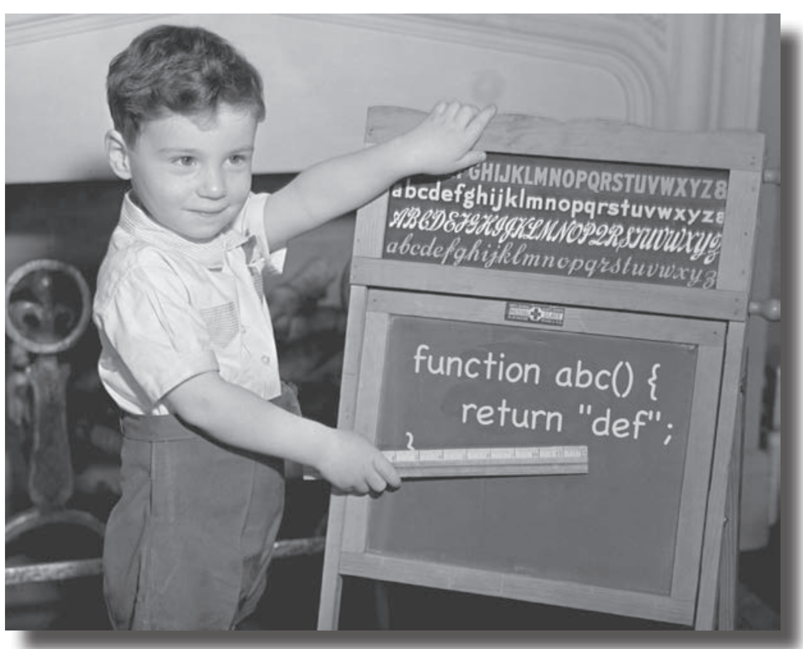
```
if (guess == location1 || guess == location2 || guess == location3) {
    hits = hits + 1;
}
```

Согласитесь, выглядит намного приятнее? Не говоря уже о том, что код становится более понятным.

И если нам когда-нибудь потребуется изменить способ обновления hits, изменения будут вноситься в одном месте, что существенно снижает риск ошибки.

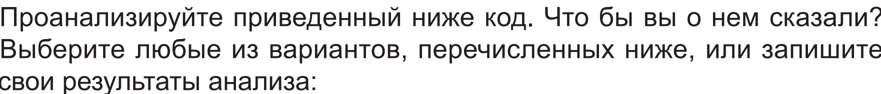
Функции для всех





В этой главе вы овладеете своей первой суперспособностью. Вы уже кое-что знаете о программировании; пришло время сделать следующий шаг и освоить работу с функциями. Функции позволяют писать код, который может **повторно использоваться** в разных ситуациях; код, существенно более **простой в сопровождении**; код, который можно **абстрагировать** и присвоить ему простое имя, чтобы вы могли забыть о рутинных подробностях и заняться действительно важными делами. Вы увидите, что функции не только открывают путь к мастерству программиста, но и играют ключевую роль в стиле программирования JavaScript. В этой главе мы начнем с основ: механики и всех тонкостей работы функций, а потом в оставшейся части книги будем совершенствовать ваши навыки работы с функциями. Итак, начнем с азов... *прямо сейчас*.

В книге эта тема будет рассматриваться более подробно.



Проанализируйте приведенный ниже код. Что бы вы о нем сказали? Выберите любые из вариантов, перечисленных ниже, или запишите свои результаты анализа:

```
var dogName = "rover";
var dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

☐ A. Код выглядит избыточным.

☐ B. Если потребуется изменить выходные данные или добавить новую весовую категорию, в программу придется вносить много изменений.

☐ C. Этот код однообразен, его скучно набирать!

☐ D. Код выглядит, мягко говоря, непонятно.

☐ E. ______________________________
____

Так чем плох этот код?

По сути, в этой программе один фрагмент выполняется снова и снова. И что в этом плохого, спросите вы? На первый взгляд — ничего. В конце концов, программа работает, не так ли? Давайте повнимательнее присмотримся к тому, что здесь происходит:

```
var dogName = "rover";
var dogWeight = 23;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
 ...
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
```

Вес собаки (weight) сравнивается с 20, и если он больше 20, выводится «громкое» сообщение WOOF WOOF. Если же вес меньше 20, то выводится «тихое» сообщение woof woof.

Так, а здесь... Вот это да! Здесь происходит ТОЧНО то же самое. А потом еще и еще...

```
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

Конечно, этот код выглядит невинно, но его скучно писать и трудно читать, а если со временем потребуется его изменить, это создаст массу проблем. Последнее обстоятельство становится все более существенным по мере накопления опыта программирования, весь код меняется, а сопровождение кода с дублированием логики превращается в кошмар: если логику потребуется изменить, то коррекция происходит в нескольких местах. И чем больше программа, тем больше будет изменений, а следовательно, и возможностей для ошибок. Значит, нужно как-то выделить повторяющийся код и разместить его в одном месте, где его можно будет легко использовать повторно при необходимости.

Как улучшить этот код? Потратьте несколько минут на размышления — удастся ли вам что-нибудь предложить? А может, в JavaScript есть что-то такое, что пригодится в таких ситуациях?

Ах, если бы только код можно было **повторно использовать** везде, где потребуется... Я бы просто **вызывала** его вместо того, чтобы **набирать заново**. И чтобы ему можно было присвоить понятное **имя**, которое легко запоминается. И чтобы изменения вносились только в **одном** месте, а не во многих... Как жаль, что это только мечты...

Кстати, а вы когда-нибудь слышали о ФУНКЦИЯХ?

Знакомьтесь: *функции.* Функции JavaScript позволяют взять фрагмент кода, присвоить ему имя, а затем ссылаться на его имя везде, где он понадобится. Судя по описанию — это то, что нам нужно.

Допустим, вы пишете программу про собак, в которой выводятся сообщения с лаем. Если программа имеет дело с большой собакой, то и лай будет громким и солидным: «WOOF WOOF». А если собака маленькая, хватит и тоненького «woof woof». И эта функциональность будет неоднократно использоваться в коде. Давайте напишем функцию `bark`, чтобы вызвать ее в нужной ситуации:

Определение функции начинается с ключевого слова function.

Далее следует имя функции, например bark.

А теперь указываются два значения, которые должны передаваться для использования функции: имя собаки и ее вес.

Это параметры функции. Параметры перечисляются в круглых скобках после имени функции.

```
function bark(name, weight) {

}
```

Мы будем называть его телом функции. Тело состоит из всего кода, заключенного между { и }.

Далее записывается код, который должен выполняться при использовании функции.

Теперь нужно написать код функции. Она должна проверять значение weight и выводить соответствующее сообщение.

Сначала проверяем значение weight...

Обратите внимание: имена переменных, используемые в коде, совпадают с именами параметров функции.

```
function bark(name, weight) {
    if (weight > 20) {
        console.log(name + " says WOOF WOOF");
    } else {
        console.log(name + " says woof woof");
    }
}
```

...а затем выводим имя собаки с одним из двух сообщений: WOOF WOOF или woof woof.

Итак, у нас имеется функция, которую мы можем использовать в своем коде. Посмотрим, как это делается...

Хорошо, но как все это работает?

Для начала перепишем наш код, чтобы в нем использовалась новая функция bark:

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}

bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

Замечательно, вся логика сосредоточена в одном месте.

Теперь программа сокращается до нескольких вызовов функции bark, каждому из которых передается имя и вес собаки.

Ого, все совсем просто!

Ого, кода стало гораздо меньше. Так что ваш коллега, которому потребуется разобраться в программе и внести небольшое изменение, легко справится с задачей. Кроме того, вся логика сосредоточена в одном месте.

Хорошо, но как разные фрагменты кода связываются друг с другом и как работает механизм вызова? Давайте подробно рассмотрим его, шаг за шагом.

Прежде всего сама функция.

Функция bark размещается в начале кода. Браузер читает этот фрагмент, видит, что перед ним функция, после чего просматривает команды в теле функции. Браузер знает, что прямо сейчас эти команды выполнять не нужно; он ждет, пока функция не будет вызвана из другой точки кода.

Также обратите внимание на *параметризацию* функции — это означает, что при вызове ей передаются параметры (имя собаки и вес). Это позволяет вызвать функцию для множества разных собак. При каждом вызове логика функции применяется к имени и весу, переданным при вызове.

Еще раз: это параметры; значения присваиваются им в момент вызова функции.

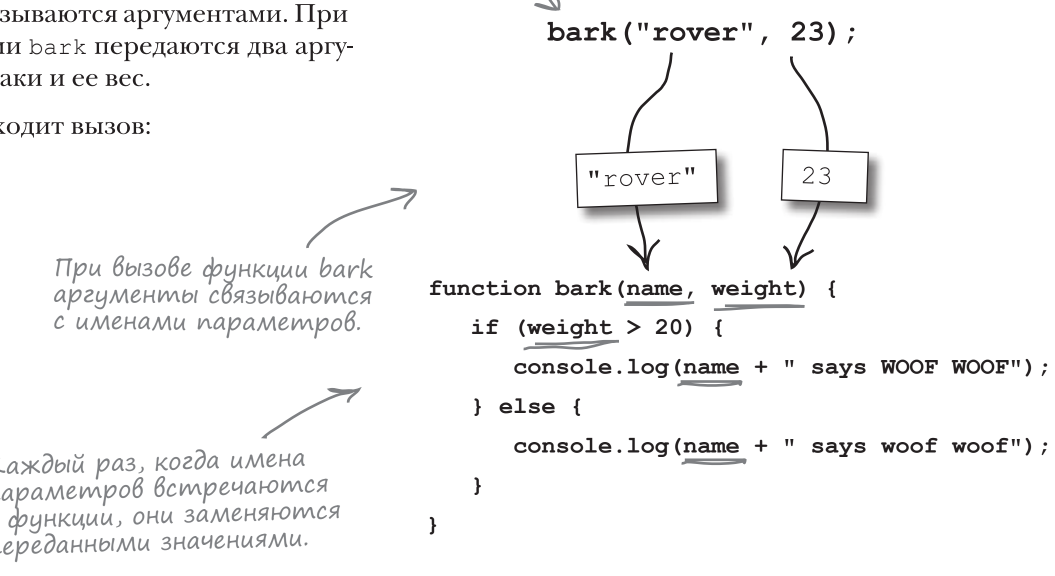
```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

А все, что находится внутри функции, составляет ее тело.

А теперь функцию нужно вызвать.

Чтобы вызвать функцию (передать ей управление), просто укажите ее имя, за которым в круглых скобках перечисляются все передаваемые значения. Значения, передаваемые при вызове, называются аргументами. При вызове функции `bark` передаются два аргумента: имя собаки и ее вес.

Вот как происходит вызов:



После вызова вся основная работа выполняется в теле функции.

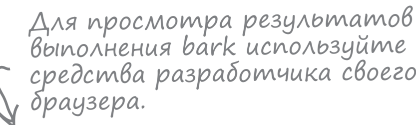
Когда значения всех параметров известны (например, параметр `name` содержит строку «rover», а параметр `weight` — число 23), можно переходить к выполнению тела функции.

Команды в теле функции выполняются сверху вниз, как и команды любого другого написанного вами кода. Единственное отличие заключается в том, что с именами `name` и `weight` связываются значения аргументов, передаваемые при вызове функции.

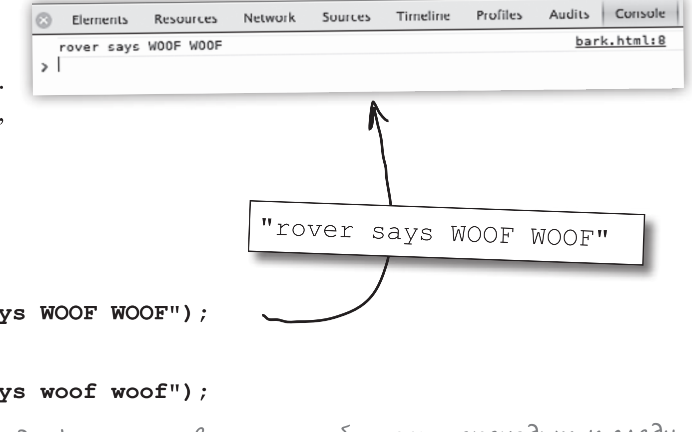
После того как значения аргументов будут связаны с именами параметров, начинается выполнение команд в теле функции.

```
function bark(name, weight) {
   if (23 > 20) {
      console.log("rover" + " says WOOF WOOF");
   } else {
      console.log("rover" + " says woof woof");
   }
}
```

Параметры работают в теле функции как переменные, которым присваиваются переданные значения аргументов.



А когда все будет сделано... Выполняется логика тела функции (в этом примере для имени Rover и веса 23 выводится сообщение “WOOF WOOF”), и работа функции на этом завершается. После этого управление возвращается команде, следующей за вызовом `bark`.




```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

Только что было сделано это...

...А теперь делается это.

Когда функция завершится, браузер переходит к следующей строке кода, идущей вслед за вызовом функции.

Здесь функция вызывается снова, с другими аргументами, и процесс начинается заново!

Возьми в руку карандаш

Ниже приведено несколько примеров вызовов bark. Рядом с каждым вызовом напишите, какой, по вашему мнению, он выдаст результат (или сообщение об ошибке). Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
bark("juno", 20); ____________________
bark("scottie", -1); ____________________
bark("dino", 0, 0); ____________________
bark("fido", "20"); ____________________
bark("lady", 10); ____________________
bark("bruno", 21); ____________________
```

Напишите, какое сообщение будет выведено на консоль.

Хмм... А что вы скажете об этих вызовах? Как вы думаете, что произойдет?

Развлечения с магнитами

Магниты с фрагментами программы JavaScript перепутались. Удастся ли вам расставить магниты по местам и восстановить работоспособную программу, чтобы она выдавала приведенный далее результат? Возможно, какие-то магниты окажутся лишними. Сверьтесь с ответами в конце главы.

```
,
,
,
}
}
}
(
function
)
{
whatShallIWear(80);
```

```
else {
    console.log("Wear t-shirt");
}
```

```
whatShallIWear
```

```
else if (temp < 70) {
    console.log("Wear a sweater");
}
```

```
temperature
```

```
if (temp < 60) {
    console.log("Wear a jacket");
}
```

```
temp
whatShallIWear(60);
whatShallIWear(50);
```

Консоль JavaScript

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

Так должна выглядеть консоль.

Откровенно о функциях

Интервью недели:
С функциями на дружеской ноге

Head First: Добрый вечер, Функция! Мы надеемся на откровенный разговор, очень хочется узнать о вас побольше.

Функция: Спасибо за приглашение.

Head First: Многие новички JavaScript о вас забывают. Они просто берутся за дело и пишут код — строку за строкой, сверху вниз, обходясь без функций. А от вас вообще есть польза?

Функция: Зря они так. Им же хуже, потому что от меня очень много пользы. Можно посмотреть на это так: я предоставляю возможность написать код один раз, а потом использовать его повторно снова и снова.

Head First: Простите за бестактность, но вы всего лишь позволяете делать одно и то же снова и снова... Скучновато звучит, вы не находите?

Функция: Вовсе нет, вызовы функций параметризуются: при каждом использовании функции вы передаете аргументы для выполнения конкретной операции, которая вам нужна.

Head First: Ммм... Например?

Функция: Допустим, вы хотите сообщить пользователю суммарную стоимость товаров, лежащих в корзине; вы пишете для этого функцию `computeShoppingCartTotal`. Этой функции можно передавать корзины разных пользователей, и каждый раз она будет вычислять стоимость конкретной корзины.

Head First: Если вы так хороши, почему неопытные разработчики не хотят вас использовать?

Функция: Ничего подобного, они постоянно используют меня: `alert`, `prompt`, `Math.random`, `document.write`. Трудно написать осмысленную программу без функций. Дело не в том, что неопытные пользователи не используют функции, просто они не определяют *свои собственные* функции.

Head First: Хорошо, с `alert` и `prompt` все понятно, но возьмем `Math.random` — это и на функцию-то не похоже.

Функция: `Math.random` — это функция, но она присоединена к другой полезной штуке, которой неопытные разработчики тоже пренебрегают: к *объекту*.

Head First: Ах да, объекты. Наши читатели еще познакомятся с ними в одной из последующих глав.

Функция: Вот и ладно. Тогда и вернемся к этому разговору

Head First: А что там с аргументами/параметрами? Все это выглядит довольно запутанно.

Функция: Считайте, что каждый параметр — это своего рода переменная в теле функции. При вызове функции каждое передаваемое значение присваивается соответствующему параметру.

Head First: Тогда что такое аргументы?

Функция: Просто другое название значений, передаваемых функции при вызове.

Head First: Честно говоря, я не понимаю, из-за чего столько разговоров. Да, с вашей помощью можно повторно использовать код и передавать значения в параметрах. И это все? Вроде все просто и понятно.

Функция: О, это далеко не все. Я еще многое умею: возвращать значения, скрывать ваш код от посторонних глаз, проделывать одну классную штуку под названием «замыкание», а еще я нахожусь в *очень близких* отношениях с объектами.

Head First: Оооо... ДА НЕУЖЕЛИ?! И мы сможем провести эксклюзивное интервью на эту тему?

Функция: Посмотрим...

Что можно передать функции?

При вызове функции ей передаются аргументы, которые сопоставляются с параметрами из определения функции. В аргументе можно передать практически любое значение JavaScript: строку, число, булевское значение:

В аргументе можно передать любое значение JavaScript.

Каждый аргумент присваивается соответствующему параметру функции.

```
saveMyProfile("krissy", 1991, 3.81, false);

function saveMyProfile(name, birthday, GPA, newuser) {
    if (birthday >= 2004) {
        // Для родившихся в 2004 году и позже...
    }
   // ...Остальной код функции...
}
```

И каждый параметр используется внутри функции как переменная.

Переменные тоже могут передаваться как аргументы, причем этот способ применяется даже чаще непосредственной передачи значений. Вызов той же функции с использованием переменных:

```
var student = "krissy";
var year = 1991;
var GPA = 381/100;
var status = "existinguser";
var isNewUser = (status == "newuser");
saveMyProfile(student, year, GPA, isNewUser);
```

Каждое из переданных значений хранится в переменной. При вызове функции текущие значения переменных передаются как аргументы.

Итак, в данном случае значение переменной student, "krissy", становится аргументом, соответствующим параметру name.

Переменные также передаются в других аргументах.

В качестве аргументов могут использоваться даже выражения:

```
var student = "krissy";
var status = "existinguser";
var year = 1991;

saveMyProfile(student, year, 381/100, status == "newuser");
```

В каждом случае программа сначала вычисляет выражение, получает конкретное значение и передает его функции.

Да, в аргументах могут передаваться даже такие выражения!

Выражение может быть числовым...

... или булевским, как в данном примере (функции передается false).

Что-то я не очень понимаю, чем параметр отличается от аргумента. Это два разных названия для одного и того же?

Нет, это разные понятия.

Определяя функцию, вы определяете ее с одним или несколькими параметрами.

Здесь определяются три параметра: degrees, mode и duration.

```
function cook(degrees, mode, duration) {
   // Код функции
}
```

При вызове функции вы *передаете* ей *аргументы*:

```
cook(425.0, "bake", 45);
```

Это аргументы. Здесь при вызове передаются три аргумента: вещественное число, строка и целое число.

```
cook(350.0, "broil", 10);
```

Итак, параметры определяются только один раз, но функция будет вызываться с разными аргументами.

Какой результат выведет эта программа? Уверены?

```
function doIt(param) {
    param = 2;
}
var test = 1;
doIt(test);
console.log(test);
```

Упражнение

Ниже приведен фрагмент кода JavaScript с переменными, определениями функций и вызовами функций. Ваша задача — найти все переменные, функции, аргументы и параметры. Запишите имена в соответствующих прямоугольниках справа. Свертесь с ответами в конце главы, прежде чем читать дальше.

```
function dogYears(dogName, age) {
    var years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
var myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
var guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    var defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
var pet = prompt("Enter a type of pet: ");
speak(pet);
```

Переменные

Функции

Встроенные функции

Аргументы

Параметры

B JavaScript используется передача по значению

То есть передача посредством копирования

Очень важно правильно понимать, как в JavaScript организована передача аргументов. JavaScript передает аргументы функциям по значению. Это означает, что каждый аргумент копируется в переменную-параметр. Рассмотрим простой пример, который покажет, как работает этот механизм передачи.

1 Объявим переменную age и инициализируем ее значением 7.

```
var age = 7;
```

2 Теперь объявим функцию addOne с параметром x, которая увеличивает значение x на 1.

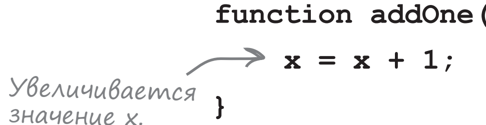
```
function addOne(x) {
    x = x + 1;
}
```

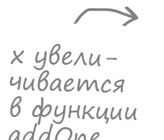
3 Вызовем функцию addOne и передадим ей в аргументе переменную age. Значение age копируется в параметр x.

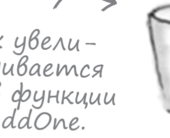
```
addOne(age);
```

4 Значение x увеличивается на 1. Однако следует помнить, что x содержит копию исходного значения, так что увеличивается только x, но не age.

```
function addOne(x) {
    x = x + 1;
}
```







Переменная age сохраняет прежнее значение, хотя x изменяется.

Что я думаю о передаче по значению? С одной стороны, этот механизм работает очень просто, с другой — такое впечатление, что чего-то недостает.

Хорошо, что вы об этом подумали. Очень важно понимать, как JavaScript передает значения функциям. С одной стороны, передача по значению действительно проста: когда аргумент передается функции, его значение сначала *копируется*, а затем присваивается соответствующему параметру. Тот, кто недостаточно хорошо это понимает, может сделать неверные предположения о том, как взаимодействуют функции, аргументы и параметры.

Главное последствие передачи по значению заключается в том, что любые изменения параметра в функции затрагивают *только сам параметр*, но не исходную переменную. Вот, собственно, и все.

Но у каждого правила найдутся исключения, и мы еще вернемся к этой теме через пару глав, когда будем изучать объекты. Не беспокойтесь, если вы хорошо понимаете суть передачи по значению, то у вас не будет особых проблем с изучением этого материала.

А пока просто запомните, что из-за передачи по значению все, что происходит с параметром внутри функции, *остается в границах функции и не выходит наружу.*

Мозговой штурм
второй заход

Помните это упражнение? Что вы скажете теперь, когда вам известно о передаче по значению? А может, вы дали правильный ответ с первого раза?

```
function doIt(param) {
    param = 2;
}
var test = 1;
doIt(test);
console.log(test);
```

Эксперименты с функциями

До настоящего момента мы говорили о «нормальном» использовании функций. Но что произойдет, если передать функции слишком много аргументов? Или слишком мало? Интуиция подсказывает, что ничего хорошего. Проверим?

functions

ЭКСПЕРИМЕНТ №1: что произойдет, если передать слишком мало аргументов?

Рискованная затея, но все ограничивается тем, что параметрам, не получившим аргументов, присваивается значение undefined. Пример:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3);
```

Консоль JavaScript

```
Brewing 3 cups of undefined
```

Обратите внимание: параметр tea содержит значение undefined, потому что мы не передали его значение.

ЭКСПЕРИМЕНТ №2: а если аргументов будет слишком много?

Тогда JavaScript просто игнорирует лишние аргументы. Пример:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3, "Earl Grey", "hey ma!", 42);
```

Консоль JavaScript

```
Brewing 3 cups of Earl Grey
```

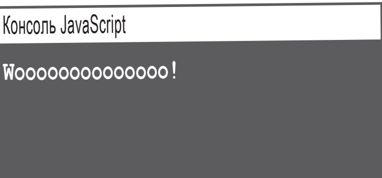
Прекрасно работает, функция игнорирует лишние аргументы.

На самом деле получить значения лишних аргументов возможно, но пока об этом рано беспокоиться...

ЭКСПЕРИМЕНТ №3: что произойдет, если у функции вообще НЕТ параметров?

Ничего страшного, таких функций очень много!

```
function barkAtTheMoon() {
  console.log("Woooooooooooooo!");
}
barkAtTheMoon();
```



А еще функции могут возвращать значения

Вы уже умеете взаимодействовать со своими функциями в одном направлении — умеете передавать аргументы *функциям*. Но как насчет обратной связи? Может ли функция передать данные пользователю? Знакомьтесь, команда `return`:

```
function bake(degrees) {
    var message;

    if (degrees > 500) {
        message = "I'm not a nuclear reactor!";
    } else if (degrees < 100) {
        message = "I'm not a refrigerator!";
    } else {
        message = "That's a very comfortable temperature for me.";
        setMode("bake");
        setTemp(degrees);
    }
    return message;
}

var status = bake(350);
```

Новая функция bake при вызове получает температуру печи в градусах.

Затем функция присваивает переменной строку, которая зависит от температуры, переданной в параметре degrees.

Вероятно, здесь выполняется какая-то реальная работа, но сейчас нас такие подробности не интересуют...

Сейчас нас интересует команда return, которая возвращает сообщение — результат выполнения функции.

Когда вызванная функция возвращает управление, возвращаемая строка будет присвоена переменной status.

Если в этом примере вывести значение status, переменная будет содержать строку «That's a very comfortable temperature for me». Разберитесь, как работает код, и вы все поймете!

350 градусов по Фаренгейту — идеальная температура для приготовления печенья. Испеките немного, если разыгрался аппетит, и переходите к следующей странице.

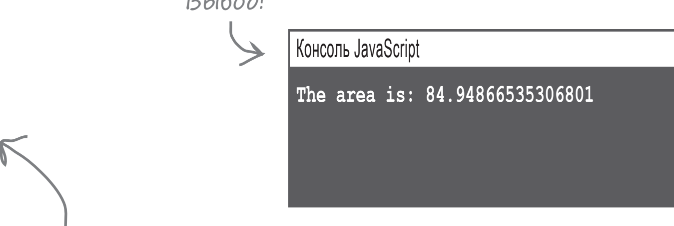
Пошаговое выполнение функции с командой return

Теперь, когда вы знаете, как работают аргументы и параметры и как функция возвращает значение, мы последовательно проанализируем вызов функции от начала до конца и посмотрим, что происходит на каждом шаге. Обязательно следите за выполнением действий в указанном порядке.

1. Объявляем переменную radius и инициализируем ее значением 5.2.
2. Затем программа вызывает функцию calculateArea и передает переменную radius как аргумент.
3. Аргумент присваивается параметру r. Функция calculateArea начинает выполняться с параметром r, содержащим значение 5.2.
4. Тело функции начинает выполняться с объявления переменной area. Затем мы проверяем, имеет ли параметр r значение <= 0.
5. Если r <= 0, то функция возвращает 0 и перестает выполняться. В нашем примере передается значение 5.2, так что эта строка НЕ выполняется.
6. Вместо этого выполняется секция else.
7. Программа вычисляет площадь круга, используя значение 5.2 параметра r.
8. Функция возвращает вычисленное значение. Выполнение функции на этом прекращается.
9. Значение, возвращаемое функцией, сохраняется в переменной theArea.
10. Выполнение продолжается со следующей строки.

```
(3) function calculateArea(r) {
        var area;
(4)     if (r <= 0) {
(5)         return 0;
(6)     } else {
(7)         area = Math.PI * r * r;
(8)         return area;
        }
    }
(1) var radius = 5.2;
                  (2)
(9) var theArea = calculateArea(radius);

(10) console.log("The area is: " + theArea);
```

Вывод!



Разработчики часто называют такой анализ «трассировкой потока управления». Как видите, при вызове функций и возврате значений управление может передаваться из одной точки программы в другую. Просто внимательно и без спешки разберите происходящее, шаг за шагом.

Анатомия функции

Мы рассмотрели смысл основных компонентов определения и вызова функций; новые знания нужно закрепить в памяти, чтобы они не забылись. Основные части определения функции:

```
function addScore ( level , score ) {
    var bonus = level * score * .1;
    return score + bonus;
}
```

Определение функции всегда начинается с ключевого слова function.

За ключевым словом function следует имя функции.

Затем в круглых скобках перечисляются параметры, разделенные запятыми (нуль и более).

Даже если функция не имеет параметров, пара круглых скобок () все равно необходима.

Тело функции находится между фигурными скобками и содержит набор команд (точно таких же, которые используются в обычном коде).

Переменные, используемые внутри функции, объявляются в теле функции.

Закрывающая фигурная скобка.

Функция может содержать команду return, но это не обязательно.

Команда return содержит значение или выражение, которое возвращается как результат вызова функции.

Часто задаваемые вопросы

В: Что произойдет, если я перепутаю порядок аргументов и параметрам будут присвоены аргументы, предназначенные для других параметров?

О: Все, что угодно! Почти наверняка дело кончится ошибкой во время выполнения или некорректным кодом. Всегда внимательно следите за параметрами функций и знайте, какие аргументы должна получать функция.

В: Почему перед именами параметров не ставится var? Ведь параметр — это просто новая переменная, верно?

О: По сути верно. Но функция сама выполняет работу по созданию переменной, так что вам не нужно ставить ключевое слово var перед именами параметров.

В: Каким правилам должны подчиняться имена функций?

О: Имена функций должны подчиняться тем же правилам, что и имена переменных. Как и в случае с переменными, выбирайте имена, которые несут полезную информацию и дают представление о том, что делает функция. В именах функций (как и в именах переменных) рекомендуется использовать «горбатый регистр» (например, camelCase).

В: Что произойдет, если переменная-аргумент использует то же имя, что и параметр? Допустим, в обоих случаях используется имя x?

О: Даже если имена аргумента и параметра совпадают, параметр x содержит копию аргумента x, поэтому они считаются разными переменными. Изменение значения параметра x не изменяет значения аргумента x.

В: Что вернет функция, не содержащая команды return?

О: Функция без команды return вернет `undefined`.

Я смотрю, мы начали размещать объявления переменных прямо внутри функций. Они работают точно так же, как обычные переменные?

Верно замечено. И да, и нет.

Эти объявления работают внутри функции точно так же, как снаружи, — они инициализируют новую переменную. Однако между переменной, объявленной *внутри функции*, и переменной, объявленной *за ее пределами*, существует принципиальное отличие в отношении места ее использования — места, где в коде JavaScript может содержаться ссылка на переменную. Если переменная объявляется вне функции, ее можно использовать в *любой* точке кода. Если же переменная объявляется внутри функции, то она может использоваться только внутри *этой же функции*. Область, в которой может использоваться переменная, называется ее областью действия (scope). Существуют две разные области действия: глобальная и локальная.

Мы должны поговорить о том, как вы пользуетесь переменными...

Глобальные и локальные переменные

Узнайте, чем они отличаются, или попадете впросак

Вы уже знаете, что переменную можно объявить в любой точке сценария с ключевым словом `var` и именем:

```
var avatar;
var levelThreshold = 1000;
```

Это глобальные переменные, доступные в любой точке вашего кода JavaScript.

И вы уже знаете, что переменные могут объявляться внутри функций:

```
function getScore(points) {
    var score;
    var i = 0;
    while (i < levelThreshold) {
       //Код
       i = i + 1;
    }
    return score;
}
```

Переменные points, score и i объявляются внутри функции.

Такие переменные называются локальными, потому что они известны только локально внутри самой функции.

Даже если levelThreshold используется внутри функции, эта переменная остается глобальной, ведь она объявлена за пределами функции.

Переменная, объявленная за пределами функции, называется ГЛОБАЛЬНОЙ. Если переменная объявлена внутри функции, она называется ЛОКАЛЬНОЙ.

Почему это важно, спросите вы? Ведь переменная все равно остается переменной? Дело в том, что от *места* объявления переменной зависит ее *видимость* в других частях кода. А позднее вы поймете, что понимание особенностей этих двух видов переменных упрощает сопровождение кода (не говоря уже о понимании кода, написанного другими разработчиками).

У меня вопрос... Мы говорили о выборе осмысленных имен переменных, но вы только что использовали переменную с именем i. Не очень-то осмысленно.

Верно подмечено.

Существует давняя традиция использовать переменную с именем i для управления циклом. Эта традиция идет от тех времен, когда объем программы был ограничен (а код пробивался на перфокартах) и у коротких имен были реальные преимущества. Теперь это всего лишь традиция, понятная любому программисту. Для этой же цели часто используются переменные j, k, а иногда даже x и y. Впрочем, это всего лишь исключение из общего полезного правила о выборе осмысленных имен переменных.

Область действия локальных и глобальных переменных

Место определения переменных задает их *область действия*, то есть место, где они будут (или не будут) «видны» для вашего кода. Рассмотрим примеры переменных с локальной и глобальной областью действия. Не забудьте, что переменные, определяемые внутри функции, имеют глобальную область действия, а переменные в функциях имеют локальную область действия.

```
var avatar = "generic";
var skill = 1.0;
var pointsPerLevel = 1000;
var userPoints = 2008;

function getAvatar(points) {
    var level = points / pointsPerLevel;

    if (level == 0) {
       return "Teddy bear";
    } else if (level == 1) {
       return "Cat";
    } else if (level >= 2) {
       return "Gorilla";
    }
}

function updatePoints(bonus, newPoints) {
    var i = 0;
    while (i < bonus) {
        newPoints = newPoints + skill * bonus;
        i = i + 1;
    }
    return newPoints + userPoints;
}

userPoints = updatePoints(2, 100);
avatar = getAvatar(2112);
```

Эти четыре переменные имеют глобальную область действия. Они будут видны во всем коде, приведенном внизу.

Если к вашей странице подключаются дополнительные сценарии, то эти глобальные переменные будут видны им, а их глобальные переменные будут видны вам!

Переменная level является локальной; она видна только в коде, содержащемся в функции getAvatar. Так что к переменной level можно обращаться только внутри этой функции.

И не забывайте о параметре points, он тоже имеет локальную область видимости в функции getAvatar.

Обратите внимание: getAvatar также использует глобальную переменную pointsPerLevel.

В updatePoints определяется локальная переменная i. Переменная i видна во всем коде updatePoints.

bonus и newPoints также локальны по отношению к updatePoints, тогда как переменная userPoints является глобальной.

А здесь в коде могут использоваться только глобальные переменные. Все переменные, объявленные внутри функций, невидимы в глобальной области действия.

Короткая жизнь переменных

Переменным приходится много работать, а их жизнь коротка. Ну кроме глобальных переменных, конечно... но даже у глобальных переменных срок жизни ограничен. Но от чего он зависит? Запомните пару простых правил:

Глобальные переменные существуют, пока существует страница. Жизнь глобальной переменной начинается с загрузки ее кода JavaScript в странице. Когда страница перестает существовать, то же происходит и с глобальной переменной. Даже если перезагрузить ту же страницу, все глобальные переменные будут уничтожены и созданы заново в перезагруженной странице.

Локальные переменные обычно исчезают при завершении функции. Локальные переменные создаются при первом вызове функции и продолжают существовать до выхода из нее (с возвращением значения или без). Таким образом, вы можете взять значения локальных переменных и вернуть их из функции, пока эти переменные еще не отправились к своему цифровому творцу.

Мы говорим «обычно», потому что жизнь локальных переменных можно продлить всякими хитроумными способами, но сейчас мы пока не будем их рассматривать.

В общем, жизнь переменной неразрывно связана со страницей. Локальные переменные приходят и быстро уходят, а более везучие глобальные переменные продолжают существовать до тех пор, пока браузер не перезагрузит страницу.

Не забывайте объявлять локальные переменные!

Если вы используете переменную, которая не была ранее объявлена, такая переменная будет глобальной. Это означает, что даже если переменная впервые используется внутри функции (потому что вы собирались сделать ее локальной), эта переменная на самом деле будет глобальной и окажется доступной и за пределами функции (что позднее может создать путаницу). В общем, не забывайте объявлять локальные переменные!

Если вы забудете объявить переменную перед использованием, то такая переменная всегда будет глобальной (даже если она впервые используется внутри функции).

```
function playTurn(player, location) {
    points = 0;
    if (location == 1) {
       points = points + 100;
    }
    return points;
}
var total = playTurn("Jai", 1);
alert(points);
```

Мы забыли объявить points с ключевым словом var перед использованием. Переменная points автоматически становится глобальной.

Получается, что переменную points можно будет использовать за пределами функции! Ее значение не исчезает (как должно быть) после завершения функции.

Программа ведет себя так, словно она была написана в следующем виде:

Так как мы забыли использовать var, JavaScript считает, что переменная points должна быть глобальной, и ведет себя так, словно объявление points располагается на глобальном уровне.

```
var points = 0;
function playTurn(player, location) {
    points = 0;
    if (location == 1) {
       points = points + 100;
    }
    return points;
}
var total = playTurn("Jai", 1);
alert(points);
```

Пропущенные объявления локальных переменных могут создать проблемы, если их имена совпадают с именами других глобальных переменных. В итоге вы можете случайно изменить значение, которое изменять не собирались.

Происходит «замещение» глобальной переменной. Допустим, в программе имеется глобальная переменная **beanCounter**, и вы объявляете функцию:

```
var beanCounter = 10;

function getNumberOfItems(ordertype) {
    var beanCounter = 0;
    if (ordertype == "order") {
       // Что-то делаем с beanCounter...
    }
    return beanCounter;
}
```

Глобальная и локальная переменные.

Все обращения к **beanCounter** внутри функции относятся к локальной переменной, а не к глобальной. Мы говорим, что глобальная переменная «замещается» локальной (то есть она не видна, потому что локальная переменная ее загораживает).

Локальная и глобальная переменные никак не влияют друг на друга; если изменить одну, это изменение никак не отразится на другой. Эти переменные существуют независимо.

Упражнение

Ниже приведен фрагмент кода JavaScript с переменными, определениями функций и вызовами функций. Ваша задача — найти переменные, используемые во всех аргументах, параметрах, локальных и глобальных переменных. Запишите имена переменных в соответствующих прямоугольниках справа, затем обведите кружком «замещенные» переменные. Сверьтесь с ответами в конце главы.

```
var x = 32;
var y = 44;
var radius = 5;

var centerX = 0;
var centerY = 0;
var width = 600;
var height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}

function computeDistance(x1, y1, x2, y2) {
    var dx = x1 - x2;
    var dy = y1 - y2;
    var d2 = (dx * dx) + (dy * dy);
    var d = Math.sqrt(d2);
    return d;
}

function circleArea(r) {
    var area = Math.PI * r * r;
    return area;
}

setup(width, height);
var area = circleArea(radius);
var distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);
```

Аргументы

Параметры

Локальные переменные

Глобальные переменные

Беседа у камина

Сегодня в студии: глобальная и локальная переменные спорят о том, кто из них важнее в программе.

Глобальная переменная:

Привет, Локальная переменная. Кстати, а зачем ты вообще здесь? Я и так могу сделать для программиста все необходимое. В конце концов, я видна везде!

Локальная переменная:

Да, но повсеместное использование глобальных переменных считается плохим стилем программирования. Во многих функциях нужны локальные переменные. Понимаешь, их личные переменные, которые посторонним видеть не положено. Глобальные переменные видны повсюду.

Глобальная переменная:

Признай, что все локальные переменные можно просто заменить глобальными, и функции будут работать точно так же.

Локальная переменная:

Да и нет. Если действовать очень осторожно — да, конечно. Но это очень непростая задача, и при малейшей ошибке функции начнут работать с переменными, которые используются другими функциями для других целей. К тому же программа загромождается глобальными переменными, используемыми только при вызове одной функции. Все это только создает путаницу.

Глобальная переменная:

Можно обойтись без всякой путаницы. Программист просто создает все необходимые переменные в начале программы, и все они располагаются в одном месте...

Локальная переменная:

Да, и что же произойдет, если кому-то потребуется вызвать функцию с переменной — допустим x, а он не может вспомнить, не использовалась ли переменная x прежде? Придется проводить поиск по всему коду и выяснять, не встречалась ли прежде переменная x! Какой кошмар...

Глобальная переменная:

Если разумно подходить к выбору имен, будет проще следить за использованием переменных.

Локальная переменная:

А как насчет параметров? Параметры функций всегда локальны. С этим ничего не поделаешь.

Глобальная переменная:

Верно. Но зачем возиться с аргументами и параметрами, если все необходимые значения хранятся в глобальных переменных?

Глобальная переменная:

Локальная переменная: Прости, ты понимаешь, что сейчас говоришь? Функции нужны как раз для того, чтобы код можно было повторно использовать для выполнения разных вычислений с разными данными.

Глобальная переменная: Но локальные переменные так… недолговечны. Они появляются и тут же исчезают.

Локальная переменная: Признай: хороший стиль программирования рекомендует использовать локальные переменные, если только нет крайней необходимости в глобальных переменных. К тому же глобальные переменные нередко создают проблемы. Я видела программы JavaScript, в которых глобальные переменные вообще не использовались!

Глобальная переменная: Вообще не используются? Да глобальные переменные — опора для любого программиста JavaScript!

Локальная переменная: У неопытных программистов, конечно. Но со временем программист начинает правильно формировать структуру своего кода с учетом правильности, удобства сопровождения, да и просто хорошего стиля программирования — и он постепенно понимает, что глобальные переменные не стоит использовать без крайней необходимости.

Глобальная переменная: Кажется, мне нужно выпить.

Локальная переменная: Пьяная глобальная переменная? А вот это *точно* добром не кончится.

Часто Задаваемые Вопросы

В: Следить за локальными и глобальными переменными хлопотно, почему не ограничиться одними глобальными? Я всегда так делаю.

О: Если вы пишете сколько-нибудь сложный код (или код, который будет сопровождаться в течение некоторого времени), вам неизбежно придется заняться управлением переменными. При злоупотреблении глобальными переменными становится трудно следить за тем, где используются переменные (и где изменяются их значения), а это может привести к ошибкам в коде. Это обстоятельство начинает играть еще более важную роль при совместной работе над кодом или при использовании сторонних библиотек (хотя хорошо написанная библиотека имеет структуру, при которой такие проблемы не возникают).

Итак, используйте глобальные переменные там, где это оправданно, но будьте умеренны — и везде, где это возможно, используйте локальные переменные. По мере накопления опыта JavaScript вы освоите дополнительные способы структурирования кода, упрощающие его сопровождение.

В: В моей странице используются глобальные переменные, но я также загружаю другие файлы JavaScript. В этих файлах используются отдельные наборы глобальных переменных?

О: Глобальная область действия только одна, так что все загружаемые файлы работают с одним набором переменных (и создают глобальные переменные в этом же пространстве). Вот почему так важно избегать конфликтов имен (а также по возможности сократить использование глобальных переменных или даже отказаться от них).

В: Если имя параметра совпадает с именем глобальной переменной, значит ли это, что параметр «замещает» глобальную переменную?

О: Да — точно так же, как происходит при объявлении в функции новой локальной переменной с таким же именем, как у глобальной. Замещение глобального имени вполне допустимо, если вы уверены, что глобальная переменная не должна использоваться внутри функции. Впрочем, лучше включить в программу комментарий, чтобы не создавать проблем при последующем чтении вашего кода.

В: Если перезагрузить страницу в браузере, все глобальные переменные будут инициализированы заново?

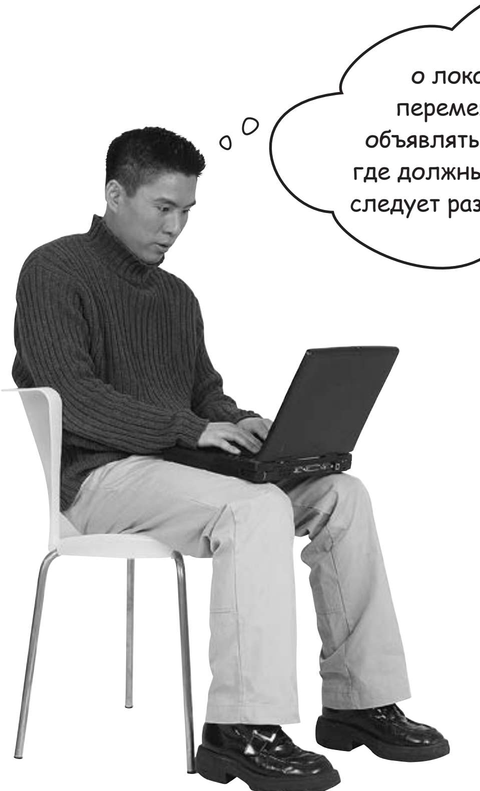
О: Да. Перезагрузка страницы фактически начинает ее существование заново (в том, что касается переменных). А если во время перезагрузки выполнялся какой-то код, то все локальные переменные тоже исчезнут.

В: Должны ли локальные переменные всегда объявляться в начале функции?

О: Как и в случае с глобальными переменными, локальные переменные можно объявлять непосредственно перед их первым использованием в функции. Тем не менее у программистов принято объявлять переменные в начале функции, чтобы каждый, кто будет читать ваш код, мог легко найти эти объявления и сразу получить представление обо всех локальных переменных, используемых в функции. Кроме того, если вы отложите объявление переменной, а потом случайно используете переменную в теле функции раньше, чем предполагалось, это может привести к неожиданным последствиям. JavaScript создает все локальные переменные в начале функции независимо от того, объявляются они или нет, но все переменные содержат undefined до момента инициализации — возможно, это не то, что вам нужно.

В: Часто приходится слышать жалобы на злоупотребление глобальными переменными в JavaScript. Почему это происходит? Язык был плохо спроектирован, или люди не понимают, что творят... или что? И как с этим бороться?

О: Действительно, многие программисты злоупотребляют глобальными переменными в JavaScript. Отчасти это объясняется тем, что с этим языком легко взяться за работу и начать программировать, — и это хорошо, потому что JavaScript не заставляет программиста обязательно использовать жесткую структуру кода. С другой стороны, когда на JavaScript так пишется серьезный код, который должен сопровождаться в течение значительного времени (а это относится практически ко всем веб-страницам), возникают проблемы. Впрочем, JavaScript — мощный язык, в котором предусмотрена поддержка таких конструкций, как объекты, обеспечивающие модульное строение кода. На эту тему написано много книг, а ваше первое знакомство с объектами состоится в главе 5 (остались какие-то две главы).



На самом деле функции могут размещаться в любом месте файла JavaScript.

JavaScript не обращает внимания на то, где объявляется функция — до или после ее использования. Для примера возьмем следующий код:

```
var radius = 5;
var area = circleArea(radius);
alert(area);

function circleArea(r) {
    var a = Math.PI * r * r;
    return a;
}
```

Обратите внимание: функция circleArea используется до ее определения в программе!

Функция circleArea определяется уже после того, как мы вызываем ее в приведенном коде. Как такое возможно?

На первый взгляд выглядит странно, особенно если вспомнить, что при загрузке страницы браузер начинает выполнять код JavaScript сверху вниз, от начала к концу файла. Но в действительности JavaScript дважды обрабатывает страницу: при первом проходе читаются все определения функций, а при втором начинается выполнение кода. Этот факт и позволяет размещать функции в любом месте файла.

Умная Машина

Умная Машина — потрясающее изобретение. Она пыхтит, стучит и бренчит, но что же она делает...? Лучше спросите кого-нибудь другого. Например, программиста — они утверждают, что знают, как работает этот агрегат. Удастся ли вам разобраться в коде и найти, что же с ним не так?

```
function clunk(times) {
    var num = times;
    while (num > 0) {
        display("clunk");
        num = num - 1;
    }
}

function thingamajig(size) {
    var facky = 1;
    clunkCounter = 0;
    if (size == 0) {
       display("clank");
    } else if (size == 1) {
       display("thunk");
    } else {
        while (size > 1) {
            facky = facky * size;
            size = size - 1;
        }
        clunk(facky);
    }
}

function display(output) {
    console.log(output);
    clunkCounter = clunkCounter + 1;
}
var clunkCounter = 0;
thingamajig(5);
console.log(clunkCounter);
```

Попробуйте передать Умной Машине числа 0, 1, 2, 3, 4, 5 и т.д. Сможете ли вы понять, как она работает?

Руководство по гигиене кода

Обитатели мира JavaScript стремятся к тому, чтобы вокруг все было чисто, аккуратно и готово к расширению. Естественно, каждый программист должен начинать именно со своего кода, но JavaScript порой излишне благодушно относится к тому, что связано с организацией ваших переменных и функций. По этой причине мы подготовили для вас небольшое руководство с советами для новичков. Не стесняйтесь, берите листовки — они БЕСПЛАТНЫЕ.

Глобальные переменные — В САМОМ НАЧАЛЕ!

Глобальные переменные рекомендуется по возможности сгруппировать и разместить в начале кода, где их будет легко найти. Может, это и не обязательно, но разве не проще будет вам и другим программистам найти объявление любой переменной, используемой в коде, если оно будет находиться в самом начале?

Функции любят общество себе подобных.

Вообще-то нет; на самом деле им это безразлично, ведь это всего лишь функции. Но если вы будете держать свои функции рядом друг с другом, вам будет намного проще найти их. Как вы знаете, браузер просматривает код JavaScript в поисках функций, прежде чем делать что-то еще. Следовательно, функции можно разместить как в начале, так и в конце файла, но если они будут находиться в одном месте — вам же будет проще. Здесь в мире JavaScript мы часто начинаем с объявления глобальных переменных, а затем размещаем свои функции.

Объявляйте локальные переменные В НАЧАЛЕ той функции, которой они принадлежат.

Разместите все объявления локальных переменных в начале тела функции. Это упростит их поиск и гарантирует, что все они будут объявлены перед использованием.

Вот и все, соблюдайте правила – и радуйтесь жизни в новом мире JavaScript!

Несколько понятий JavaScript, облачившись в маскарадные костюмы, развлекаются игрой «Кто я?». Они дают подсказки, а вы должны угадать их по тому, что они говорят о себе. Предполагается, что участники всегда говорят о себе правду. Запишите рядом с каждым высказыванием имя соответствующего участника.

Сегодняшние участники:

function, аргумент, return, область действия, локальная переменная, глобальная переменная, передача по значению, параметр, вызов функции, Math.random, встроенные функции, повторное использование кода.

Меня передают функции.	________________
Я возвращаю результат вызова.	________________
Я — очень важное ключевое слово.	________________
Мне передаются значения аргументов.	________________
По сути я означаю «создание копии».	________________
Я повсюду.	________________
Синоним «передачи управления функции».	________________
Пример функции, связанной с объектом.	________________
Примеры — alert и prompt.	________________
Для этого хорошо подходят функции.	________________
Где меня можно увидеть.	________________
Я существую в границах своей функции.	________________

Дело незадачливого грабителя

Код под увеличительным стеклом

Шерлок закончил свой телефонный разговор с начальником полиции Лестрейдом, уселся у камина и снова взялся за газету. Ватсон вопросительно посмотрел на него.

«Что такое?» — спросил Шерлок, не отрываясь от газеты.

«Ну как? Что сказал Лестрейд?» — поинтересовался Ватсон.

«Он сказал, что они нашли посторонний код на банковском счете, с которым происходило что-то подозрительное».

«И?» — продолжал Ватсон, тщетно пытаясь скрыть нетерпение.

«Лестрейд переслал мне код, а я сказал, что этим делом не стоит заниматься. Преступник совершил фатальную ошибку, и ему все равно не удастся ничего украсть», — ответил Шерлок.

«Откуда вы знаете?» — спросил Ватсон.

«Это очевидно, если знаешь, куда смотреть», — воскликнул Шерлок. — А теперь перестаньте отвлекать меня вопросами и дайте спокойно дочитать».

Пока Шерлок был занят последними новостями, Ватсон украдкой заглянул в телефон Шерлока, увидел сообщение Лестрейда и нашел код.

```
var balance = 10500;
var cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}

var amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

Баланс, с которым происходит что-то подозрительное.

Почему Шерлок решил, что дело не стоит расследовать? Как он узнал, что преступник не сможет украсть деньги, просто взглянув на код? Может, в коде кроется какая-то проблема? И даже не одна?

КЛЮЧЕВЫЕ МОМЕНТЫ

- Функция объявляется ключевым словом **function**, за которым следует имя функции.
- Все **параметры** функции заключаются в круглые скобки. Если функция не имеет параметров, используется пустая пара скобок.
- **Тело** функции заключается в фигурные скобки.
- При вызове функции выполняются команды, содержащиеся в теле функции.
- **Вызов** и **передача управления** функции — одно и то же.
- При вызове функции указывается ее имя и аргументы, которые становятся значениями параметров (если они есть).
- Функция может возвращать значение командой **return**.
- Функция создает локальную область действия параметров и локальных переменных, используемых функцией.
- Переменные имеют либо **глобальную область действия** (то есть видимы в любой точке программы), либо **локальную область действия** (видимы только в границах функции, в которой они объявлены).
- Локальные переменные рекомендуется объявлять в начале тела функции.
- Если вы забудете объявить локальную переменную с ключевым словом **var**, эта переменная будет глобальной, что может привести к непредвиденным последствиям в программе.
- Функции — хорошее средство упорядочения кода и создания фрагментов, пригодных для повторного использования.
- Передача аргументов позволяет адаптировать код функции для конкретных потребностей (и использовать разные аргументы для получения разных результатов).
- Функции также помогают сократить дублирование кода или полностью устранить его.
- Вы также можете использовать многочисленные встроенные функции JavaScript's — такие, как alert, prompt и Math.random, — для выполнения полезной работы в своих программах.
- Вызывая встроенные функции, вы используете существующий код, который вам не нужно писать самостоятельно.
- Программный код желательно организовать таким образом, чтобы объявления функций и глобальных переменных располагались вместе в начале файла JavaScript.

Возьми в руку карандаш
Решение

Проанализируйте приведенный ниже код. Что бы вы о нем сказали? Выберите любые из вариантов, перечисленных ниже, или запишите свои результаты анализа. Посмотрите на наше решение.

```
var dogName = "rover";
var dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

Мы выбрали все варианты!

A. Код выглядит избыточным.

B. Если нам потребуется изменить выходные данные или добавить новую весовую категорию, в программу придется вносить много изменений.

C. Этот код однообразен, его будет скучно набирать!

D. Код выглядит, мягко говоря, не очень понятно.

E. *Похоже, разработчик думал, что значения weight могут изменяться со временем.*

Возьми в руку карандаш
Решение

Ниже приведено несколько примеров вызовов bark. Рядом с каждым вызовом напишите, какой, по вашему мнению, он выдаст результат (или сообщение об ошибке). Ниже приведено наше решение.

`bark("juno", 20);` *juno says woof woof*

`bark("scottie", -1);` *scottie says woof woof*

Функция bark не проверяет, что вес собаки положителен. Следовательно, этот вызов будет работать, потому что -1 меньше 20.

`bark("dino", 0, 0);` *dino says woof woof*

Функция bark просто игнорирует лишний аргумент 0. Нулевое значение weight не имеет смысла, но программа работает.

`bark("fido", "20");` *fido says woof woof*

Строка «20» сравнивается с числом 20. «20» не больше 20, поэтому выводится значение «woof woof». (О том, как JavaScript сравнивает «20» и 20, будет рассказано позднее.)

`bark("lady", 10);` *lady says woof woof*

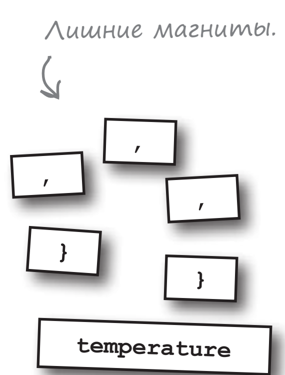
`bark("bruno", 21);` *bruno says WOOF WOOF*

Развлечения с магнитами. Решение

Магниты с фрагментами программы JavaScript перепутались на холодильнике. Удастся ли вам расставить магниты по местам и восстановить работоспособную программу, чтобы она выдавала приведенный далее результат? Возможно, какие-то магниты окажутся лишними. Ниже приведено наше решение.

```
function whatShallIWear ( temp ) {
    if (temp < 60) {
        console.log("Wear a jacket");
    }
    else if (temp < 70) {
        console.log("Wear a sweater");
    }
    else {
        console.log("Wear t-shirt");
    }
}
whatShallIWear(50);
whatShallIWear(80);
whatShallIWear(60);
```

Лишние магниты.



Консоль JavaScript

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

Ниже приведен фрагмент кода JavaScript с переменными, определениями функций и вызовами функций. Ваша задача — найти все переменные, функции, аргументы и параметры. Запишите каждое имя в соответствующем прямоугольнике справа. Ниже приведено наше решение.

```
function dogYears(dogName, age) {
    var years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
var myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
var guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    var defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
var pet = prompt("Enter a type of pet: ");
speak(pet);
```

Переменные

myDog, guests, pet, years, defaultSound

Функции

dogYears, makeTea, secret, speak,

Встроенные функции

alert, console.log, prompt

Аргументы

myDog, 4, guests, “Earl Grey”, pet, plus all the string arguments to alert and console.log

Параметры

dogName, age, cups, tea, kind

Упражнение
Решение

Ниже приведен фрагмент кода JavaScript с переменными, определениями функций и вызовами функций. Ваша задача — найти переменные, используемые во всех аргументах, параметрах, локальных и глобальных переменных. Запишите имена переменных в соответствующих прямоугольниках справа, затем обведите кружком «замещенные» переменные. Ниже приведено наше решение.

```
var x = 32;
var y = 44;
var radius = 5;

var centerX = 0;
var centerY = 0;
var width = 600;
var height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}

function computeDistance(x1, y1, x2, y2) {
    var dx = x1 - x2;
    var dy = y1 - y2;
    var d2 = (dx * dx) + (dy * dy);
    var d = Math.sqrt(d2);
    return d;
}

function circleArea(r) {
    var area = Math.PI * r * r;
    return area;
}

setup(width, height);
var area = circleArea(radius);
var distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);
```

Аргументы

width, height, radius, x, y, centerX, centerY, "Area: " + area, "Distance: "+ distance

Не забудьте аргументы функции alert.

Параметры

width, height, x1, y1, x2, y2, r

Локальные переменные

dx, dy, d2, d, area

Локальная переменная area замещает глобальную переменную area.

Глобальная переменная

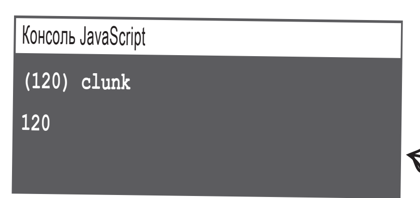
x, y, radius, centerX, centerY, width, height, area, distance

Не забывайте о area и distance — это тоже глобальные переменные.

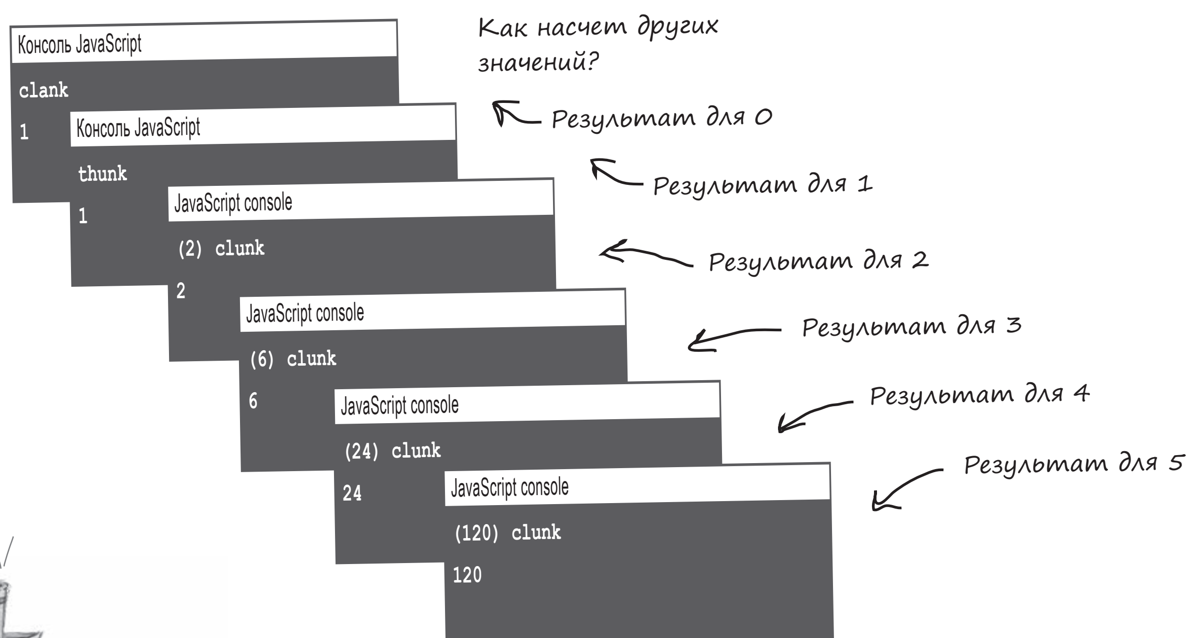
Умная Машина

Умная Машина — потрясающее изобретение. Она пыхтит, стучит и бренчит, но что же она делает?.. Лучше спросите кого-нибудь другого. Например, программиста — они утверждают, что знают, как работает этот агрегат. Удастся ли вам разобраться в коде и найти, что же с ним не так?

Наше решение:



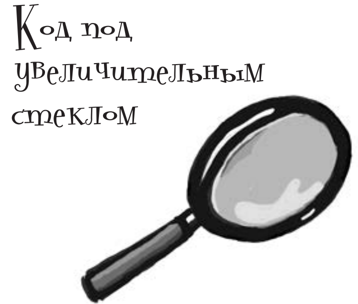
Если передать Умной Машине число 5, на консоль 120 раз выводится сообщение «clunk» (также ваша консоль может сократить 120 сообщений до одного, как показано слева), после чего выводится число 120.



И что же это значит? Говорят, Умную Машину изобрел один любопытный тип, который развлекался перестановкой букв в словах. Знаете — берем слово DOG и генерируем все возможные перестановки: GOD, OGD, DGO, GDO и ODG. Таким образом, Умная Машина подсчитывает, что из трех букв можно составить всего шесть возможных комбинаций. А если взять слово из пяти букв, то комбинаций будет уже 120 — ого! В общем, такие ходили слухи. А тут вдруг выясняется, что Умная Машина вычисляет факториал! Кто бы мог подумать?!

Что такое факториал? Посмотрите в Википедии!

Код под увеличительным стеклом



Почему Шерлок решил, что дело не стоит расследовать? Как он узнал, что преступник не сможет украсть деньги, просто взглянув на код? Может, в коде кроется какая-то проблема? И даже не одна? Ниже приведено наше решение.

```
var balance = 10500;
var cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}

var amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

balance — глобальная переменная...

... но здесь она замещается параметром.

И при попытке изменить balance в функции steal изменяется вовсе не банковский счет!

Функция возвращает «украденную» сумму...

... но исходный банковский счет остается в том же состоянии, в котором он находился изначально.

Преступник думает, что он украл деньги со счета. Ничего подобного!

Недотепа-преступник не только ничего не украл, но и забыл снова включить камеру наблюдения — полиция сразу догадается, что происходит нечто подозрительное. Помните: при возврате управления функция перестает выполняться, поэтому все строки кода после return игнорируются!

Несколько понятий JavaScript, облачившись в маскарадные костюмы, развлекаются игрой «Кто я?». Они дают подсказки, а вы должны угадать их по тому, что они говорят о себе. Предполагается, что участники всегда говорят о себе правду. Запишите рядом с каждым высказыванием имя соответствующего участника. Ниже приведено наше решение.

Сегодняшние участники:

function, аргумент, return, область действия, локальная переменная, глобальная переменная, передача по значению, параметр, вызов функции, Math.random, встроенные функции, повторное использование кода.

Меня передают функции.	аргумент
Я возвращаю результат вызова.	return
Я — очень важное ключевое слово.	function
Мне передаются значения аргументов.	параметр
По сути я означаю «создание копии».	передача по значению
Я повсюду.	глобальная переменная
Синоним «передачи управления функции».	вызов функции
Пример функции, связанной с объектом.	Math.random
Примеры — alert и prompt.	встроенные функции
Для этого хорошо подходят функции.	повторное использование кода
Где меня можно увидеть.	область действия
Я существую в границах своей функции.	локальная переменная

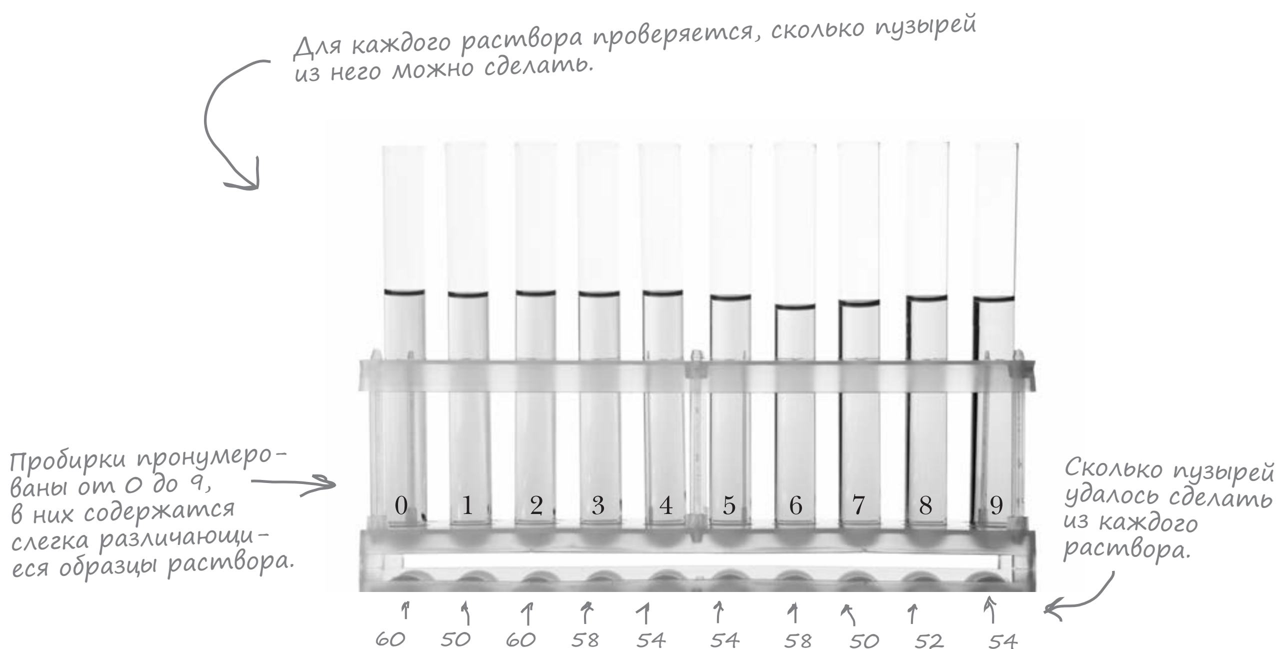
4 наведение порядка в данных

Массивы

JavaScript может работать не только с числами, строками и логическими значениями. До настоящего момента мы работали исключительно с **примитивами** – простыми строками, числами и логическими значениями (например, “Fido”, 23 и true). С примитивными типами можно сделать многое, но в какой-то момент возникнет необходимость в **расширенных данных** для представления всех позиций в корзине покупок, всех песен в плейлисте, группы звезд и их звездных величин или целого каталога продуктов. Подобные задачи требуют более серьезных средств. Типичным инструментом для представления таких однородных данных является **массив** JavaScript. В этой главе вы узнаете, как помещать данные в массив, передавать их и работать с ними. В последующих главах будут рассмотрены другие способы **структурирования данных**, но начнем мы с массивов.

Вы нам поможете?

Знакомьтесь: компания Bubbles-R-Us специализируется на мыльных пузырях. Ее непрекращающиеся исследования направлены на то, чтобы пользователи по всему миру выдували только самые красивые и крупные мыльные пузыри. Сегодня компания занимается тестированием нескольких версий нового изобретения — «пузырь-фактора», а проще говоря, проверяет, сколько пузырей дает каждая из новых разновидностей раствора. Вот как выглядят экспериментальная установка:



Конечно, мы хотим сохранить эти данные, чтобы написать код JavaScript для их анализа. Но работать придется не с одним значением, а с целым набором. Как написать код для работы с этими значениями?

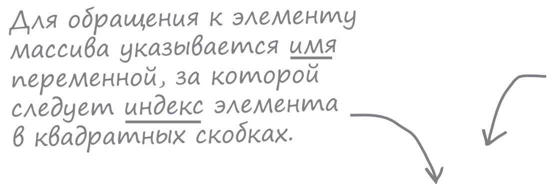
Как представить набор значений в JavaScript

Вы уже знаете, как в JavaScript представляются одиночные значения (строки, числа, логические значения), но как представить *набор* значений — как пузырь-факторы десяти разных образцов раствора? Для этого используются *массивы* JavaScript. Массив — это тип JavaScript, способный хранить несколько однотипных значений. Следующий массив JavaScript содержит данные нашего эксперимента:

```
var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54];
```

Десять значений объединяются в массив, который присваивается переменной scores.

Вы можете работать с набором значений как с единым целым или же обращаться к отдельным значениям при необходимости. Смотрите, как это делается:



Индексы массивов начинаются с нуля. Так что данные о количестве пузырей для первого образца хранятся в элементе с индексом 0, а для обращения к ним используется запись scores[0]; аналогично данные третьего образца хранятся в элементе с индексом 2, а для обращения к ним используется запись scores[2].

```
var solution2 = scores[2];
alert("Solution 2 produced " + solution2 + " bubbles.");
```

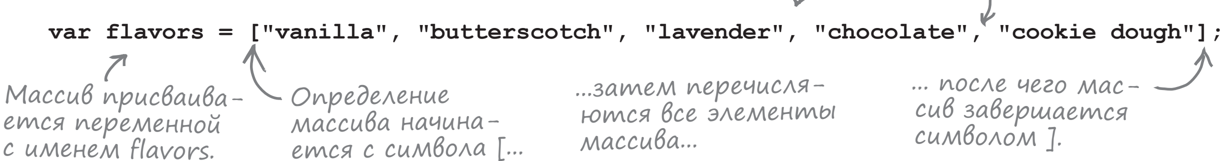
Как работают массивы

Прежде чем браться за программу для Bubbles-R-Us, новые знания нужно закрепить в памяти. Как было сказано выше, массивы предназначены для хранения нескольких значений (в отличие от переменных, которые хранят только одно значение, например строку или число). Чаще всего массивы используются для группировки сходных данных — температур, количества пузырей, сортов мороженого и даже ответов на вопросы анкеты «да/нет». Так что группируем набор значений, создаем массив и обращаемся к значениям по индексу.

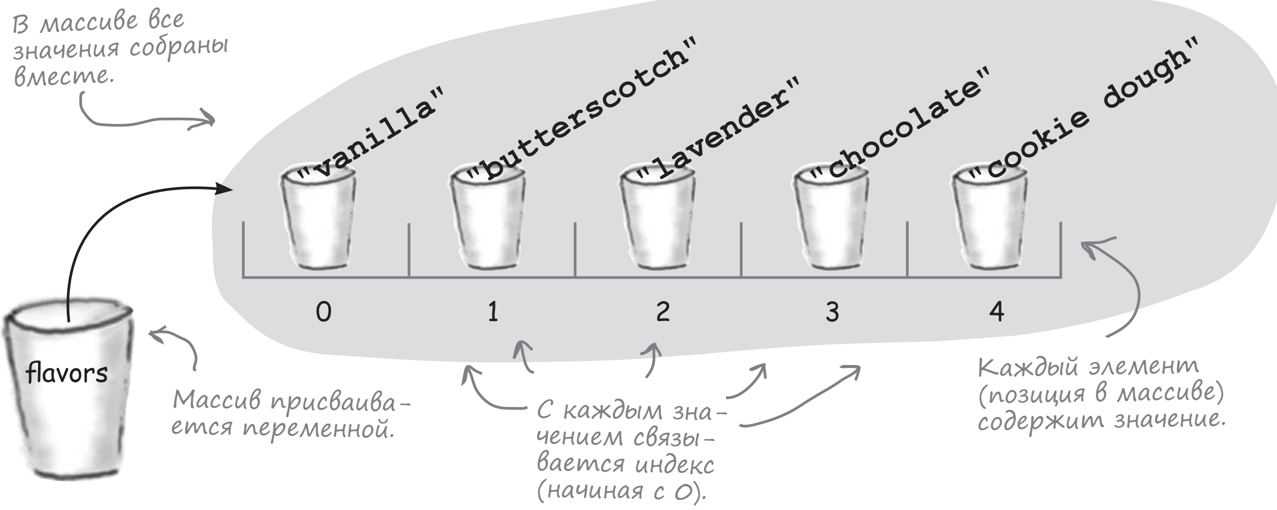
Как создать массив

Предположим, вы хотите создать массив для хранения сортов мороженого. Вот как это делается:

Обратите внимание: элементы массива разделяются запятыми.



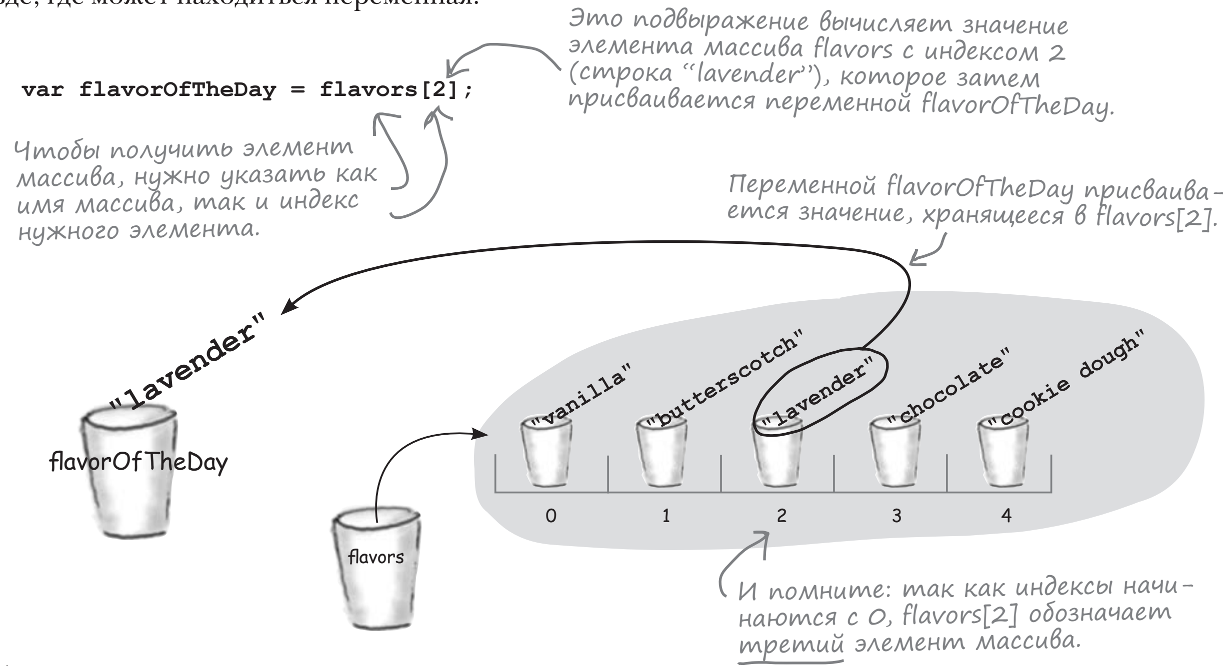
При создании массива каждый элемент находится в некоторой позиции, которая определяется его индексом. В массиве сортов мороженого flavors первый элемент — vanilla — имеет индекс 0, второй — butterscotch — индекс 1, и так далее. Вот как можно представлять этот массив:



Как обратиться к элементу массива

Каждый элемент массива обладает индексом — «ключом», используемым для обращения и изменения элемента массива. Чтобы обратиться к элементу, укажите имя переменной массива и индекс, заключенный в квадратные скобки. Эта запись может использоваться везде, где может находиться переменная:

```
var flavorOfTheDay = flavors[2];
```



Обновление значения в массиве

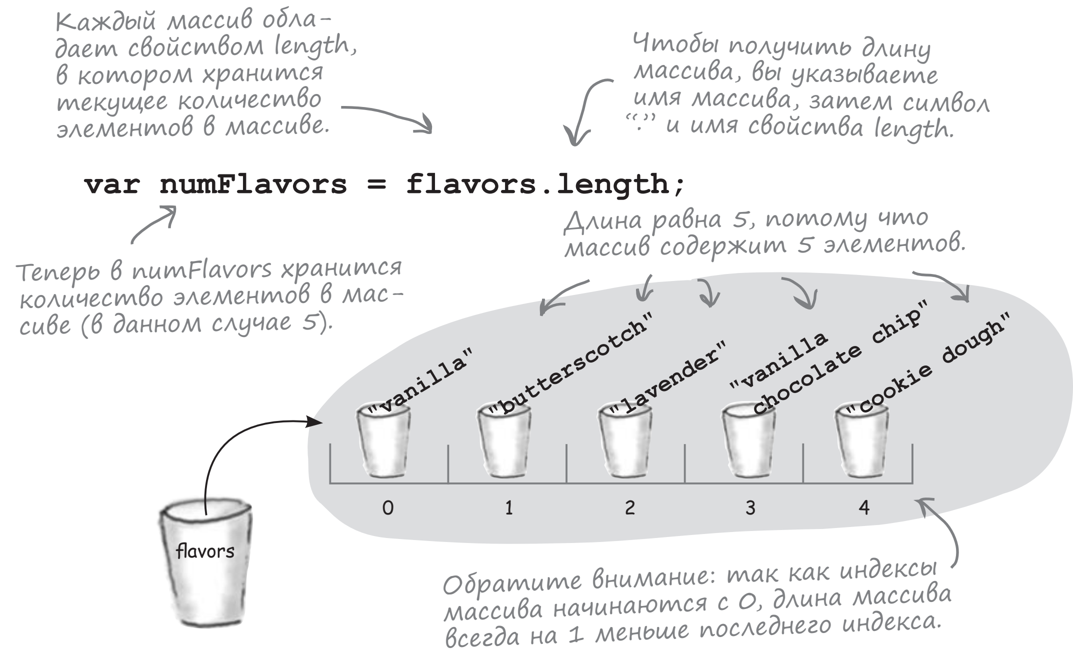
Индекс также может использоваться для изменения значений, хранящихся в массиве:

```
flavors[3] = "vanilla chocolate chip";
```

После выполнения этой строки массив будет выглядеть так:

Сколько же элементов в массиве?

Допустим, вы получаете большой массив с важными данными. Вы знаете, что в нем находится, но, скорее всего, точная длина массива вам неизвестна. К счастью, каждый массив обладает свойством `length`. О том, что такое свойства и как они работают, будет рассказано в следующей главе. А пока достаточно сказать, что свойство — это просто значение, связанное с массивом. Вот как используется свойство `length`:



В массиве products хранятся названия сортов мороженого. Названия добавлялись в массив в порядке создания. Закончите код так, чтобы он определял и возвращал в переменной recent сорт, созданный *последним*.

```
var products = ["Choo Choo Chocolate", "Icy Mint", "Cake Batter", "Bubblegum"];
var last = ________________;
var recent = products[last];
```

Опробуйте мой новый Генератор Красивых Фраз — и вы заговорите так же красиво, как ваш начальник или парни из отдела маркетинга.

Вам показалось, что наше Серьезное Бизнес-Приложение из главы 1 было недостаточно серьезным? Ладно. Попробуйте это, если вам нужно показать что-то начальнику.

Просмотрите код модного приложения Генератор Красивых Фраз. Удастся ли вам самостоятельно разобраться в том, что оно делает?

```
<!doctype html>
<html lang="en">
<head>
  <title>Phrase-o-matic</title>
  <meta charset="utf-8">
  <script>
    function makePhrases() {
      var words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
      var words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
      var words3 = ["process", "solution", "tipping-point", "strategy", "vision"];

      var rand1 = Math.floor(Math.random() * words1.length);
      var rand2 = Math.floor(Math.random() * words2.length);
      var rand3 = Math.floor(Math.random() * words3.length);

      var phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
      alert(phrase);
    }
    makePhrases();
  </script>
</head>
<body></body>
</html>
```

Генератор Красивых Фраз

Вероятно, вы уже поняли, что эта программа — идеальный инструмент для создания маркетинговых слоганов. В прошлом она уже сгенерировала немало шедевров и еще неоднократно проявит себя в будущем. Давайте разберемся, как же она работает:

1. Сначала мы определяем функцию `makePhrases`. Эта функция вызывается столько раз, сколько фраз потребуется сгенерировать:

```
function makePhrases() {

}
makePhrases();
```

Мы определяем функцию makePhrases, которая будет вызываться позднее.

Здесь размещается весь код makePhrases, скоро мы доберемся до него...

Здесь функция makePhrases вызывается один раз. Но если вдруг потребуется сгенерировать несколько фраз, ее можно будет вызвать многократно.

2. Разобравшись с общей структурой, можно переходить к написанию кода функции `makePhrases`. Начнем с создания трех массивов со словами, которые будут использоваться при построении фраз. На следующем шаге мы случайным образом выбираем одно слово из каждого массива и составляем фразу из трех слов.

Мы создаем переменную с именем words1 для первого массива.

```
var words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
```

В массив помещаются пять строк. Ничто не мешает вам заменить их другими, более модными словечками.

```
var words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
var words3 = ["process", "solution", "tipping-point", "strategy", "vision"];
```

Еще два массива со словами присваиваются переменным words2 и words3.

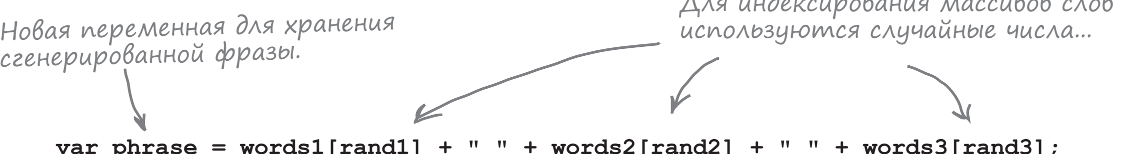
3. Программа генерирует три случайных числа — по одному для каждого из трех случайных слов, которые выбираются для фразы. Вспомните главу 2: функция Math.random генерирует число от 0 до 1 (не включая 1). Если умножить сгенерированное число на границу массива и отсечь дробную часть вызовом Math.floor, мы получим число в диапазоне от 0 до длины массива, уменьшенной на 1.

```
var rand1 = Math.floor(Math.random() * words1.length);
var rand2 = Math.floor(Math.random() * words2.length);
var rand3 = Math.floor(Math.random() * words3.length);
```

rand1 — число в диапазоне от 0 до последнего индекса массива words1.

Аналогично для rand2 и rand3.

4. Мы берем случайно выбранные слова и склеиваем их, разделяя пробелами для удобства чтения:

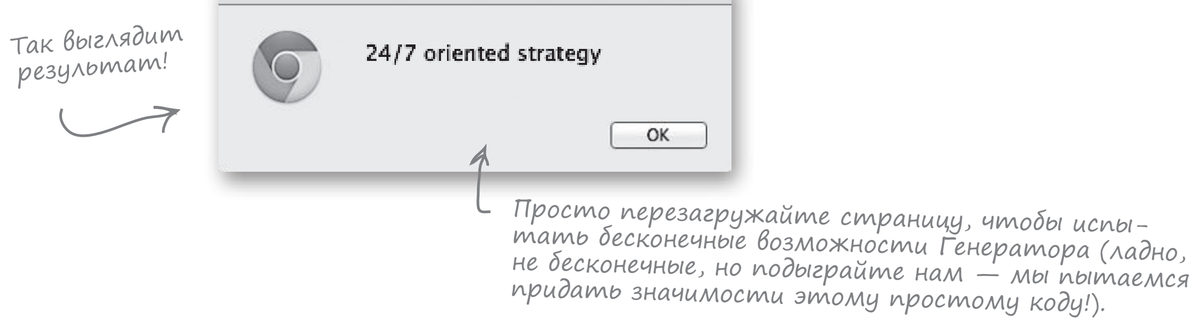



```
var phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
```

5. Почти готово; фраза построена, осталось вывести ее. Для этого мы, как обычно, воспользуемся функцией `alert`.

```
alert(phrase);
```

6. Введите последнюю строку кода, еще раз посмотрите на плоды своих трудов, прежде чем загружать их в браузере, — здесь есть чем гордиться. Запустите программу и подумайте, как получше применить сгенерированные фразы.



Часто задаваемые вопросы

В: Порядок элементов в массиве важен?

О: Чаще всего важен, но бывают и исключения. В массиве из примера с пузырями порядок очень важен, потому что индекс массива определяет номер образца — раствор 0 дает результат 60, и это число сохраняется в элементе с индексом 0. Если данные в массиве будут перепутаны, весь эксперимент пойдет насмарку! Однако в некоторых случаях порядок элементов роли не играет. Например, если в массиве хранятся случайно выбранные слова, вполне возможно, что вас интересует только само присутствие слова в массиве. Но если позднее вы решите, что слова должны быть упорядочены по алфавиту, порядок станет важен. Итак, ответ на этот вопрос зависит от использования массива. Пожалуй, при работе с массивами порядок все же важен.

В: Сколько значений можно поместить в массив?

О: Теоретически, сколько угодно. На практике это число ограничено объемом памяти вашего компьютера. Каждый элемент занимает небольшую часть памяти. Помните, что JavaScript работает в браузере, а браузер — всего лишь одна из программ, запущенных на вашем компьютере. Если элементы добавляются в массив снова и снова, то вскоре память будет исчерпана. В зависимости от типа данных максимальное количество элементов в массиве исчисляется многими тысячами, если не миллионами; на практике необходимость в таких объемах встречается нечасто. Не забывайте, что при большом количестве элементов программа начинает работать медленнее, поэтому размер массива лучше ограничить какой-нибудь разумной величиной — скажем, несколькими сотнями элементов.

В: Может ли массив быть пустым?

О: Да, может. Более того, вскоре вы увидите пример использования пустого массива. Такой массив создается командой

```
var emptyArray = [ ];
```

Позднее вы сможете добавить в пустой массив элементы.

В: Мы видели массивы, в которых хранятся строки и числа; можно ли размещать в массивах другие данные?

О: Да, можно. Более того, в массивах могут храниться практически любые значения JavaScript, включая строки, числа, логические значения, другие массивы и даже объекты (об этом позднее).

В: Должны ли все значения в массиве относиться к одному типу?

О: Нет, не обязательно, хотя *обычно* мы храним в массивах однотипные значения. В отличие от многих других языков, JavaScript не требует, чтобы все значения в массиве относились к одному типу. Но если вы смешиваете в массиве разные типы, будьте внимательны при их использовании. Предположим, вы создали массив [1, 2, “fido”, 4, 5], а потом написали код, который проверяет, что все значения в массиве больше 2. Что произойдет, когда вы будете сравнивать строку “fido” и цифру 2? Чтобы избежать подобных бессмысленных операций, придется проверять тип каждого значения, прежде чем использовать его в коде. Конечно, это возможно (далее мы покажем, как это делается), но намного проще и безопаснее хранить в массивах однотипные данные.

В: Что произойдет при попытке обращения к массиву по слишком большому или малому индексу (допустим, меньше 0)?

О: Предположим, вы объявили массив:

```
var a = [1, 2, 3];
```

после чего пытаетесь обратиться к элементу a[10] или a[-1]. В обоих случаях будет получен результат undefined. Так что нужно либо следить за тем, чтобы в обращениях использовались только действительные индексы, либо проверять результат и убеждаться, что он отличен от undefined.

В: Первый элемент массива хранится в элементе с индексом 0. А как узнать индекс последнего элемента в массиве? Нужно ли точно знать, сколько элементов содержит массив?

О: Используйте свойство length, чтобы узнать размер массива. length всегда на 1 больше последнего индекса, так? Для получения последнего элемента можно написать:

```
myArray[myArray.length - 1];
```

JavaScript получает длину массива, уменьшает ее на 1, после чего получает значение по заданному индексу. Если массив содержит 10 элементов, то вы получите элемент с индексом 9 — то, что нужно. Этот прием можно использовать для получения последнего элемента массива, если вы точно не знаете, сколько элементов содержит массив.

Тем временем в фирме Bubbles-R-Us...

Директор Bubbles-R-Us.

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];
```

Новые результаты.

Что нужно построить.

Bubbles-R-Us

Вы сможете написать программу, которая будет выводить такой отчет? Мне это очень нужно для запуска новых растворов в производство!

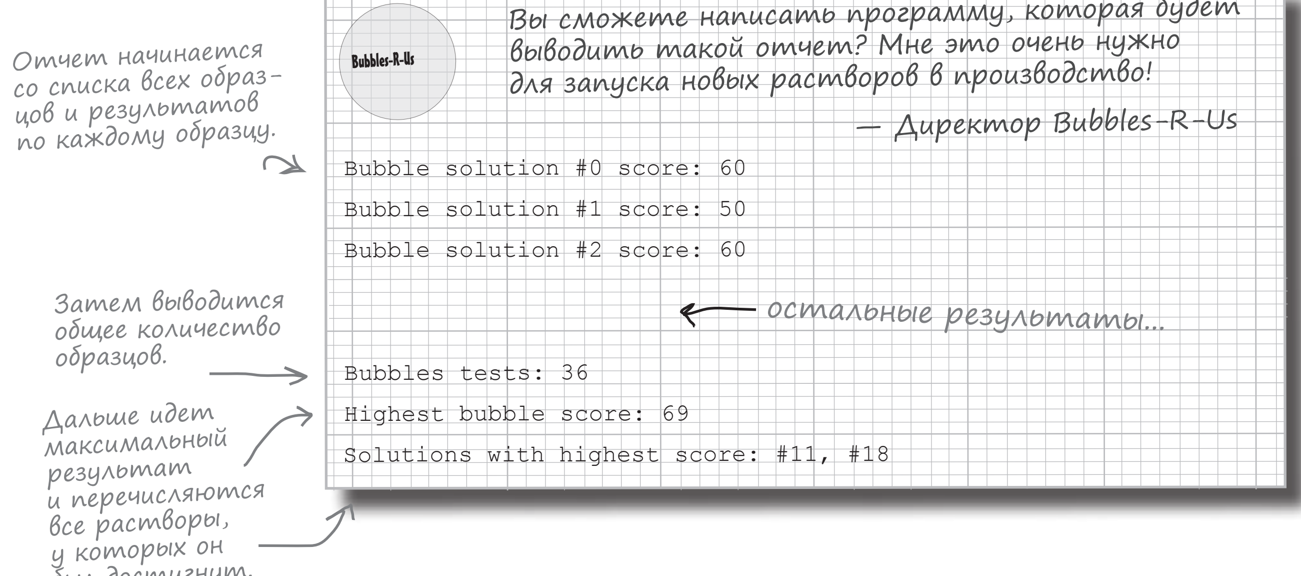
— Директор Bubbles-R-Us

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
```

← остальные результаты...

```
Bubbles tests: 36
Highest bubble score: 69
Solutions with highest score: #11, #18
```

Давайте повнимательнее присмотримся к отчету, который хочет получить директор:



Подумайте, как бы вы подошли к созданию этого отчета. Рассмотрите каждую часть отчета и подумайте, как сгенерировать правильные выходные данные. Запишите свои мысли.

Разговор в офисе

Джуди: Прежде всего нужно вывести каждый результат с номером раствора.

Джо: А номер раствора — это просто индекс в массиве, верно?

Джуди: Да, совершенно верно.

Фрэнк: Не торопитесь. Значит, нужно взять каждый результат, вывести его индекс, который совпадает с номером раствора, и затем вывести соответствующий результат.

Джуди: Именно так. А результат — соответствующее значение из массива.

Джо: Значит, для раствора № 10 результат определяется как `scores[10]`.

Джуди: Верно.

Фрэнк: Но растворов много. Как написать код, который будет выводить данные обо всех образцах?

Джуди: Перебор, друг мой.

Фрэнк: Ты имеешь в виду while?

Джуди: Верно, мы перебираем все значения от нуля до длины массива... то есть от длины минус 1, конечно.

Джо: Кажется, что-то прорисовывается. Можно браться за программирование; вроде бы уже все понятно.

Джуди: Согласна! Давайте сначала сделаем эту часть, а потом займемся всем остальным.

Как перебрать элементы массива

Наша цель — вывести данные, которые выглядят примерно так:

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
.
.
.
Bubble solution #35 score: 44
```

Это результаты с 3-го по 34-й... Мы пропустили их ради экономии бумаги (или битов, в зависимости от того, какую версию книги вы читаете).

0 1 2 3 4 5 6 7 8 9
60 50 60 58 54 54 58 50 52 54

Сначала выводится результат с индексом 0, затем то же самое делается для индексов 1, 2, 3 и так далее, пока не будет достигнут последний индекс в массиве. Вы уже знаете, как работают циклы while; давайте посмотрим, как использовать такой цикл для вывода всех результатов:

Вскоре мы покажем более удобный способ...

```
var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

var output;

var i = 0;
while (i < scores.length) {
  output = "Bubble solution #" + i + " score: " + scores[i];
  console.log(output);
  i = i + 1;
}
```

Переменная используется в цикле для построения выводимой строки.

Переменная для хранения текущего индекса.

Цикл продолжается, пока индекс остается меньше длины массива.

Затем мы создаем выходную строку с номером раствора (просто индекс массива) и результатом.

Строка выводится вызовом console.log.

Наконец, индекс увеличивается на 1 перед повторным прохождением цикла.

Развлечения с магнитами

Мы написали программу, проверяющую, в каких сортах мороженого есть кусочки жевательной резинки. Весь код был аккуратно разложен, но магниты упали на пол. Ваша задача — вернуть их на место. Будьте внимательны: кажется, пока магниты собирали с пола, среди них оказались несколько лишних. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
while (i < hasBubbleGum.length)
```

```
i = i + 2;
```

```
{
```

```
}
```

```
i = i + 1;
```

```
}
```

```
var i = 0;
```

```
{
```

```
{
```

```
if (hasBubbleGum[i])
```

```
while (i > hasBubbleGum.length)
```

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
```

```
var hasBubbleGum = [false,
                    false,
                    false,
                    true];
```

```
console.log(products[i] +
    " contains bubble gum");
```

Результат, который мы хотим получить.

Консоль JavaScript

```
Bubblegum contains bubble gum
```

Но постойте, существует и более удобный способ перебора!

Вероятно, нам стоит извиниться. Трудно поверить, что мы дошли уже до главы 4, и до сих пор не рассказали о цикле **for**. Цикл for можно считать «родственником» цикла while. По сути они делают одно и то же, хотя цикл for обычно немного удобнее. Возьмем только что использованный цикл while и посмотрим, как выглядит соответствующий цикл for.

```
Ⓐ var i = 0;
while Ⓑ (i < scores.length) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
  Ⓒ i = i + 1;
}
```

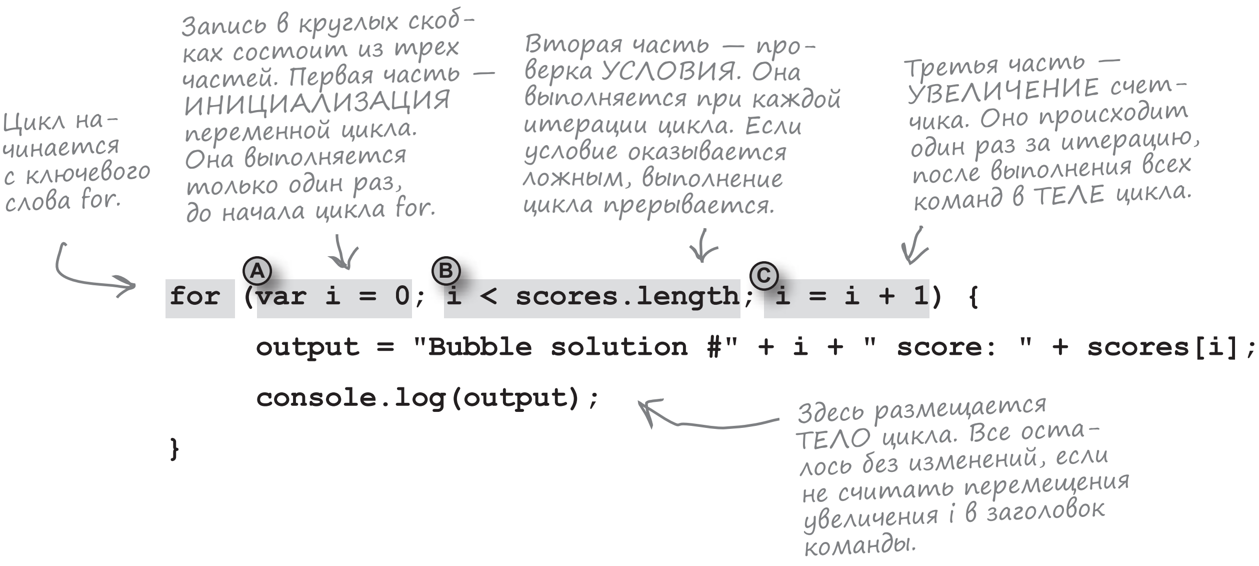
Сначала счетчик ИНИЦИАЛИЗИРУЕТСЯ.

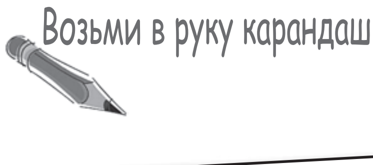
Затем счетчик проверяется в УСЛОВНОМ выражении.

Исполняемое ТЕЛО цикла — то есть все команды, заключенные в фигурные скобки { }.

Наконец, мы УВЕЛИЧИВАЕМ счетчик.

А теперь посмотрите, насколько цикл for упрощает выполнение этих операций:





Перепишите программу с магнитами (вернитесь на пару страниц назад) так, чтобы вместо цикла while в ней использовался цикл for. Если вам понадобится подсказка, обратитесь к описанию цикла while на предыдущей странице и посмотрите, какая часть цикла for соответствует той или иной части исходного цикла.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];

var hasBubbleGum = [false,
                    false,
                    false,
                    true];

var i = 0;

while (i < hasBubbleGum.length)

{

if (hasBubbleGum[i])

{

console.log(products[i] +
    " contains bubble gum");

}

i = i + 1;

}
```

Запишите свой код.

У нас есть все, что необходимо для первой части отчета. Собираем все вместе...

Стандартная разметка веб-страницы. Нам от нее много не требуется — ровно столько, чтобы создать сценарий.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    var scores = [60, 50, 60, 58, 54, 54,
                  58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51,
                  69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18,
                  41, 53, 55, 61, 51, 44];

    var output;

    for (var i = 0; i < scores.length; i = i + 1) {

        output = "Bubble solution #" + i +
                 " score: " + scores[i];

        console.log(output);

    }

  </script>
</head>
<body></body>
</html>
```

Массив результатов.

Цикл for, который будет использоваться для перебора результатов по всем образцам.

При каждом прохождении цикла создается строка со значениями i (номер образца) и scores[i] (количество пузырей, полученных из этого образца).

Строка выводится на консоль. Вот и все! Пришло время запустить отчет.

(Обратите внимание на разрыв длинной строки. Так можно делать, если при этом не разрывается текст, заключенный в кавычки. Здесь разбивка выполняется после оператора конкатенации (+), и поэтому проблем не возникает. Будьте внимательны и наберите текст точно так, как он приведен здесь.)

Тест-драйв

Сохраните файл с именем bubbles.html и загрузите его в браузере. Убедитесь, что окно консоли открыто (возможно, вам придется перезагрузить страницу, если консоль была активизирована после загрузки страницы). Просмотрите отчет, сгенерированный по данным Bubbles-R-Us.

Консоль JavaScript

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
Bubble solution #3 score: 58
Bubble solution #4 score: 54
Bubble solution #5 score: 54
Bubble solution #6 score: 58
Bubble solution #7 score: 50
Bubble solution #8 score: 52
Bubble solution #9 score: 54
Bubble solution #10 score: 48
Bubble solution #11 score: 69
Bubble solution #12 score: 34
Bubble solution #13 score: 55
Bubble solution #14 score: 51
Bubble solution #15 score: 52
Bubble solution #16 score: 44
Bubble solution #17 score: 51
Bubble solution #18 score: 69
Bubble solution #19 score: 64
Bubble solution #20 score: 66
Bubble solution #21 score: 55
Bubble solution #22 score: 52
Bubble solution #23 score: 61
Bubble solution #24 score: 46
Bubble solution #25 score: 31
Bubble solution #26 score: 57
Bubble solution #27 score: 52
Bubble solution #28 score: 44
Bubble solution #29 score: 18
Bubble solution #30 score: 41
Bubble solution #31 score: 53
Bubble solution #32 score: 55
Bubble solution #33 score: 61
Bubble solution #34 score: 51
Bubble solution #35 score: 44
```

То, что хотел директор.

Отчет со всеми результатами выглядит хорошо, можно переходить к следующему пункту — определению максимальных результатов. Но с их поиском все еще остаются проблемы. Чтобы немного упростить обнаружение «победителей», нужно поработать с остальными требованиями.

Беседа у камина

Сегодня в студии: циклы while и for отвечают на вопрос: «Кто важнее?»

Цикл WHILE

Вы что, меня разыгрываете? Эй? Я — *главный* среди циклов в JavaScript. Я не привязываюсь к глупому счетчику. Я могу использоваться с любыми типами условных конструкций. И кто-нибудь обратил внимание, что изучение циклов в этой книге началось именно с меня?

И еще одно: вы заметили, что у цикла FOR нет чувства юмора? Хотя, наверное, каждый, кому приходится постоянно повторять однообразные итерации, неизбежно станет таким же.

Честно говоря, сомнительное утверждение.

В этой книге было показано, что циклы FOR и WHILE делают фактически одно и то же. Как такое возможно?

Цикл FOR

Что-то мне не нравится этот тон.

Забавно. Но кто-нибудь обратил внимание, что в девяти случаях из десяти программисты используют цикл FOR?

Не говоря уже о том, что перебор фиксированного количества элементов массива в цикле WHILE — плохое, неуклюжее решение.

Ага, так ты согласен, что у нас больше общего, чем признавал сначала?

Я скажу тебе, как такое возможно...

Цикл WHILE

Цикл FOR

При использовании цикла WHILE необходимы разные команды, чтобы инициализировать счетчик и увеличивать его значение. Если после множества изменений в коде одна из этих команд случайно пропадет или переместится... ничем хорошим это не кончится. С циклом FOR все управление циклом упаковано прямо в команде FOR, находится на виду и никуда не пропадет.

Надо же, как удобно. Но большинство циклов, которые я видел, вообще были без счетчиков; они выглядели примерно так:

```
while (answer != "forty-two")
```

Попробуйте-ка сделать это в цикле FOR!

Пожалуйста:

```
for (;answer != "forty-two";)
```

Ха, и это работает?

Прекрасно работает.

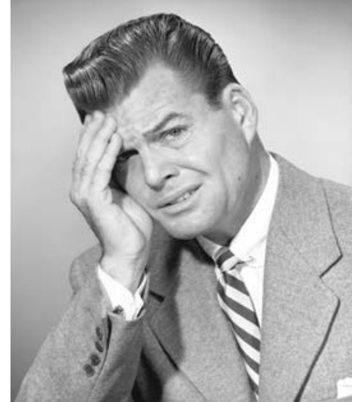
Неуклюжие выкрутасы.

И это все? Выходит, ты лучше только при использовании обобщенных условий?

Не только лучше — красивее.

Ах, извините, я и не понял, что у нас тут конкурс красоты.

Что, опять?..
Нельзя ли покороче?



В своих программах мы неоднократно использовали команды следующего вида:

Допустим, myImportantCounter содержит число — например, 0.

Берем переменную и увеличиваем ее на единицу.

```
myImportantCounter = myImportantCounter + 1;
```

После выполнения этой команды myImportantCounter содержит значение на 1 больше.

Эта команда встречается так часто, что в JavaScript для нее предусмотрена сокращенная запись. Она называется оператором постфиксного увеличения (инкремента), и несмотря на название, работает очень просто. С оператором постфиксного увеличения предыдущую строку кода можно заменить следующей:

Просто добавьте «++» за именем переменной.

```
myImportantCounter++;
```

После выполнения этой команды myImportantCounter содержит значение на 1 больше.

Было бы логично ожидать, что существует парный оператор постфиксного уменьшения. Применение его к переменной уменьшает ее значение на 1:

Просто добавьте «--» после имени переменной.

```
myImportantCounter--;
```

После выполнения этой команды значение myImportantCounter становится на 1 меньше.

Почему мы сейчас рассказываем об этом? Потому что этот оператор часто используется в циклах for. Сейчас мы немного «подчистим» код при помощи оператора постфиксного увеличения...

Доработка цикла for с оператором постфиксного увеличения

Сейчас мы немного доработаем свой код, протестируем его и убедимся в том, что он работает как прежде:

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

for (var i = 0; i < scores.length; i++) {

    var output = "Bubble solution #" + i +
                 " score: " + scores[i];

    console.log(output);

}
```

Единственное изменение: переменная цикла изменяется оператором постфиксного увеличения.

Короткий тест-драйв

Пробный запуск поможет убедиться в том, что переход на постфиксное увеличение не нарушил работу программы. Сохраните файл bubbles.html и перезагрузите программу. На экране должен появиться тот же отчет, который вы видели ранее.

```
Консоль JavaScript
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
```

Отчет выглядит точно так же.

Ради экономии места здесь не приводятся данные по всем образцам. Поверьте, они здесь.

Разговор в офисе продолжается...

Результаты по всем образцам выводятся, теперь нужно сгенерировать оставшуюся часть отчета.

Вы сможете написать программу, которая будет выводить примерно такой отчет? Мне это очень нужно для запуска новых растворов в производство!
— Директор Bubbles-R-Us

Bubbles-R-Us

Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
остальные результаты...
Bubbles tests: 36
Highest bubble score: 69
Solutions with highest score: #11, #18

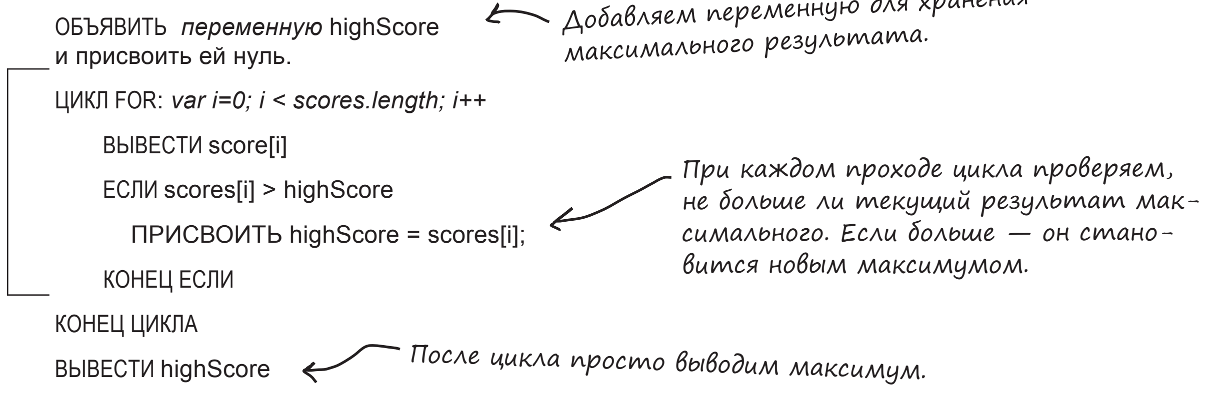
Джуди: Начинать нужно с определения общего количества образцов. Здесь все просто — это длина массива scores.

Джо: Точно. А еще нужно определить максимальный результат и вывести соответствующие номера образцов.

Джуди: Да, с последней задачей будет труднее всего. Начнем с определения максимального результата.

Джо: Верно, это будет хорошим началом.

Джуди: Думаю, нам понадобится переменная для хранения максимального результата, которая будет сравниваться с текущими результатами при переборе массива. Вот, я написала немного псевдокода:



```
ОБЪЯВИТЬ переменную highScore
и присвоить ей нуль.

ЦИКЛ FOR: var i=0; i < scores.length; i++
    ВЫВЕСТИ score[i]
    ЕСЛИ scores[i] > highScore
        ПРИСВОИТЬ highScore = scores[i];
    КОНЕЦ ЕСЛИ
КОНЕЦ ЦИКЛА
ВЫВЕСТИ highScore
```


Джо: Отлично; существующий код стал длиннее всего на несколько строк.

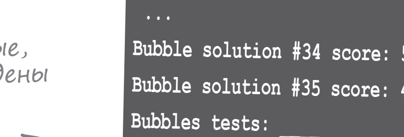
Джуди: При каждой итерации мы сравниваем текущий результат с максимумом `highScore`, и если он больше, то максимум обновляется. Остается только вывести максимальное значение после завершения цикла.

Возьми в руку карандаш

Напишите реализацию псевдокода на предыдущей странице, заполнив пропуски в приведенном ниже коде. Когда это будет сделано, попробуйте запустить программу в браузере — измените код в файле bubbles.html и перезагрузите страницу. Проверьте результаты в консоли, впишите в области консоли (внизу страницы) количество образцов и максимальный результат. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

var highScore = _____;
var output;
for (var i = 0; i < scores.length; i++) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (_____________ > highScore) {
        _____________ = scores[i];
    }
}
console.log("Bubbles tests: " + ____________);
console.log("Highest bubble score: " + _____________);
```

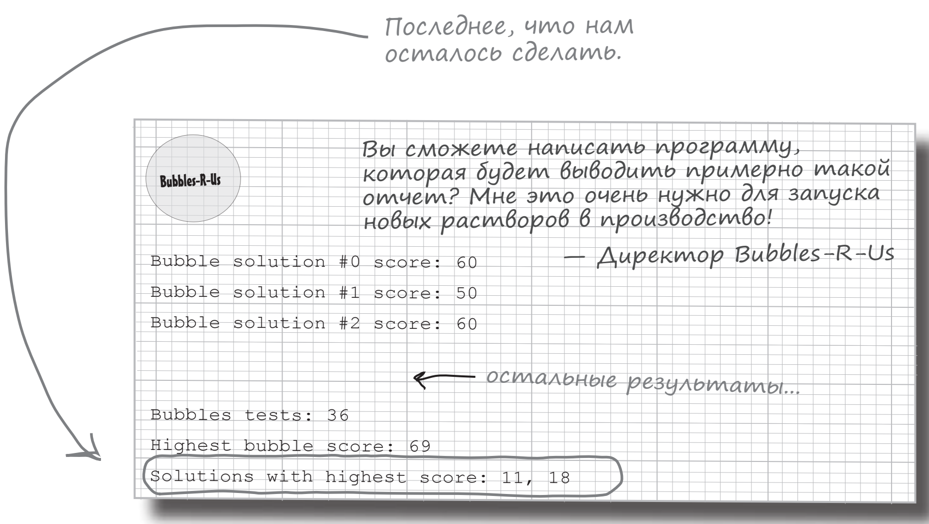
Заполните пустые места в коде...



Консоль JavaScript

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: _______
Highest bubble score: ______
```

«Может быть несколько»... Хммм... Если нам нужно сохранить несколько значений, что мы будем делать? Массив, конечно. Может быть, стоит перебрать массив scores, найти элементы с максимальным результатом и поместить их в массив, который будет позднее выводиться в отчете? Да, это возможное решение, однако сначала нужно научиться создавать пустые массивы и добавлять в них новые элементы.



Слздание пустого массива (и добавление элементов)

Чтобы завершить работу над кодом, сначала нужно узнать, как создать новый массив и как добавить в него элементы. Вы уже знаете, как создать массив с набором значений:

```
var genres = ["80s", "90s", "Electronic", "Folk"];
```

Этот список называется «литералом массива» — мы перечисляем, какие значения входят в массив.

Но также можно создать пустой массив без указания начальных элементов:

```
var genres = [];
```

Новый массив готов к работе; он не содержит элементов и имеет нулевую длину.

Это тоже литерал массива, хотя он и не содержит данных.

И вы уже знаете, как добавлять новые элементы в массив. Для этого достаточно присвоить значение элементу с указанием индекса:

```
var genres = [];
genres[0] = "Rockabilly";
genres[1] = "Ambient";
var size = genres.length;
```

Создается новый элемент массива со строкой "Rockabilly".

Здесь создается второй элемент массива, содержащий строку "Ambient".

Здесь size содержит значение 2 — длину массива.

При добавлении новых элементов необходимо следить за тем, по какому индексу размещается добавляемое значение. В противном случае вы создадите разреженный массив, то есть массив с «дырами» (например, массив со значениями в позициях 0 и 2, но без значения в позиции 1). Разреженный массив — это не обязательно плохо, но он требует особого внимания. А пока следует сказать, что существует другой способ добавления новых элементов, с которым не нужно беспокоиться об индексе. Это метод `push`, и вот как он работает:

```
var genres = [];
genres.push("Rockabilly");
genres.push("Ambient");
var size = genres.length;
```

Создает новый элемент по следующему доступному индексу (который равен 0) и присваивает ему значение "Rockabilly".

Создает еще один элемент в следующей свободной позиции (1 в данном случае) и присваивает ему значение "Ambient".

Часто Задаваемые Вопросы

В: Команда for содержит объявление переменной и ее инициализацию. Вы говорили, что объявления переменных следует размещать в начале файла. Как это понимать?

О: Да, размещайте объявления в начале файла — для локальных переменных, в начале функции — для локальных переменных. Но иногда разумнее объявить переменную непосредственно там, где она будет использоваться; циклы for — один из таких случаев. Как правило, переменные циклов (например, i) используются только для управления циклами и после завершения уже не нужны. Вы, конечно, можете использовать i в своем коде, но так поступать не рекомендуется. Таким образом, объявление прямо в команде for делает программный код лаконичнее и выразительнее.

В: Что означает myarray.push(value)?

О: Мы кое-что скрыли от вас: в JavaScript массив является особой разновидностью объектов. Как вы узнаете в следующей главе, с объектом могут быть связаны функции. Так что считайте push функцией, которая работает с myarray. В данном случае функция добавляет в массив новый элемент — значение, которое передается в аргументе для сохранения в массиве. Запись

```
genres.push("Metal");
```

означает, что вы вызываете функцию push и передаете ей строковый аргумент “Metal”. Функция push получает аргумент и добавляет его как новое значение в конец массива genres.

В: Можно ли чуть подробнее о том, что такое «разреженный массив»?

О: Разреженный массив содержит значения по небольшому подмножеству индексов, и не содержит значений в других позициях. Создать разреженный массив несложно:

```
var sparseArray = [ ];
sparseArray[0] = true;
spraseArray[100] = true;
```

В этом примере sparseArray содержит всего два значения с индексами 0 и 100 (true в обоих элементах). Значения по всем остальным индексам остаются неопределенными. Длина массива равна 101, но массив содержит всего два значения.

В: Предположим, имеется массив длиной 10. Я добавляю элемент с индексом 10 000. Что произойдет с индексами с 10 по 9999?

О: Все эти элементы будут содержать undefined. Если вы еще не забыли, значение undefined присваивается переменной, которая еще не была инициализирована. В общем, считайте, что вы создаете 9989 переменных, но не инициализируете их. Помните, что все эти переменные занимают память на компьютере, даже если они не содержат полезных данных. Будьте уверены в том, что у вас есть веские причины для создания разреженного массива.

В: Если я перебираю массив, и некоторые элементы содержат undefined, не нужно ли проверять значения перед использованием?

О: Если вы думаете, что массив может быть разреженным или содержит хотя бы одно значение undefined, то перед использованием элемента массива вам стоит убедиться в том, что его значение отлично от undefined. Если вы ограничиваетесь выводом значений на консоль, проблем не будет, но скорее всего, значения будут использоваться для более серьезных целей, например для каких-то вычислений. В таком случае попытка использовать undefined приведет к ошибке, или по крайней мере к неожиданному поведению программы. Чтобы проверить значение на undefined, используйте запись:

```
if (myarray[i] == undefined) {
  ...
}
```

Обратите внимание: undefined не заключается в кавычки (потому что это не строка, а значение).

В: Все массивы, создаваемые нами ранее, были литералами. Существует ли другой способ создания массивов?

О: Да. Возможно, вы видели синтаксис:

```
var myarray = new Array(3);
```

Эта команда создает новый массив с тремя пустыми позициями (то есть массив длины 3, который не содержит ни одного значения). Далее массив можно заполняться так, указывая значения myarray с индексами 0, 1 и 2. Пока вы не добавите значения, элементы myarray будут undefined.

Массив, созданный таким образом, ничем не отличается от обычного, созданного на базе литерала. На практике синтаксис с литералом применяется чаще, поэтому в оставшейся части книги мы чаще будем использовать именно эту форму записи.

Не обращайте внимания на нюансы синтаксиса (что такое «new», почему Array начинается с буквы в верхнем регистре и т. д.). Мы доберемся до них позднее!

Теперь мы умеем добавлять элементы в массив и можем закончить работу над отчетом. Ведь можно просто создать массив образцов с максимальным результатом во время перебора scores для поиска максимума?

Джуди: Да, мы начнем с пустого массива для образцов, дающих максимальный результат, и будем добавлять новый элемент при обнаружении очередного образца в процессе перебора массива scores.

Фрэнк: Прекрасно, начинаем.

Джуди: Подожди минутку... Я думаю, нам понадобится отдельный цикл.

Фрэнк: Точно? А мне кажется, что это можно сделать в существующем.

Джуди: Уверена, что понадобится. И вот почему: мы должны знать максимальный результат *до того*, как приступим к поиску всех образцов с таким результатом. Выходит, нам понадобятся два цикла: один для поиска максимального результата, который мы уже написали, и другой для поиска всех образцов с таким результатом.

Фрэнк: Ага, понятно. И во втором цикле мы будем сравнивать текущий результат с максимальным, и если они равны, добавлять индекс образца в новый массив, который создаем для образцов с максимальным результатом.

Джуди: Точно! Вот теперь начинаем.

Сможете ли вы написать цикл для поиска всех образцов, у которых результат совпадает с максимальным? Попробуйте сделать это, прежде чем перевернуть страницу и проверить, что же у нас получилось.

В переменной highScore уже хранится максимальный результат; мы можем использовать ее в своем коде.

Новый массив, в котором будут храниться индексы образцов с максимальным результатом.

```
var bestSolutions = [];
for (var i = 0; i < scores.length; i++) {



}
```

Запишите свой код.

Возьми в руку карандаш. Решение

Сможете ли вы написать цикл для поиска всех образцов, у которых результат совпадает с максимальным? Ниже приведено наше решение.

И снова мы начинаем с создания нового массива, который будет содержать все образцы с максимальным результатом.

Затем программа перебирает весь массив scores и отбирает элементы с максимальным результатом.

```
var bestSolutions = [];

for (var i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {

        bestSolutions.push(i);

    }
}

console.log("Solutions with the highest score: " + bestSolutions);
```

При каждой итерации результат элемента с индексом i сравнивается с highScore. Если они равны, то индекс добавляется в массив bestSolutions вызовом push.

В конце выводится список образцов с максимальным результатом. Обратите внимание на использование console.log для вывода массива bestSolutionsarray. Можно было создать еще один цикл для вывода элементов массива друг за другом, но к счастью, console.log делает это за нас (а если вы присмотритесь к выводу — еще и добавляет запятые между элементами!).

Взгляните на код из приведенного выше упражнения. А что, если функция push внезапно пропадет? Смогли бы вы переписать этот код без использования push? Запишите свою версию:

Тест-драйв итоговой версии отчета

Добавьте в файл bubbles.html код определения образцов с максимальным результатом и проведите еще один пробный запуск. Весь код JavaScript приведен ниже:

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

var highScore = 0;
var output;
for (var i = 0; i < scores.length; i++) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (scores[i] > highScore) {
        highScore = scores[i];
    }
}
console.log("Bubbles tests: " + scores.length);
console.log("Highest bubble score: " + highScore);

var bestSolutions = [];
for (var i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {
        bestSolutions.push(i);
    }
}
console.log("Solutions with the highest score: " + bestSolutions);
```

А вот и наши победители...

Образцы 11 и 18 показали наивысший результат: 69! Они заслуженно признаются победителями в этой группе образцов.

Консоль JavaScript

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
 ...
Bubble solution #34 score: 51
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11,18
```

Все верно, стоило бы использовать. Но с учетом того, что вы недавно узнали о функциях, мы решили изложить основы массивов, прежде чем вы начнете активно работать с ними. Тем не менее вам стоит думать о том, какие части кода можно абстрагировать в функцию. Мало того, представьте, что весь код, написанный для вычислений с пузырями, должен использоваться повторно (вами или другими разработчиками). Гораздо приятнее работать с удобным набором функций, чем с аморфным кодом.

Вернемся к только что написанному коду и переработаем его в набор функций. Такая переработка называется рефакторингом — мы изменим структуру кода, сделаем его более удобочитаемым, но при этом не будем менять функциональность. Другими словами, когда все будет сделано, код продолжит работать точно так же, как прежде, но при этом окажется гораздо лучше организован.

Краткий обзор кода...

Пройдемся по написанному нами коду и определим, какие части стоило бы абстрагировать в функции.

Код Bubbles-R-Us.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    var scores = [60, 50, 60, 58, 54, 54,
                  58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51,
                  69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18,
                  41, 53, 55, 61, 51, 44];

    var highScore = 0;
    var output;

    for (var i = 0; i < scores.length; i++) {
      output = "Bubble solution #" + i + " score: " + scores[i];
      console.log(output);
      if (scores[i] > highScore) {
        highScore = scores[i];
      }
    }
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    var bestSolutions = [];

    for (var i = 0; i < scores.length; i++) {
      if (scores[i] == highScore) {
        bestSolutions.push(i);
      }
    }

    console.log("Solutions with the highest score: " + bestSolutions);
  </script>
</head>
<body> </body>
</html>
```

Массив scores не должен объявляться в функциях, работающих с результатами, потому что они будут различаться при каждом использовании функции. Вместо этого мы будем передавать scores в аргументе, чтобы функции могли использовать массив scores для генерирования результатов.

Этот первый фрагмент кода используется для вывода всех результатов одновременно с вычислением максимального результата в массиве. Его можно вынести в функцию printAndGetHighScore.

Второй фрагмент кода предназначен для вычисления лучших результатов для максимального счета. Его можно вынести в функцию getBestResults.

Работа над функцией printAndGetHighScore

Код функции `printAndGetHighScore` уже готов, но чтобы преобразовать его в функцию, нужно решить, какие аргументы ему следует передавать и должен ли он вернуть какое-либо значение.

Передача массива результатов выглядит разумно — это даст нам использовать функцию с другими массивами, содержащими результаты. И функция будет возвращать максимальный результат, вычисленный в этой функции, чтобы вызвавший ее код мог сделать с результатом вызова что-нибудь полезное (она понадобится нам для определения лучших образцов).

И еще одно: обычно функция должна справляться с одной конкретной задачей, а наша решает сразу две: она выводит все данные из массива и вычисляет максимальный результат. В принципе ее можно было бы разбить на две функции, но в такой простой программе, как у нас, это излишне. Вероятно, в среде профессиональной разработки мы бы создали две функции, `printScores` и `getHighScore`, но пока ограничимся одной функцией. Итак, проведем рефакторинг кода:

Мы создаем функцию, получающую один аргумент — массив scores.

```
function printAndGetHighScore(scores) {
    var highScore = 0;
    var output;
    for (var i = 0; i < scores.length; i++) {
        output = "Bubble solution #" + i + " score: " + scores[i];
        console.log(output);
        if (scores[i] > highScore) {
            highScore = scores[i];
        }
    }
    return highScore;
}
```

Этот код выглядит точно так же. Точнее, он ВЫГЛЯДИТ точно так же, но теперь использует параметр scores вместо глобальной переменной scores.

Здесь добавлена одна строка, возвращающая highScore в точку вызова функции.

Рефакторинг кода с определением функции printAndGetHighScore

Теперь нужно изменить остальной код так, чтобы в нем использовалась новая функция. Для этого мы просто вызываем новую функцию и присваиваем переменной `highScore` результат функции `printAndGetHighScore`:

```
<!doctype html>
<html lang="en">
<head>
  <title>Bubble Factory Test Lab</title>
  <meta charset="utf-8">
  <script>
    var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

    function printAndGetHighScore(scores) {
        var highScore = 0;
        var output;
        for (var i = 0; i < scores.length; i++) {
            output = "Bubble solution #" + i + " score: " + scores[i];
            console.log(output);
            if (scores[i] > highScore) {
                highScore = scores[i];
            }
        }
        return highScore;
    }

    var highScore = printAndGetHighScore(scores);
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    var bestSolutions = [];

    for (var i = 0; i < scores.length; i++) {
      if (scores[i] == highScore) {
        bestSolutions.push(i);
      }
    }

    console.log("Solutions with the highest score: " + bestSolutions);
  </script>
</head>
<body> </body>
</html>
```

Наша новая функция готова к использованию.

Мы вызываем функцию, передаем массив scores и присваиваем возвращенное значение переменной highScore.

Этот код нужно преобразовать в функцию, а также внести соответствующие изменения в остальном коде.

Возьми в руку карандаш

Следующую функцию мы напишем вместе. Она должна создавать массив образцов, имеющих максимальный результат (таких образцов может быть несколько, поэтому-то мы и используем массив). Ей будет передаваться массив scores и значение highScore, вычисленное функцией printAndGetHighScore. Закончите приведенный ниже код. Ответ дан на следующей странице — не подглядывайте! Сначала выполните упражнение самостоятельно, только так вы по-настоящему разберетесь.

Исходная версия кода — на случай, если она вам понадобится.

```
var bestSolutions = [];
for (var i = 0; i < scores.length; i++) {
    if (scores[i] == highScore) {
        bestSolutions.push(i);
    }
}
console.log("Solutions with the highest score: " + bestSolutions);
```

Работа уже началась, но закончим мы ее вместе с вами!

```
function getBestResults(__________, ____________) {
    var bestSolutions = ______;
    for (var i = 0; i < scores.length; i++) {
        if (___________ == highScore) {
            bestSolutions.__________;
        }
    }
    return _______________;
}

var bestSolutions = _____________(scores, highScore);
console.log("Solutions with the highest score: " + bestSolutions);
```

А теперь все вместе...

Завершив рефакторинг своего кода, внесите все изменения в bubbles.html (посмотрите на приведенный код) и перезагрузите отчет. Результат должен оставаться таким же, как прежде. Но теперь код имеет существенно лучшую структуру и идеально подходит для повторного использования. Создайте собственный массив scores и поэкспериментируйте!

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Bubble Factory Test Lab</title>
  <script>
    var scores = [60, 50, 60, 58, 54, 54, 58, 50, 52, 54, 48, 69,
                  34, 55, 51, 52, 44, 51, 69, 64, 66, 55, 52, 61,
                  46, 31, 57, 52, 44, 18, 41, 53, 55, 61, 51, 44];

    function printAndGetHighScore(scores) {
        var highScore = 0;
        var output;
        for (var i = 0; i < scores.length; i++) {
            output = "Bubble solution #" + i + " score: " + scores[i];
            console.log(output);
            if (scores[i] > highScore) {
                highScore = scores[i];
            }
        }
        return highScore;
    }

    function getBestResults(scores, highScore) {
        var bestSolutions = [];
        for (var i = 0; i < scores.length; i++) {
            if (scores[i] == highScore) {
                bestSolutions.push(i);
            }
        }
        return bestSolutions;
    }

    var highScore = printAndGetHighScore(scores);
    console.log("Bubbles tests: " + scores.length);
    console.log("Highest bubble score: " + highScore);

    var bestSolutions = getBestResults(scores, highScore);
    console.log("Solutions with the highest score: " + bestSolutions);

  </script>
</head>
<body> </body>
</html>
```

Это новая функция getBestResults.

Результат этой функции используется для вывода списка лучших образцов в отчете.

Прекрасно! Так, и еще один пустяк... вы сможете определить самый эффективный по затратам образец? С этими данными весь рынок мыльных пузырей будет за нами. Вот вам массив с затратами на производство каждого образца для принятия решения.

Вот вам массив с затратами на производство каждого образца из массива scores.

```
var costs = [.25, .27, .25, .25, .25, .25,
             .33, .31, .25, .29, .27, .22,
             .31, .25, .25, .33, .21, .25,
             .25, .25, .28, .25, .24, .22,
             .20, .25, .30, .25, .24, .25,
             .25, .25, .27, .25, .26, .29];
```

Итак, что же нужно сделать? Нужно взять образцы с максимальным результатом и выбрать из них образец с минимальными затратами. К счастью, у нас имеется массив `costs`, повторяющий структуру массива `scores`, то есть затраты для элемента scores с индексом 0 хранятся в элементе массива `costs` с индексом 0 (.25), затраты для элемента scores с индексом 1 хранятся в элементе массива `costs` с индексом 1 (.27), и так далее. То есть затраты на производство каждого образца хранятся в элементе с тем же индексом, что и результат этого образца. Такие массивы называются *параллельными*:

Scores и costs — параллельные массивы, потому что для каждого результата из scores имеется соответствующее значение затрат в costs с тем же индексом.

```
var costs = [.25, .27, .25, .25, .25, .25, .33, .31, .25, .29, .27, .22, ..., .29];
```

В позиции 0 массива costs хранятся затраты на производство образца в позиции 0...

И так же с другими значениями затрат и результатов в массивах.

```
var scores = [60,  50,  60,  58,  54,  54,  58,  50,  52,  54,  48,  69,  ..., 44];
```

А вот это уже сложнее. Нужно не только отобрать максимальные результаты, но и выбрать среди них результат с наименьшими затратами?

Джуди: Максимальный результат мы уже знаем.

Фрэнк: Верно, но как его использовать? И у нас есть два массива. Как организовать их совместную работу?

Джуди: Думаю, любой из нас легко напишет простой цикл for, который снова перебирает массив `scores` и выбирает элементы с максимальным результатом.

Фрэнк: Да, это нетрудно. Но что потом?

Джуди: Каждый раз, когда находится результат, совпадающий с максимальным, мы проверяем, являются ли затраты на его производство наименьшими из всех найденных.

Фрэнк: Ага, понятно, нам понадобится переменная для хранения индекса «максимального результата с минимальными затратами»... Попробуй сама выговорить.

Джуди: Именно. И после перебора всего массива в переменной окажется индекс образца, который не только имеет максимальный результат, но и требует наименьших затрат.

Фрэнк: А если у двух образцов будут одинаковые затраты?

Джуди: Хмм... Надо подумать, что делать в таких случаях. Я бы поступила просто: выбираем первый из найденных образцов. Конечно, можно придумать более сложный критерий, но давай придерживаться этого варианта, если директор не будет возражать.

Фрэнк: Как все сложно... Пожалуй, нам стоит набросать псевдокод, прежде чем браться за программу.

Джуди: Согласна; управление индексами нескольких массивов сильно усложняет дело. Давай так и сделаем. Если подумать о будущем, планирование сэкономит нам время.

Фрэнк: Давай, я тут уже сделал первые наброски...

Я справился с псевдокодом. Посмотрите, что у меня получилось. Теперь ваша очередь — переведите его на JavaScript. И не забудьте свериться с ответом.

FUNCTION GETMOSTCOSTEFFECTIVESOLUTION (SCORE, COSTS, HIGHSCORE)

ОБЪЯВИТЬ *переменную* cost и присвоить ей 100.

ОБЪЯВИТЬ *переменную* index.

FOR: *var i=0; i < scores.length; i++*

ЕСЛИ результат образца в score[i] является максимальным

ЕСЛИ текущее значение затрат больше затрат образца

ТО

ПРИСВОИТЬ index значение переменной i

ПРИСВОИТЬ затраты на производство образца переменной cost

КОНЕЦ ЕСЛИ

КОНЕЦ ЕСЛИ

КОНЕЦ FOR

RETURN index

```
function getMostCostEffectiveSolution(scores, costs, highscore) {









}
var mostCostEffective = getMostCostEffectiveSolution(scores, costs, highScore);
console.log("Bubble Solution #" + mostCostEffective + " is the most cost effective");
```

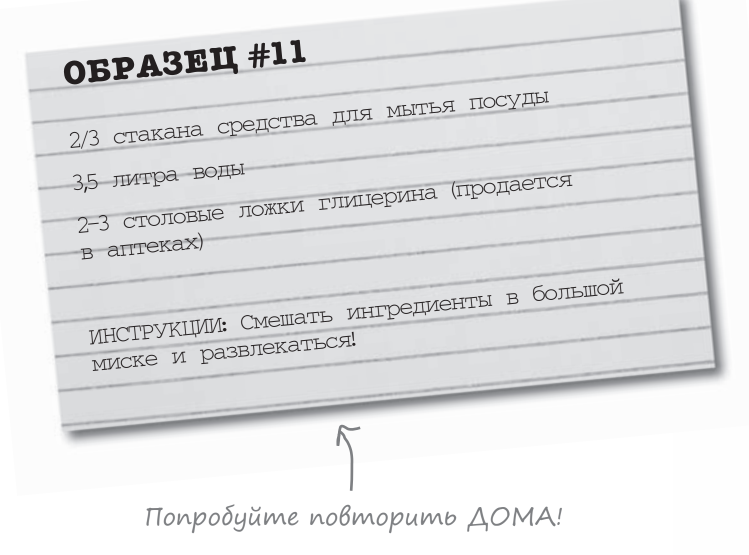
Переведите псевдокод на JavaScript.

ПОБЕДИТЕЛЬ: ОБРАЗЕЦ #11

Последний фрагмент кода помог определить НАСТОЯЩЕГО победителя — то есть раствор, позволяющий получить максимальное количество пузырей при минимальных затратах. Поздравляем, вы обработали большой массив данных и получили результаты, позволяющие компании Bubbles-R-Us принимать реальные бизнес-решения!

Если вы так же любопытны, как и мы, то вам интересно узнать, что же это за образец №11. Не нужно ничего искать; директор Bubble-R-Us сказал, что он с радостью предоставит вам секретный рецепт после такой самоотверженной работы.

Итак, рецепт образца №11. Расслабьтесь и выдуйте десяток-другой мыльных пузырей, чтобы привести в порядок массивы данных в своем мозгу перед тем, как переходить к следующей главе. Да, и не забудьте про список ключевых моментов этой главы!



Попробуйте повторить ДОМА!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Массивы представляют собой **структуры данных** для работы с упорядоченными данными.
- Массив содержит набор элементов, каждый из которых обладает **индексом**.
- Индексы массивов начинаются с нуля, соответственно индекс первого элемента равен нулю.
- Все массивы имеют свойство **length**, в котором хранится количество элементов в массиве.
- К любому элементу можно обратиться по индексу. Например, запись myArray[1] используется для обращения к элементу с индексом 1 (второй элемент массива).
- Если элемент не существует, при попытке обратиться к нему будет получено значение undefined.
- Присваивание существующему элементу приведет к изменению его текущего значения.
- Присваивание элементу, не существующему в массиве, приведет к созданию нового элемента.
- В элементах массивов могут храниться значения любого типа.
- Значения элементов массива не обязаны относиться к одному типу.
- Для создания новых массивов обычно используются **литералы массивов**.
- Пустой массив можно создать командой var myArray = [ ];
- Для перебора элементов массива обычно используется **цикл for**.
- Заголовок цикла for объединяет инициализацию переменной, проверку условия и изменение переменной.
- Цикл while чаще всего используется, когда количество итераций неизвестно, и цикл должен выполняться, пока условие остается истинным. При известном количестве выполнений обычно применяется цикл for.
- Если в середине массива существуют неопределенные элементы, такой массив называется разреженным.
- Значение переменной можно увеличить **оператором постфиксного увеличения** ++.
- Значение переменной можно уменьшить **оператором постфиксного уменьшения** --.
- Для добавления новых значений в массив можно использовать метод **push**.

Возьми в руку карандаш Решение

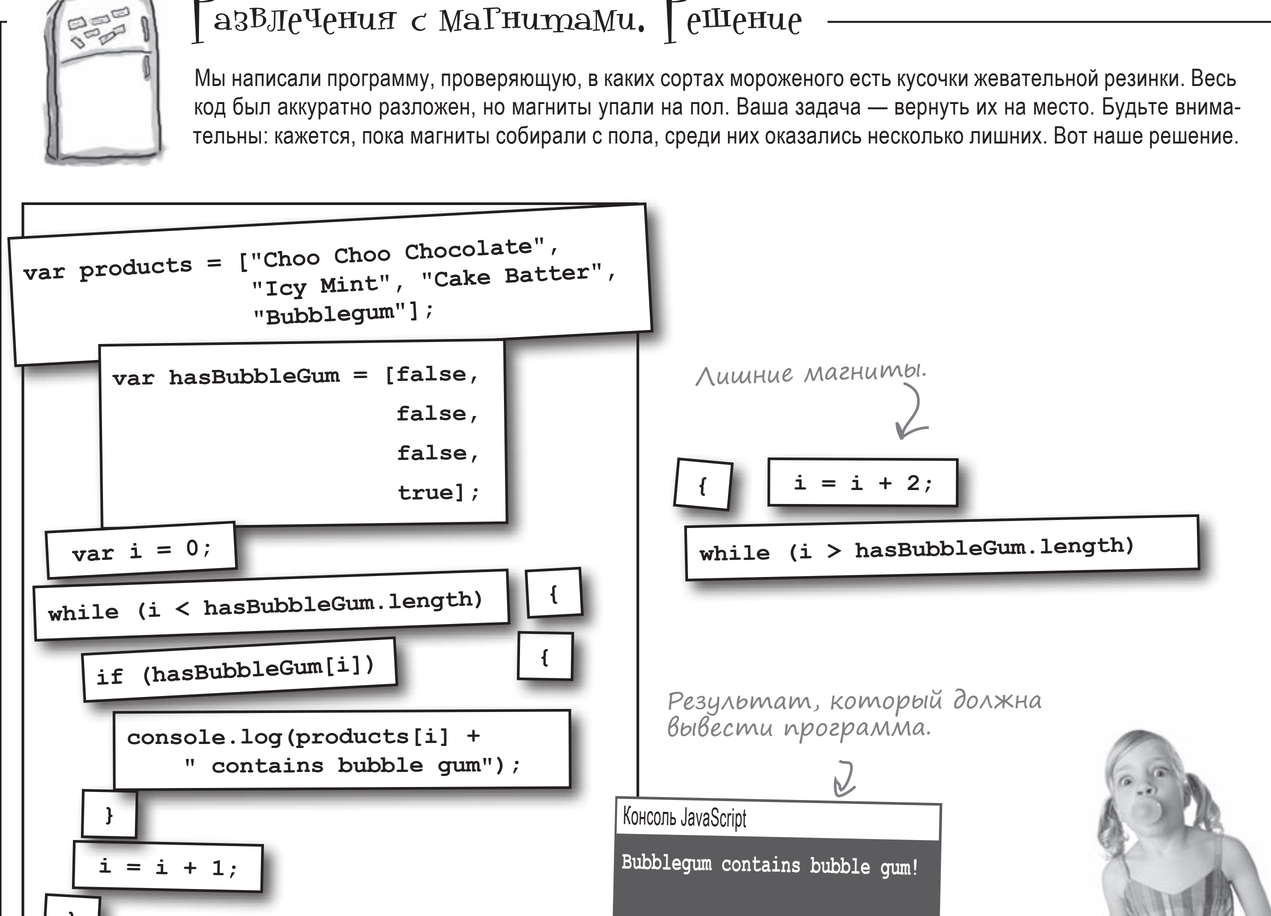
В массиве products хранятся названия сортов мороженого. Названия добавлялись в массив в порядке создания. Закончите код так, чтобы он определял и возвращал в переменной recent сорт, созданный *позже всех остальных*. Ниже приведено наше решение.

```
var products = ["Choo Choo Chocolate", "Icy Mint", "Cake Batter", "Bubblegum"];
var last = products.length - 1;
var recent = products[last];
```

Для получения индекса последнего элемента можно использовать длину массива, уменьшенную на 1. При длине 4 индекс последнего элемента равен 3 (так как индексы начинаются с нуля).

Развлечения с магнитами. Решение

Мы написали программу, проверяющую, в каких сортах мороженого есть кусочки жевательной резинки. Весь код был аккуратно разложен, но магниты упали на пол. Ваша задача — вернуть их на место. Будьте внимательны: кажется, пока магниты собирали с пола, среди них оказались несколько лишних. Вот наше решение.



Возьми в руку карандаш Решение

Перепишите программу с магнитами так, чтобы вместо while использовался цикл for. Если вам понадобится подсказка, обратитесь к описанию цикла while и посмотрите, какая часть цикла for соответствует той или иной части исходного цикла. Ниже приведено наше решение.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];

var hasBubbleGum = [false,
                    false,
                    false,
                    true];

var i = 0;
while (i < hasBubbleGum.length)
{
    if (hasBubbleGum[i])
    {
        console.log(products[i] +
            " contains bubble gum");
    }
    i = i + 1;
}
```

Запишите свой код.

```
var products = ["Choo Choo Chocolate",
                "Icy Mint", "Cake Batter",
                "Bubblegum"];
var hasBubbleGum = [false,
                    false,
                    false,
                    true];
for (var i = 0; i < hasBubbleGum.length; i = i + 1) {
    if (hasBubbleGum[i]) {
        console.log(products[i] + " contains bubble gum");
    }
}
```

Возьми в руку карандаш
Решение

Напишите реализацию псевдокода на предыдущей странице, заполнив пропуски в приведенном коде. Когда это будет сделано, попробуйте запустить программу в браузере — измените код в файле bubbles.html и перезагрузите страницу. Проверьте результаты в консоли, впишите в области консоли (внизу страницы) количество образцов и максимальный результат. Ниже приведено наше решение.

```
var scores = [60, 50, 60, 58, 54, 54,
              58, 50, 52, 54, 48, 69,
              34, 55, 51, 52, 44, 51,
              69, 64, 66, 55, 52, 61,
              46, 31, 57, 52, 44, 18,
              41, 53, 55, 61, 51, 44];

var highScore = 0;
var output;
for (var i = 0; i < scores.length; i++) {
    output = "Bubble solution #" + i + " score: " + scores[i];
    console.log(output);
    if (scores[i] > highScore) {
        highScore = scores[i];
    }
}
console.log("Bubbles tests: " + scores.length);
console.log("Highest bubble score: " + highScore);
```

Заполните пустые места в коде...

... а затем запишите в пустых полях данные, которые будут выведены на консоль.

Консоль JavaScript

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: 36
Highest bubble score: 69
```

Вот как выглядит наша реализация функции getMostCostEffectiveSolution, которая получает массив результатов, массив затрат и максимальный результат, и определяет индекс образца с максимальным результатом при минимальных затратах. Протестируйте код bubbles.html и убедитесь в том, что вы получили те же результаты.

Функция getMostCostEffectiveSolution получает массив результатов, массив затрат и максимальный результат.

```
function getMostCostEffectiveSolution(scores, costs, highscore) {
    var cost = 100;
    var index;

    for (var i = 0; i < scores.length; i++) {
        if (scores[i] == highscore) {
            if (cost > costs[i]) {
                index = i;
                cost = costs[i];
            }
        }
    }
    return index;
}
var mostCostEffective = getMostCostEffectiveSolution(scores, costs, highScore);
console.log("Bubble Solution #" + mostCostEffective + " is the most cost effective");
```

Сначала cost присваивается большое число, которое уменьшается при нахождении образца с меньшими затратами (и максимальным результатом).

Решение с минимальными затратами отслеживается в переменной cost...

... а индекс образца с минимальными затратами — в переменной index.

Перебираем массив scores так же, как прежде...

... и проверяем, является ли результат максимальным.

Если результат максимален, мы проверяем затраты. Если текущие затраты больше сохраненных, значит, мы нашли образец с меньшими затратами; его позиция (индекс в массиве) сохраняется в переменной index, а затраты — в cost (хранящей минимальные затраты по всем просмотренным элементам).

После завершения цикла в переменной index хранится индекс образца с минимальными затратами. Он возвращается коду, вызвавшему функцию.

Индекс (номер образца) выводится на консоль.

Из итогового отчета видно, что победителем стал образец №11: он обеспечивает максимальный результат с наименьшими затратами.

Консоль JavaScript

```
Bubble solution #0 score: 60
Bubble solution #1 score: 50
Bubble solution #2 score: 60
 ...
Bubble solution #34 score: 51
Bubble solution #35 score: 44
Bubbles tests: 36
Highest bubble score: 69
Solutions with the highest score: 11,18
Bubble Solution #11 is the most cost effective
```

ДОПОЛНИТЕЛЬНО: Эту функцию можно реализовать с помощью массива bestSolutions, чтобы не перебирать весь массив результатов заново. Вспомните: в массиве bestSolutions уже хранятся индексы образцов с максимальным результатом. При сравнении затрат можно использовать элементы bestSolutions для индексирования массива costs. Такой код чуть эффективнее, но чуть сложнее для чтения и понимания! Если вас интересует эта тема, реализацию можно найти в архиве с кодами на сайте wickedlysmart.com.

5 знакомьтесь: объекты

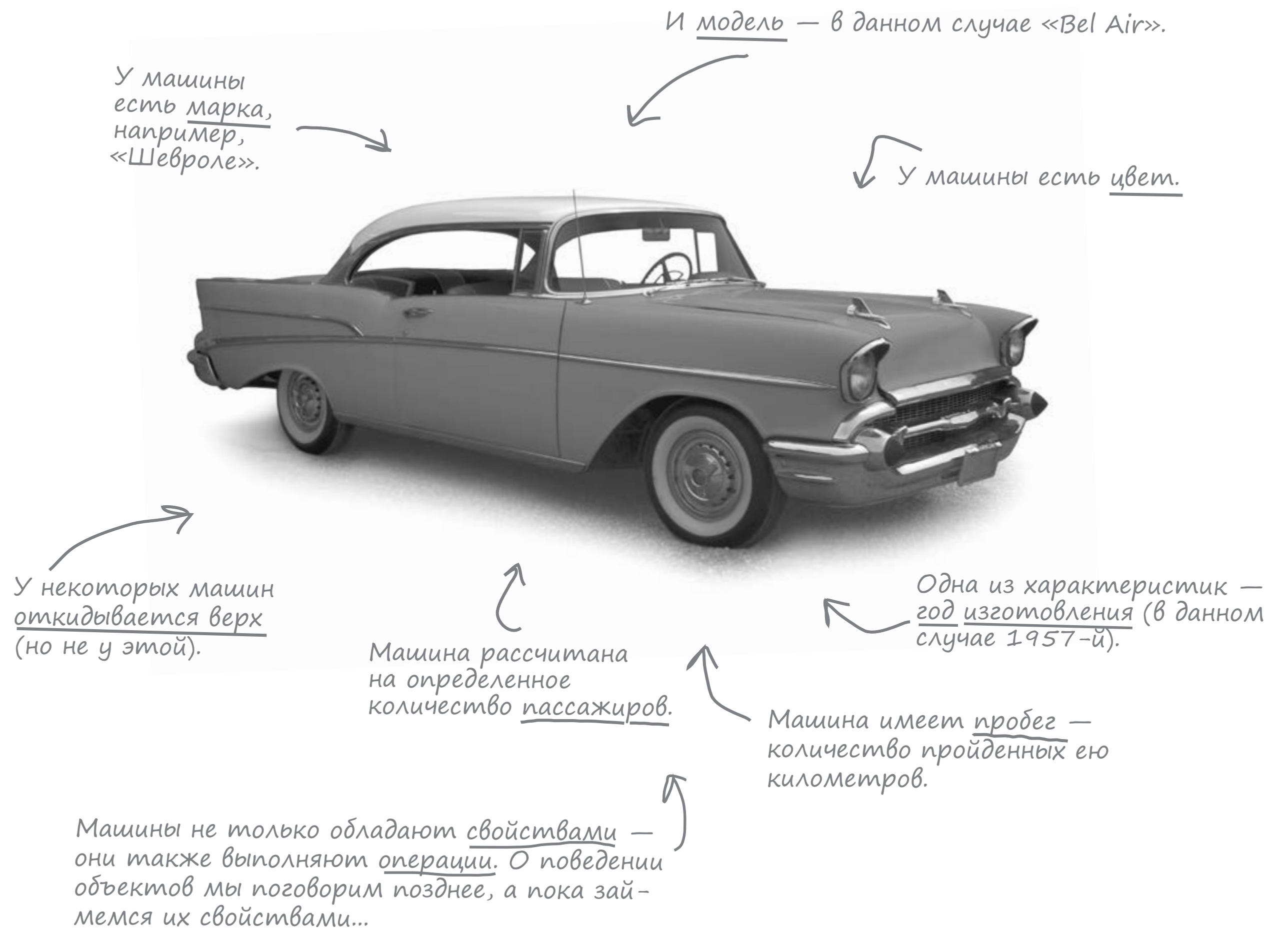
Поездка в Объектвиль

До настоящего момента мы использовали примитивы и массивы. И при этом применялась методология **процедурного программирования с простыми командами, условиями, циклами** for/while и функциями. Такой подход был далек от принципов **объектно-ориентированного программирования**. Собственно, он вообще не имел *ничего общего* с объектно-ориентированным программированием. Мы использовали объекты время от времени (причем вы об этом даже не знали), но еще не написали ни одного собственного объекта. Пришло время покинуть скучный процедурный город и заняться созданием собственных **объектов**. В этой главе вы узнаете, почему объекты сильно улучшают нашу жизнь — во всяком случае в области **программирования**. Так и знайте: привыкнув к объектам, вы уже не захотите возвращаться обратно. Да, и не забудьте прислать открытку, когда обживетесь.

Кто-то сказал «объекты»?!

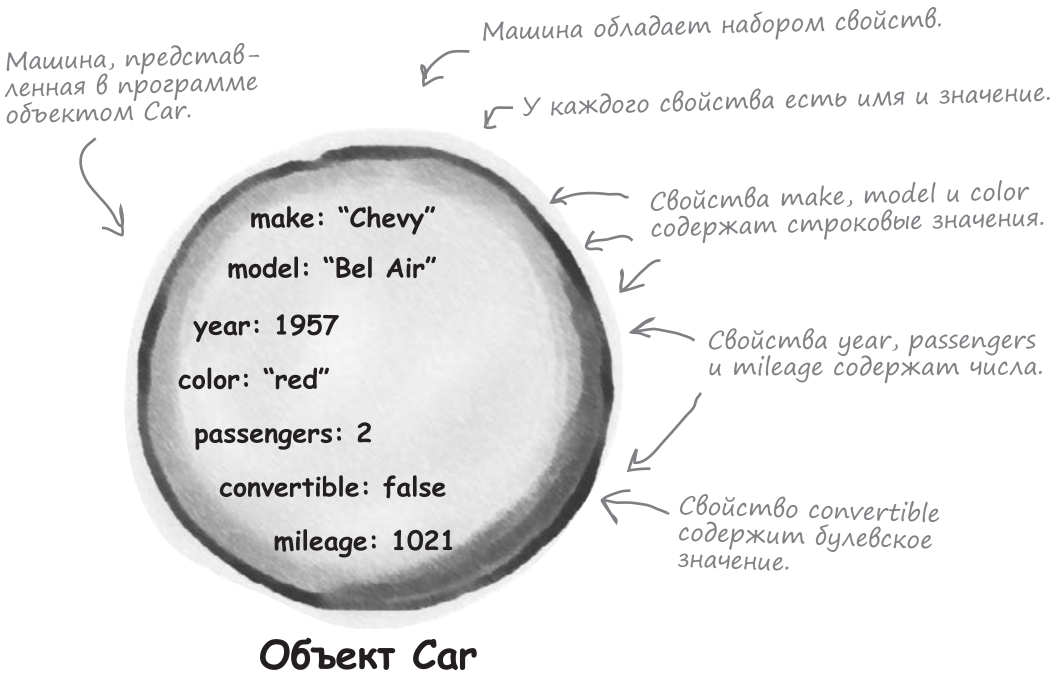
Ага, наша любимая тема! Объекты поднимут ваши навыки программирования на JavaScript на новый уровень. Они играют ключевую роль в управлении сложным кодом, в понимании модели документа браузера (мы займемся этим в следующей главе), в организации данных и даже в механизме упаковки многих библиотек JavaScript (эта тема тоже будет рассматриваться, но еще не скоро). Получается, что тема объектов достаточно сложна, верно? Ха! Мы сходу возьмемся за дело, и вы оглянуться не успеете, как начнете использовать объекты в своих программах.

Главный секрет объектов JavaScript: они представляют собой набор свойств. Для примера возьмем хотя бы автомобиль. У него есть свойства:



Подробнее о свойствах...

Конечно, характеристики настоящего автомобиля не ограничиваются несколькими свойствами, но в своей программе мы хотим отразить только эти свойства. Давайте рассмотрим их в контексте типов данных JavaScript:



Какие еще свойства вы включили бы в объект Car? Подумайте и запишите их внизу. Помните: в программе нужны далеко не все характеристики, существующие в реальном объекте.

Тонированные стекла смотрятся эффектно, но нужно ли их включать в представление объекта?

Возьми в руку карандаш

Мы начали строить таблицу имен и значений свойств объектов Car. Сможете ли вы закончить ее? Обязательно сверьте свои ответы с нашими, прежде чем двигаться дальше!

Запишите здесь имена свойств.

Здесь запишите соответствующие значения.

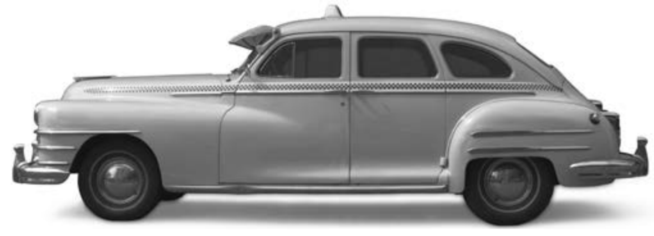
```
{
    make        : «Chevy»          ,
    model       : _______________ ,
    year        : _______________ ,
    color       : _______________ ,
    passengers  : _______________ ,
    convertible : _______________ ,
    mileage     : _______________ ,
    ___________ : _______________ ,
    ___________ : _______________
};
```

Запишите здесь свои ответы. При желании дополните список собственными свойствами.

Обратите внимание на элементы синтаксиса, которые мы разместили вокруг свойств и значений. Возможно, эта информация вам еще пригодится... Предупреждаем на всякий случай.

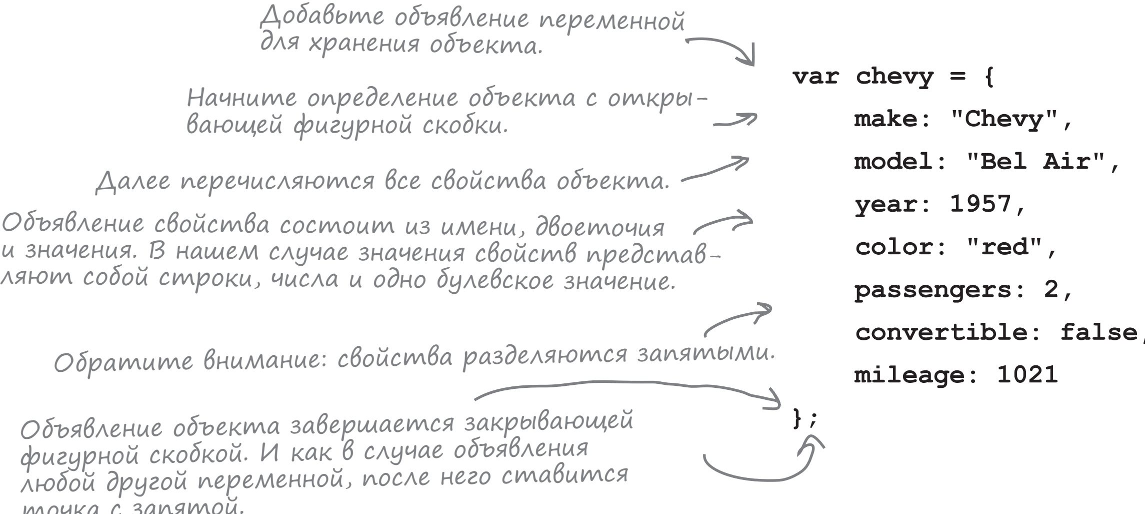
Мозговой штурм

А если машина — это такси? Какие свойства и значения будут у нее общими с Шевроле 1957 года? Какие могут отличаться? Какие дополнительные свойства может иметь (или не иметь) такси?



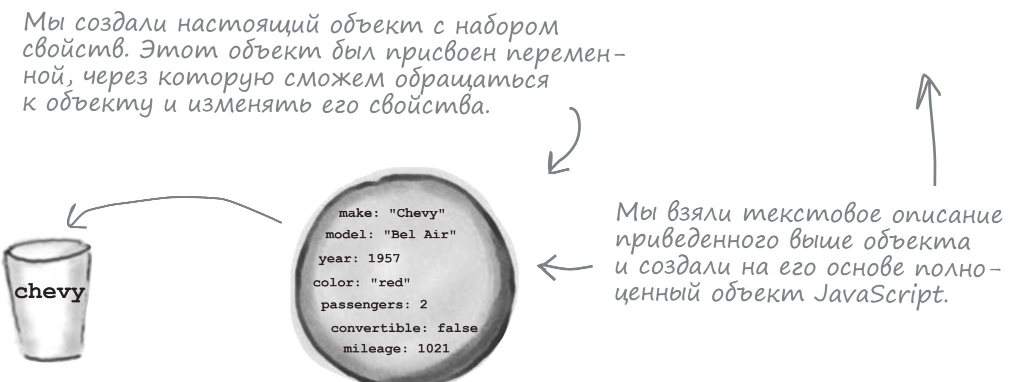
Как создать объект

У нас хорошие новости: после последнего упражнения *Возьми в руку карандаш* вы уже знаете многое из того, что необходимо для создания объектов. На самом деле нужно совсем немного: присвоить все то, что вы записали на предыдущей странице, переменной (чтобы вы могли выполнять операции с объектом после его создания). Это делается так:




```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

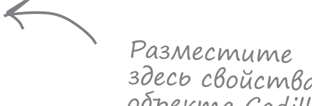
И что же мы получили? Новый объект, конечно. Рассматривайте объект как совокупность имен и значений (а проще говоря, свойств).

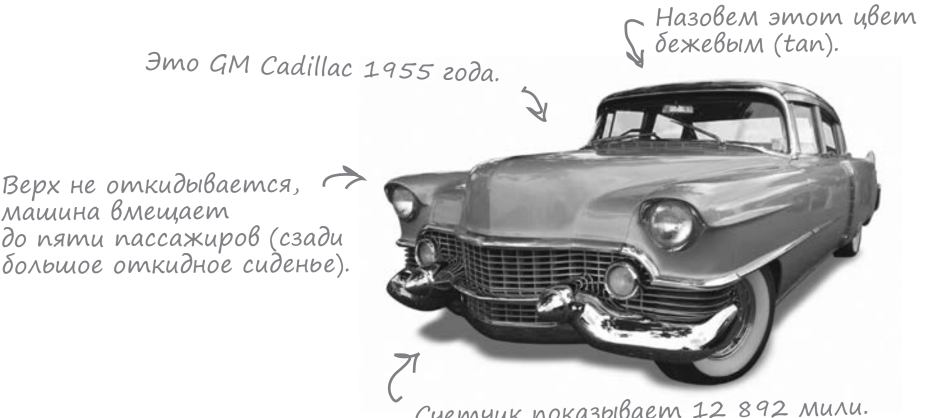


Теперь мы можем обратиться к объекту, передать его функции, прочитать значения свойств, изменить их, добавить новые свойства или удалить их из объекта. О том, как все это делается, будет рассказано чуть позже. А пока мы создадим еще несколько объектов для экспериментов...

Никто не заставляет вас ограничиваться всего одним объектом. Настоящая сила объектов (как вы вскоре увидите) проявляется в ситуациях, в которых вы создаете множество объектов и пишете код, способный работать с любым полученным объектом. Попробуйте создать с нуля новый объект — представляющий еще одну машину. Итак, напишите код для создания второго объекта.

```
var cadi = {

};
```





ШТРАФНАЯ КВИТАНЦИЯ

Выдана отделением полиции Вебвиль

№ 10

На этот раз вы легко отделались; вместо штрафа за превышение скорости мы предложим вам повторить «правила уличного движения» по созданию объектов.

Обязательно заключайте определение объекта в фигурные скобки:

```
var cat = {
  name: "fluffy"
};
```

Отделяйте имя свойства от значения двоеточием:

```
var planet = {
    diameter: 49528
};
```

Имя свойства может быть произвольной строкой, но обычно в них используются имена переменных:

```
var widget = {
    cost$: 3.14,
    "on sale": true
};
```

Если строка, используемая как имя свойства, содержит пробелы, ее необходимо заключить в кавычки.

Объект не может содержать два свойства с одинаковыми именами:

```
var forecast = {
    highTemp: 82,
    highTemp: 56
};
```

НЕВЕРНО! Работать не будет.

Пары «имя/значение» свойств разделяются запятыми:

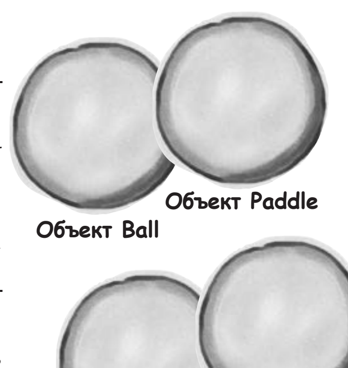
```
var gadget = {
  name: "anvil",
  isHeavy: true
};
```

После значения последнего свойства запятая не ставится:

```
var superhero = {
  name: "Batman",
  alias: "Caped Crusader"
};
```

Запятая здесь не нужна!

Что такое «объектно-ориентированный подход»?



До настоящего момента мы рассматривали задачу как совокупность объявлений переменных, условий, циклов for/while и вызовов функций. Это пример *процедурного* подхода: сначала это, потом то, и так далее. В *объектно-ориентированном* программировании задача рассматривается в контексте объектов, обладающих состоянием (уровень масла и бензина у машины) и поведением (машину можно завести, припарковать и заглушить двигатель).

Для чего это нужно? Объектно-ориентированное программирование позволяет мыслить на более высоком уровне. Как бы вы поджарили ломтик хлеба по первому принципу? Сделать нагревательную спираль из проволоки, подключить ее к источнику питания, включить и держать хлеб достаточно близко к спирали (а еще подождать, пока он поджарится, и отключить нагреватель). По второму принципу вы просто используете тостер (поместить хлеб в тостер, нажать кнопку). Первый подход — процедурный, второй — объектно-ориентированный: у вас имеется объект-тостер, который содержит простой и удобный метод для поджаривания вставленного ломтика хлеба.

Что вам нравится в объектно-ориентированном программировании?

«Оно помогает мне проектировать системы более естественным образом. Появляется возможность развития».

Джой, 27 лет, специалист по архитектуре ПО

«Мне не приходится изменять уже протестированный код, чтобы добавить новую возможность».

Брэд, 32 года, программист

«Мне нравится, что данные и методы, работающие с этими данными, объединяются в одном объекте».

Джош, 22 года, любитель пива

«Возможность повторного использования кода в других приложениях. Создавая новый объект, я могу сделать его достаточно гибким, чтобы в будущем его можно было использовать в другом приложении».

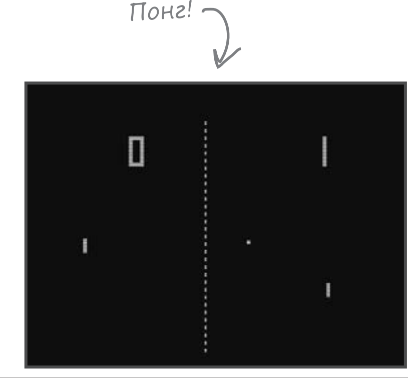
Крис, 39 лет, руководитель проекта

«Ушам своим не верю — это сказал Крис? Он не написал ни строки кода за последние пять лет».

Дэрил, 44 года, работает на Криса

Мозговой штурм

Предположим, вам поручено реализовать классическую видеоигру «пинг-понг». Какие объекты вы бы определили в своей игре? Как вы думаете, каким поведением и состоянием они должны обладать?



```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};
```

Как работают свойства

Итак, все свойства упакованы в объекте. Что теперь? Теперь вы можете получать значения свойств, изменять их, добавлять новые свойства и удалять их и вообще выполнять любые вычисления с ними. Давайте посмотрим, как эти операции реализуются на JavaScript.

Как обратиться к свойству. Чтобы обратиться к свойству объекта, укажите имя объекта, поставьте точку, а затем укажите имя свойства. Этот синтаксис, часто называемый «точечной записью», выглядит так:

Такое свойство может использоваться в любом выражении:

```
var miles = fiat.mileage;
if (miles < 2000) {
   buyIt();
}
```

Сначала указывается переменная, в которой хранится объект, затем точка и имя свойства.

Точечная запись.

- Точечная запись (.) открывает доступ к свойствам объекта.
- Например, `fiat.color` — свойство объекта `fiat` с именем `color` и значением «Medium Blue».

Как изменить свойство. Значение свойства может быть изменено в любой момент; для этого достаточно присвоить свойству новое значение. Предположим, мы хотим сбросить пробег нашего «Фиата» до 10 000. Это делается так:

```
fiat.mileage = 10000;
```

Просто укажите изменяемое свойство и его новое значение.

Как добавить новое свойство. Объект можно в любой момент дополнить новыми свойствами. Для этого достаточно указать новое свойство и присвоить ему значение. Допустим, мы хотим добавить булевское свойство, которое сообщает, пора ли помыть «Фиат»:

```
fiat.needsWashing = true;
```

Если ранее это свойство в объекте не существовало, оно добавляется в объект. В противном случае происходит обновление текущего значения этого свойства.

Новое свойство добавляется в объект.

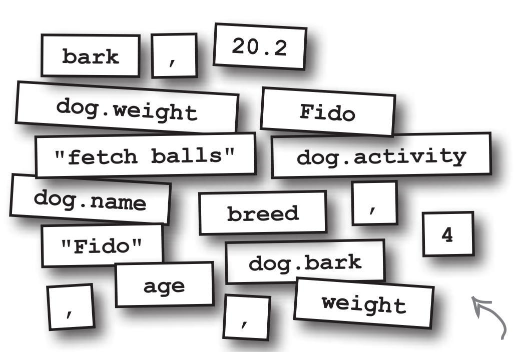
Как выполнять вычисления со свойствами. Все просто: используйте свойство точно так же, как бы вы использовали любую переменную (или произвольное значение). Несколько примеров:

Свойство объекта используется так же, как переменная, — если не считать использования точечной записи для обращения к свойству объекта.

```
if (fiat.year < 1965) {
    classic = true;
}
for (var i = 0; i < fiat.passengers; i++) {
    addPersonToCar();
}
```

Развлечения с магнитами

Потренируйтесь в создании объектов и использовании точечной записи, расставьте магниты по местам. Будьте внимательны, в набор могли затесаться посторонние магниты!



Используйте эти магниты для завершения кода.

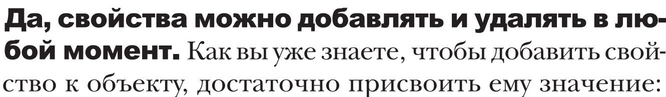
```
name: "Fido"
weight: 20.2
age: 4
breed: "mixed"
activity: "fetch balls"
```

Объект dog.

```
var dog = {
    name: _________
    _________: 20.2
    age: ________
    ________: "mixed",
    activity: ____________
};
var bark;
if (_____________ > 20) {
   bark = "WOOF WOOF";
} else {
   bark = "woof woof";
}
var speak = __________ + " says " + ________ + " when he wants to " + ____________;
console.log(speak);
```

Фидо надеется, что вы правильно зададите его свойства.

Насколько я понял, новые свойства можно добавлять в любое время. А удалять их тоже можно?



Да, свойства можно добавлять и удалять в любой момент. Как вы уже знаете, чтобы добавить свойство к объекту, достаточно присвоить ему значение:

```
fido.dogYears = 35;
```

С этого момента в `fido` появляется новое свойство `dogYears`. Все просто.

Для удаления свойств используется специальное ключевое слово — `delete` (кто бы мог подумать?). Ключевое слово `delete` используется примерно так:

```
delete fido.dogYears;
```

При удалении свойства вы уничтожаете не только его значение, но и само свойство. Если после удаления вы попытаетесь обратиться к свойству `fido.dogYears`, результат будет равен `undefined`.

Выражение `delete` возвращает `true`, если удаление свойства прошло успешно. `delete` вернет `false` только в том случае, когда свойство не было удалено (например, если свойство входит в защищенный объект, принадлежащий браузеру). `true` возвращается даже в том случае, если свойство, которое вы пытаетесь удалить, не существует в объекте.

Часто задаваемые вопросы

В: Сколько свойств может иметь объект?

О: Сколько угодно. Можно создать объект без свойств или же объект с сотнями свойств. Все зависит только от вас.

В: Как создать объект без свойств?

О: Так же, как любой другой объект, просто опустите список свойств:

```
var lookMaNoProps = { };
```

В: Я только что спрашивал, как создать объект без свойств... Но для чего это может быть нужно?

О: Например, можно создать пустой объект, а потом добавлять свойства динамически в зависимости от логики кода. Такой способ создания объекта станет понятнее в будущем, когда у вас будет больше опыта использования объектов.

```
var lookMaNoProps = { };
lookMaNoProps.age = 10;
if (lookMaNoProps.age > 5) {
  lookMaNoProps.school = "Elementary";
}
```

В: Чем объект лучше простого набора переменных? В конце концов, ничто не мешает нам представить каждое из свойств объекта fiat отдельной переменной, верно?

О: Объекты скрывают сложность данных, чтобы разработчик мог сосредоточиться на высокоуровневой структуре кода, а не на технических подробностях. Допустим, вы хотите написать модель дорожного движения с десятками машин. Конечно, вы предпочтете мыслить на уровне машин, светофоров и дорожных объектов, а не сотен мелких переменных. Объекты упрощают жизнь еще и тем, что инкапсулируют (или скрывают) сложность состояния и поведения объектов, чтобы вам не приходилось на них отвлекаться. Вскоре вы лучше поймете, как все это работает; нужно лишь накопить немного опыта работы с объектами.

В: Что произойдет при попытке добавить новое свойство в объект, уже содержащий свойство с таким именем?

О: Если вы попытаетесь добавить новое свойство (допустим, needsWashing) в объект fiat, который уже содержит свойство needsWashing, это приведет к изменению текущего значения свойства. Итак, если вы выполняете команду:

```
fiat.needsWashing = true;
```

а объект fiat уже содержит свойство needsWashing со значением false, тогда это значение будет заменено на true.

В: Что произойдет при обращении к несуществующему свойству? Допустим, если я выполняю команду

```
if (fiat.make) { ... }
```

но объект fiat не содержит свойства make?

О: Результат выражения fiat.make будет равен undefined, если fiat не содержит свойства с именем make.

В: Что произойдет, если поставить запятую после последнего свойства?

О: В большинстве браузеров это не приведет к ошибке. Однако в старых версиях некоторых браузеров выполнение кода JavaScript может быть прервано. Итак, если вы хотите, чтобы ваш код работал как можно в большем количестве браузеров, старайтесь избегать лишних запятых.

В: Можно ли воспользоваться функцией console.log для вывода объекта на консоль?

О: Можно. Просто выполните команду:

```
console.log(fiat);
```

Если теперь загрузить страницу с открытой консолью, то на консоль выводится информация об объекте.

```
JavaScript console
> console.log(fiat)
Object {make: "Fiat", model: "500", year: 1957,
color: "Medium Blue", passengers: 2…}
>
```

Как объект хранится в переменной? Любознательные умы интересуются...

Вы уже видели, что переменная работает как «контейнер», хранящий значение. Но числа, строки и булевские значения занимают относительно мало места. А как объекты? Сможет ли переменная вместить объект любого размера независимо от того, сколько свойств он содержит?

- **В переменных не хранятся собственно объекты.**
- **Вместо этого в них хранятся ССЫЛКИ на объекты.**
- **Ссылка напоминает указатель или адрес объекта.**
- **Иначе говоря, переменная содержит не сам объект, а нечто вроде указателя. И в JavaScript мы не знаем, *что* хранится в переменной. Зато мы *знаем* — что бы это ни было, оно указывает на наш объект.**
- **При использовании точечной записи интерпретатор JavaScript берет на себя обращение по ссылке к объекту и последующие операции с его свойствами.**

Итак, объект невозможно вместить в переменную, хотя мы часто думаем о происходящем именно так. Но не существует гигантских расширяемых переменных, подстраивающихся под любой размер объекта. Просто объектная переменная содержит ссылку на объект.

На ситуацию также можно взглянуть немного иначе: примитивная переменная представляет фактическое значение переменной, тогда как объектная переменная представляет *способ обращения к объекту*. На практике объекты достаточно представлять себе... как объекты: собаки, машины и т. д, а не как ссылки. И все же если вы будете знать, что переменные содержат *ссылки, это пригодится вам в будущем (вы убедитесь в этом через несколько страниц)*.

Или еще одна интерпретация: применение точечной записи (.) к ссылочной переменной как бы говорит «использовать ссылку *перед* точкой для получения объекта, содержащего свойство *после* точки». (Если потребуется, перечитайте это предложение несколько раз.) Например

```
car.color;
```

означает: «использовать объект, на который ссылается переменная `car`, для обращения к свойству `color`».

Сравнение примитивов с объектами

Ссылку на объект можно рассматривать как обычное значение переменной, хранящееся в ней так же, как хранятся примитивные значения. Примитивные переменные содержат такие значения, как 5, –26.7, «hi» или false. Ссылочная переменная содержит ссылку: значение, представляющее механизм обращения к конкретному объекту.

Это примитивные переменные. Каждая переменная содержит то значение, которое в нее было помещено.

Это ссылочная переменная; в ней хранится ссылка на объект.

Инициализация примитивной переменной

При объявлении и инициализации примитивной переменной указывается ее значение, которое сохраняется непосредственно в переменной:

```
var x = 3;
```

В переменной хранится число 3.

Числовое примитивное значение.

Инициализация объектной (ссылочной) переменной

При объявлении и инициализации объекта используется объектная запись, но сам объект в переменной не поместится. Вместо объекта в переменной сохраняется *ссылка* на объект.

```
var myCar = {...};
```

В переменной сохраняется ссылка на объект Car.

Сам объект Car в переменной не хранится!

Ссылочное значение.

За сценой

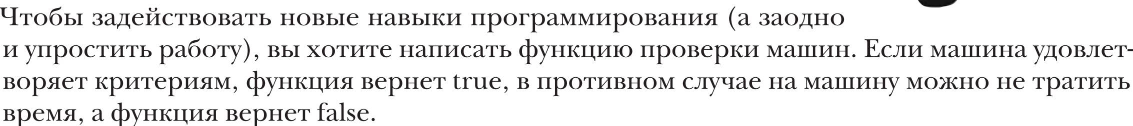
Мы не знаем, как интерпретатор JavaScript представляет ссылки на объекты (да в общем-то это для нас и не важно).

Мы знаем лишь то, что точечная запись позволяет обратиться к объекту и его свойствам.

Объекты способны на большее...

Допустим, вы подбираете машину для перемещений по Объектвилю. Какие использовать критерии? Как вам, например:

- ❒ Год выпуска — 1960-й или ранее.
- ❒ Пробег — 10 000 миль и менее.



Чтобы задействовать новые навыки программирования (а заодно и упростить работу), вы хотите написать функцию проверки машин. Если машина удовлетворяет критериям, функция вернет true, в противном случае на машину можно не тратить время, а функция вернет false.

А если говорить конкретнее, мы напишем функцию, которая получает объект *car в параметре, проверяет его и возвращает булевское значение. Функция должна работать с любыми объектами car.*

Давайте, попробуем:

Это функция

Функции будет передаваться объект car.

```
function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}
```

Для проверки свойств mileage (пробег) и year (год выпуска) достаточно применить точечную запись к параметру car.

Значение каждого из свойств проверяется по заданному критерию.

Если какая-либо из проверок не пройдена, функция возвращает false. В противном случае возвращается true — проверка пройдена успешно!

Давайте проверим, как работает функция. Для начала понадобится объект car. Как насчет этого:

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

Что вы думаете по поводу этой машинки? Нужно ли нам желтое такси? Обоснуйте свое решение.

Предварительная проверка

Довольно разговоров про объекты! Давайте создадим объект и проверим его, передав функции `prequal`. Откройте простейшую страницу HTML (prequal.html), включите в нее приведенный ниже код, загрузите страницу и посмотрите, пройдет ли проверку объект taxi:

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};

function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}

var worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

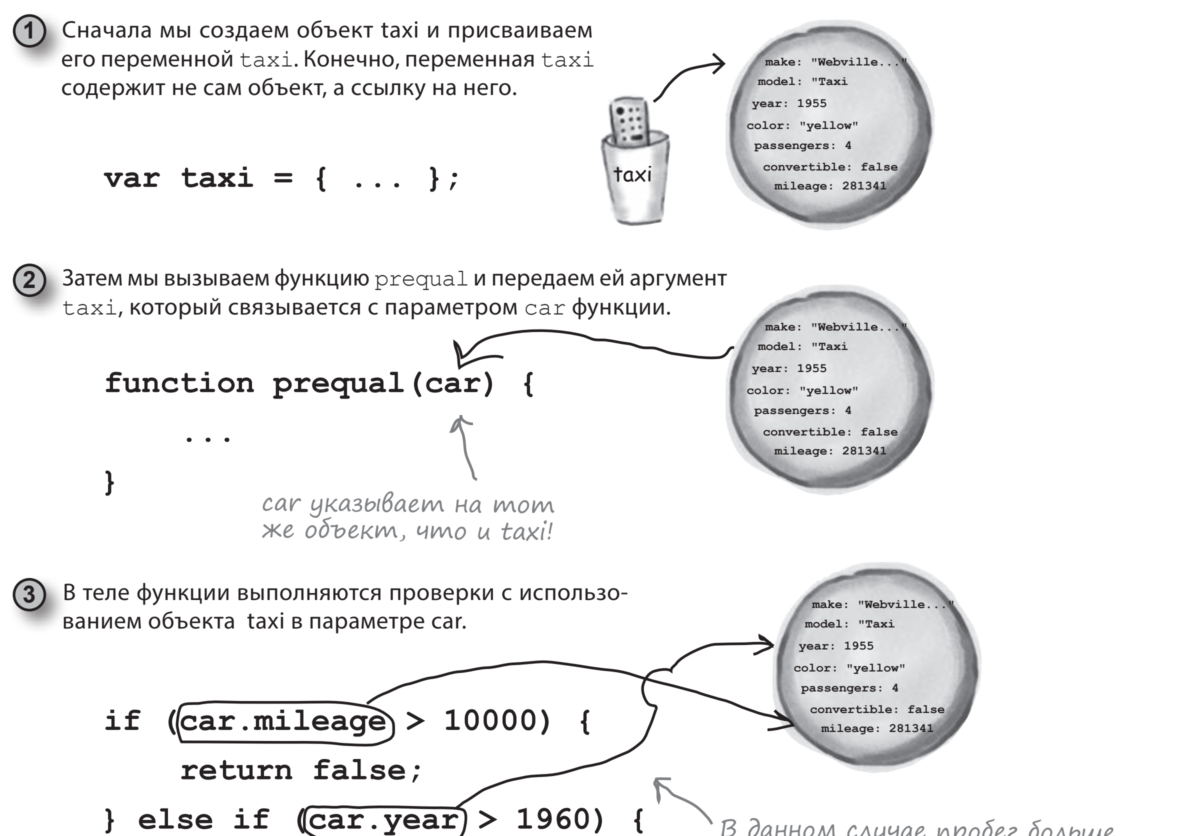
И что получилось?

А вот что получилось... Давайте быстро пройдемся по коду на следующей странице и посмотрим, почему же объект taxi был отвергнут проверочной функцией...

Консоль JavaScript

```
You should really pass on the Webville Motors Taxi
```

Проверка шаг за шагом



1. Сначала мы создаем объект taxi и присваиваем его переменной `taxi`. Конечно, переменная `taxi` содержит не сам объект, а ссылку на него.

```
var taxi = { ... };
```

2. Затем мы вызываем функцию `prequal` и передаем ей аргумент `taxi`, который связывается с параметром `car` функции.

```
function prequal(car) {
    ...
}
```

3. В теле функции выполняются проверки с использованием объекта taxi в параметре car.

```
if (car.mileage > 10000) {
    return false;
} else if (car.year > 1960) {
    return false;
}
```

4. К сожалению, такси уже набрало основательный пробег, поэтому первая проверка car.mileage > 10 000 дает результат true. Функция возвращает false, и переменной `worthALook` также присваивается false. На консоль выводится сообщение о том, что машина не прошла проверку.

Функция prequal возвращает false, поэтому выводится сообщение...

```
var worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

Консоль JavaScript

```
You should really pass on the Webville Motors Taxi
```

Возьми в руку карандаш

Ваша очередь. Перед вами еще три объекта; что получится, если передать каждый из них функции `prequal`? Сначала попробуйте определить ответ в уме, а потом напишите код для проверки ответов:

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};

prequal(cadi);
```

____

Здесь запишите значение prequal.

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

____

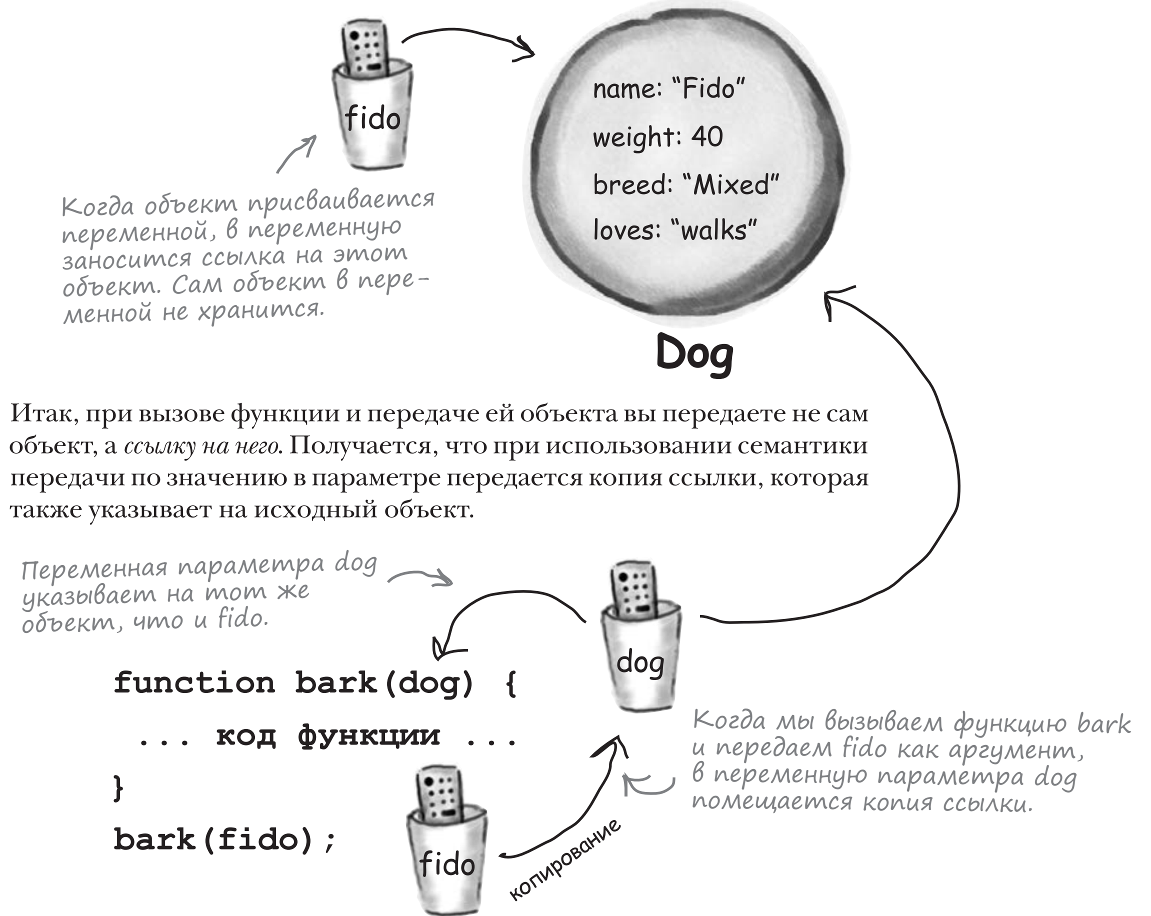
```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};

prequal(chevy);
```

____

Еще немного поговорим о передаче объектов функциям

Мы уже говорили о том, как аргументы передаются функциям — они *передаются по значению*, что подразумевает *копирование*. Итак, если мы передаем целое число, соответствующая функция получает копию этого целого числа и может ею распоряжаться. Аналогичные правила действуют и для объектов, однако необходимо подробнее разобраться, что передача по значению означает для объектов, — это поможет понять, что происходит при передаче объектов функциям.

Вы уже знаете, что при присваивании объекта переменной в переменной хранится *ссылка* на объект, а не сам объект. Еще раз напомним, что ссылку можно рассматривать как указатель на объект:

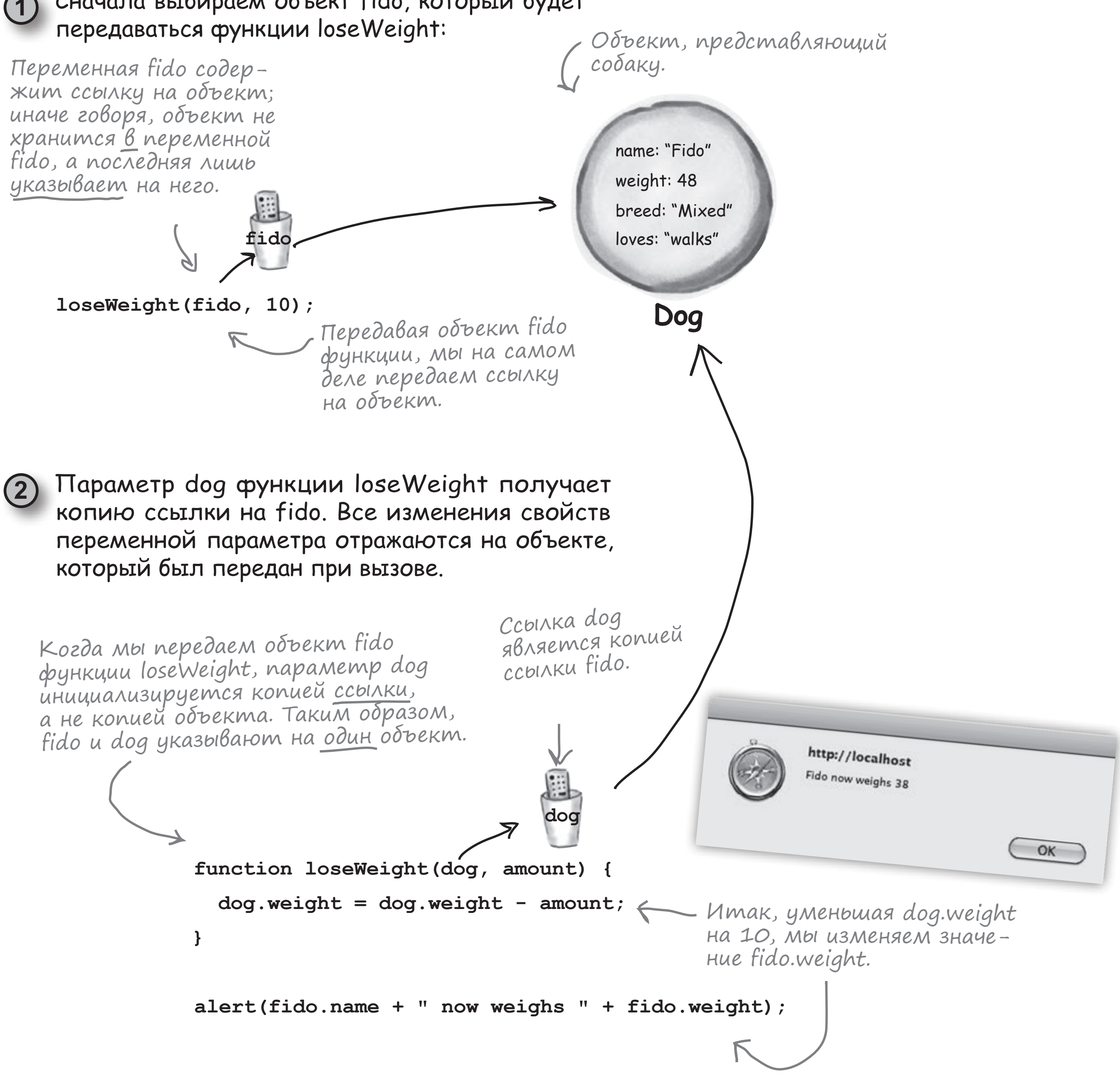


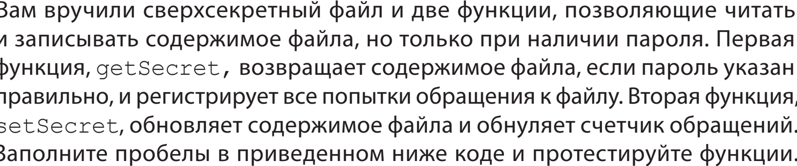
Итак, при вызове функции и передаче ей объекта вы передаете не сам объект, а *ссылку на него*. Получается, что при использовании семантики передачи по значению в параметре передается копия ссылки, которая также указывает на исходный объект.

Что же все это означает для нас? Одно из самых важных последствий заключается в том, что при изменении свойства объекта в функции изменяется свойство *исходного* объекта. Следовательно, все изменения, вносимые в объект внутри функции, продолжат действовать и после завершения функции. Рассмотрим пример...

Сажаем Фидо на диету...

Предположим, мы тестируем новый метод снижения веса для собак, который будет реализован в виде удобной функции `loseWeight`. От пользователя потребуется лишь передать `loseWeight` объект собаки и вес, который нужно сбросить, а собака, словно по волшебству, похудеет. Вот как это происходит:





Вам вручили сверхсекретный файл и две функции, позволяющие читать и записывать содержимое файла, но только при наличии пароля. Первая функция, `getSecret`, возвращает содержимое файла, если пароль указан правильно, и регистрирует все попытки обращения к файлу. Вторая функция, `setSecret`, обновляет содержимое файла и обнуляет счетчик обращений. Заполните пробелы в приведенном ниже коде и протестируйте функции.

```
function getSecret(file, secretPassword) {
    _______.opened = _______.opened + 1;
    if (secretPassword == _______.password) {
        return ______.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == _______.password) {
        ______.opened = 0;
        ______.contents = secret;
    }
}

var superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
var secret = getSecret(_______________, _____);
console.log(secret);

setSecret(_________________, _____, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(_______________, _____);
console.log(secret);
```

Я вернулся, и у меня есть новая программа. Эта крошка будет штамповать новые машины с утра до вечера.

```
<!doctype html>
<html lang="en">
<head>
  <title>Object-o-matic</title>
  <meta charset="utf-8">
  <script>
    function makeCar() {
        var makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
        var models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
        var years = [1955, 1957, 1948, 1954, 1961];
        var colors = ["red", "blue", "tan", "yellow", "white"];
        var convertible = [true, false];

        var rand1 = Math.floor(Math.random() * makes.length);
        var rand2 = Math.floor(Math.random() * models.length);
        var rand3 = Math.floor(Math.random() * years.length);
        var rand4 = Math.floor(Math.random() * colors.length);
        var rand5 = Math.floor(Math.random() * 5) + 1;
        var rand6 = Math.floor(Math.random() * 2);

        var car = {
            make: makes[rand1],
            model: models[rand2],
            year: years[rand3],
            color: colors[rand4],
            passengers: rand5,
            convertible: convertible[rand6],
            mileage: 0
        };
        return car;
    }

    function displayCar(car) {
        console.log("Your new car is a " + car.year + " " + car.make + " " + car.model);
    }

    var carToSell = makeCar();
    displayCar(carToSell);

  </script>
</head>
<body></body>
</html>
```

Эта программа напоминает Генератор Красивых Фраз из главы 4, но вместо слов в ней используются свойства машин, а вместо маркетинговых фраз генерируются новые объекты car!

Выясните, что делает эта программа и как она работает.

Генератор Машин

Генератор Машин (написанный тем же автором, что и программа Генератор Красивых Фраз) с утра до вечера производит машины со случайным набором свойств. Иначе говоря, вместо маркетинговых лозунгов эта программа генерирует случайные комбинации марки, модели, года выпуска и всех остальных свойств объекта car. *В сущности, это ваш личный автомобильный завод, только воплощенный в программном коде.* Давайте поближе познакомимся с тем, как он работает.

1. Прежде всего у нас имеется функция `makeCar`, которая вызывается для создания новой машины. Программа определяет четыре массива с маркой, моделью, годом выпуска и цветом машины, а также массив с булевским признаком, который указывает, имеет ли машина откидной верх. Мы генерируем пять случайных чисел для выбора комбинации этих признаков из всех пяти массивов. Также генерируется еще одно случайное число, которое будет обозначать количество пассажиров.

В этих четырех массивах содержатся варианты марки, модели, года выпуска и цвета...

```
var makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
var models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
var years = [1955, 1957, 1948, 1954, 1961];
var colors = ["red", "blue", "tan", "yellow", "white"];
var convertible = [true, false];
```

...а этот массив будет использоваться для выбора признака откидного верха (true или false).

Четыре случайных числа, которые будут использоваться для создания случайных комбинаций элементов четырех массивов.

```
var rand1 = Math.floor(Math.random() * makes.length);
var rand2 = Math.floor(Math.random() * models.length);
var rand3 = Math.floor(Math.random() * years.length);
var rand4 = Math.floor(Math.random() * colors.length);
var rand5 = Math.floor(Math.random() * 5) + 1;
var rand6 = Math.floor(Math.random() * 2);
```

Это случайное число будет задавать количество пассажиров. Оно увеличивается на 1, чтобы машина вмещала хотя бы одного пассажира.

...а это случайное число определяет, имеет машина откидной верх или нет.

2. Вместо построения строки посредством конкатенации, как это было сделано в Генераторе Красивых Фраз, мы будем создавать новый объект, представляющий машину. Этот объект обладает всеми предполагаемыми свойствами. Значения марки, модели, года выпуска и цвета выбираются из массивов при помощи случайных чисел, созданных на шаге 1; к ним добавляются свойства, определяющие количество пассажиров, наличие откидного верха и пробег:

```
var car = {
    make: makes[rand1],
    model: models[rand2],
    year: years[rand3],
    color: colors[rand4],
    passengers: rand5,
    convertible: convertible[rand6],
    mileage: 0
};
```

Мы создаем новый объект car со значениями свойств, взятыми из массивов.

Количество пассажиров инициализируется случайным числом, а признак откидного верха — значением true или false из массива convertible.

Наконец, пробег инициализируется значением 0 (ведь это новая машина).

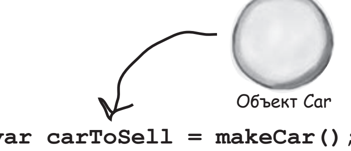
3 Последняя команда `makeCar` возвращает новый объект `car`:

```
return car;
```

Объекты возвращаются функцией точно так же, как любые другие значения. Рассмотрим код с вызовом `makeCar`:

```
function displayCar(car) {
  console.log("Your new car is a " + car.year + " " +
              car.make + " " + car.model);
}
var carToSell = makeCar();
displayCar(carToSell);
```

Не забывайте: возвращаемое значение (которое присваивается переменной carToSell) представляет собой ссылку на объект car.




```
var carToSell = makeCar();
```

Сначала мы вызываем функцию `makeCar` и присваиваем возвращаемое ей значение переменной `carToSell`. Затем объект, возвращенный `makeCar`, передается функции `displayCar`, которая просто выводит значения некоторых его свойств на консоль.

4 Загрузите Генератор Машин в браузере (autoomatic.html) и опробуйте программу в деле. Вы увидите, с какой легкостью она генерирует новые машины.

А вот и ваша новая машина!

Консоль JavaScript

```
Your new car is a 1957 Fiat Taxi
Your new car is a 1961 Tucker 500
Your new car is a 1948 GM Torpedo
```

Перезагрузите страницу несколько раз, как мы!

Ведите себя прилично! И объекты свои научите...

Вы же не думали, что возможности объектов ограничиваются хранением чисел и строк? Объекты *активны*. Они могут *выполнять операции*. Собака не сидит на одном месте... она лает, бегает, приносит палку; объект dog тоже должен все это делать! С машинами то же самое: мы их ведем, паркуем, заставляем тормозить и т. д. После всего, что вы узнали в этой главе, вы полностью готовы к добавлению поведения в объекты. Вот как это делается:

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    drive: function() {
       alert.log("Zoom zoom!");
    }
};
```

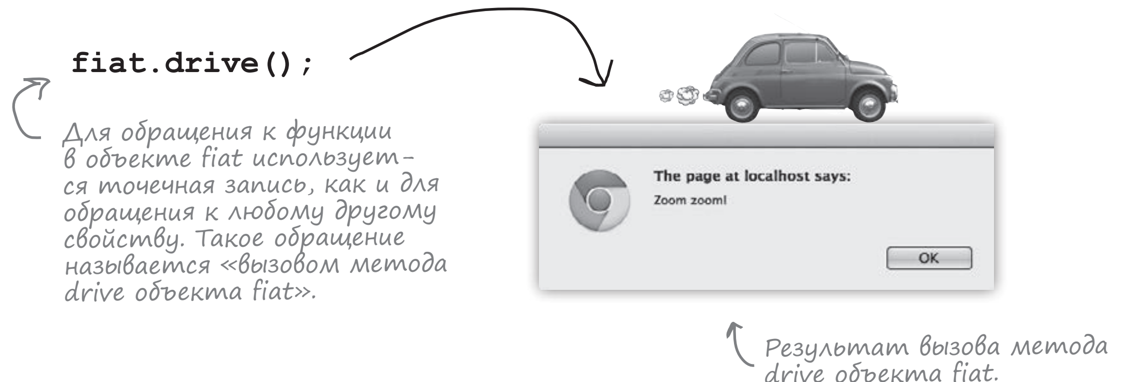
Функцию можно добавить прямо в объект — это делается так.

Просто присвойте определение функции свойству. Да, свойства тоже могут быть функциями!

Обратите внимание: мы не указываем имя функции, а просто ставим ключевое слово function, за которым следует тело. Имя функции совпадает с именем свойства.

О терминологии: функции, определяемые в объектах, обычно называются методами. Это стандартный объектно-ориентированный термин для обозначения функций, принадлежащих объектам.

При вызове функции `drive` (ах, простите — *метода* drive) тоже используется точечная запись, но на этот раз с именем объекта `fiat` и именем свойства `drive`, только за именем свойства следуют круглые скобки (как при вызове любой другой функции).



Усовершенствование метода drive

Давайте сделаем fiat чуть более похожим на типичный автомобиль. Ведь машину обычно можно вести только после того, как будет запущен ее двигатель, не так ли? Так почему бы не смоделировать это поведение? Для этого нам понадобится:

- ❒ Булевское свойство для хранения состояния двигателя (запущен или нет).
- ❒ Пара методов для запуска и остановки двигателя.
- ❒ Условная проверка в методе drive, которая удостоверяется, что двигатель запущен, прежде чем вы сможете вести машину.

Мы начнем с добавления булевского свойства `started` и методов запуска и остановки двигателя, а потом обновим метод `drive`, чтобы в нем использовалось свойство `started`.

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,

    start: function() {
       started = true;
    },

    stop: function() {
        started = false;
    },

    drive: function() {
        if (started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Свойство для хранения текущего состояния двигателя (true, если двигатель запущен; false, если остановлен).

Метод для запуска двигателя. Пока этот метод делает совсем немного — он задает свойству started значение true.

Метод для остановки двигателя. Пока он просто задает свойству started значение false.

А здесь начинается самое интересное: если двигатель запущен, то при вызове drive выводится сообщение "Zoom zoom!", а если нет — предупреждение о том, что сначала нужно запустить двигатель.

Интересно: вместо того, чтобы просто изменить свойство `started` напрямую, мы зачем-то пишем для него метод. Зачем? Разве простое изменение свойства не уменьшает объем кода?

Верно. Вы правы; чтобы запустить двигатель, мы могли бы заменить код:

```
fiat.start();
```

простым вызовом:

```
fiat.started = true;
```

Это избавило бы нас от необходимости писать метод для запуска двигателя.

Тогда почему мы создаем и вызываем метод `start` вместо того, чтобы просто изменить свойство `started` напрямую? Использование метода для изменения свойства — еще один пример инкапсуляции; часто мы можем упростить сопровождение и расширение кода, поручая технические подробности выполнения операций объекту. Лучше создать метод `start`, который знает, как запускать машину, вместо того, чтобы помнить: «чтобы запустить машину, нужно взять переменную `started` и присвоить ей true».

Возможно, вы скажете: «И что с того? Почему нельзя просто задать свойству true?!» Рассмотрим более сложный метод `start`, который проверяет ремни, уровень бензина, состояние аккумулятора, температуру двигателя и т. д., прежде чем задать `started` значение true. Конечно, вам не захочется помнить все эти действия каждый раз, когда вы запускаете машину. Гораздо удобнее иметь простой метод, который будет делать все за вас. Выделяя все эти технические подробности в специальный метод, мы получаем возможность вызвать метод для выполнения некоторой операции; при этом все подробности выполнения операции берет на себя сам метод.

Машина отправляется на тест-драйв

Отправим наш новый, усовершенствованный объект `fiat` на тест-драйв. Тестирование должно быть проведено тщательно — сначала мы попробуем вести машину с незапущенным двигателем, потом запустим двигатель, немного проедем и остановимся. Проследите за тем, чтобы в простой странице HTML (carWithDrive.html) был введен код объекта `fiat`, включая новые методы `start`, `stop` и `drive`. Затем добавьте приведенный ниже код после определения объекта:

```
fiat.drive();
fiat.start();
fiat.drive();
fiat.stop();
```

Сначала мы попытаемся вести машину с неработающим двигателем; программа должна вывести сообщение с напоминанием. Затем двигатель будет запущен, и мы поведем машину. Наконец, когда пробная поездка будет закончена, машина остановится.

Загрузите страницу в браузере — и в путь!

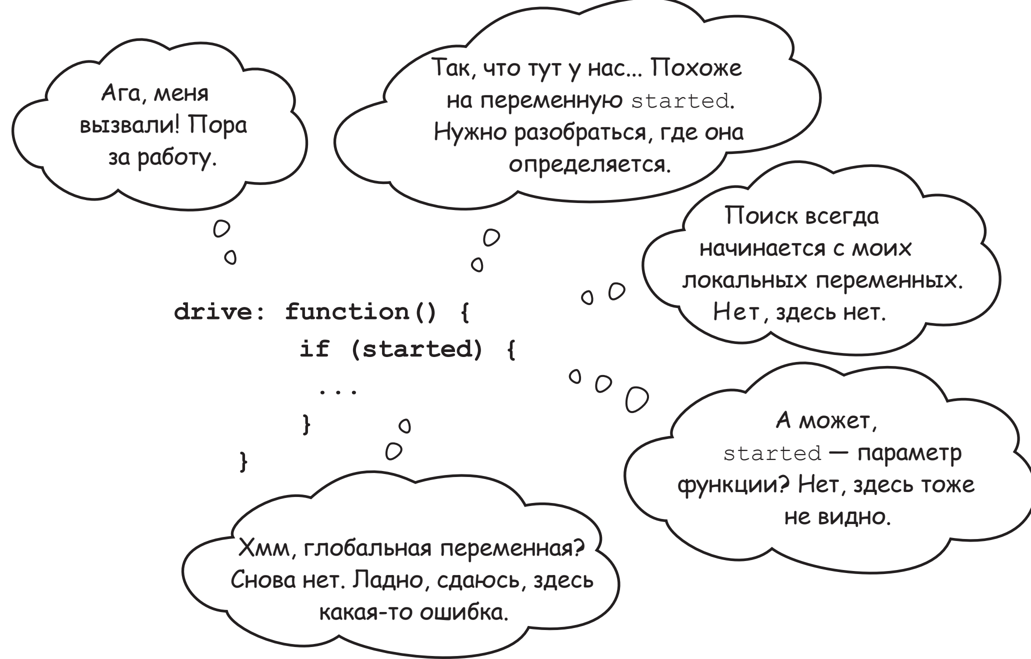
Стоп, не так быстро...

Что, не едет? Не только у вас. Если вы откроете консоль JavaScript, то, скорее всего, на ней будет выведено сообщение о том, что метод `started` не определен.

Консоль JavaScript

```
ReferenceError: started is not defined
```

Что же происходит? Давайте послушаем, что говорит метод `drive`, и разберемся, что происходит при попытке повести машину вызовом `fiat.drive()`:



Почему метод drive не знает о свойстве started?

Почему возникла эта странная ситуация? В методах объекта `fiat` содержатся ссылки на свойство `started`, а обычно при разрешении переменной в функции эта переменная оказывается локальной, глобальной или параметром функции. Но в методе `drive` переменная `started` не относится ни к одной из этих категорий; она является свойством объекта `fiat`.

Разве этот код не должен просто работать? Другими словами, в объекте `fiat` встречается переменная `started`; разве интерпретатор JavaScript не должен сообразить, что мы имеем в виду свойство `started`?

Нет. Как видите, не должен. Почему?

Дело вот в чем: то, что выглядит в методе как переменная, в действительности является свойством объекта, но мы не сообщаем JavaScript, какого именно объекта. Возможно, вы говорите себе: «Разумеется, я имею в виду ЭТОТ объект... в смысле текущий! Разве не очевидно?» Вообще-то не очевидно. В JavaScript существует ключевое слово `this`, которое предназначено именно для этой цели: оно обозначает *текущий объект, с которым мы работаем.*

А как я узнаю, какому объекту принадлежит `started`? Тебе придется сообщить мне.

```
drive: function() {
        if (started) {
         ...
        }
    }
```

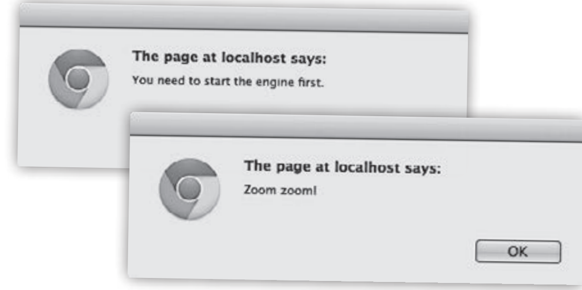
Стоит добавить ключевое слово `this`, как наш код заработает:

```
var fiat = {
    make: "Fiat",
    // Здесь перечисляются другие свойства (экономим место)
    started: false,

    start: function() {
        this.started = true;
    },

    stop: function() {
        this.started = false;
    },

    drive: function() {
        if (this.started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Используйте this в точечной записи перед каждым обращением к свойству started. Тем самым вы сообщаете интерпретатору JavaScript, что имеете в виду свойство ЭТОГО конкретного объекта (чтобы интерпретатор JavaScript не думал, что вы ссылаетесь на переменную).

Тест-драйв для this

Обновите код и проверьте его в деле.
Вот что получилось у нас:



СТАНЬ браузером

Ниже приведен код JavaScript, содержащий ошибки. Представьте себя на месте браузера и попробуйте найти ошибки в коде. Выполнив упражнение, сверьтесь с ответами в конце главы и посмотрите, удалось ли вам найти все ошибки до единой.

Не стесняйтесь, делайте пометки прямо здесь...

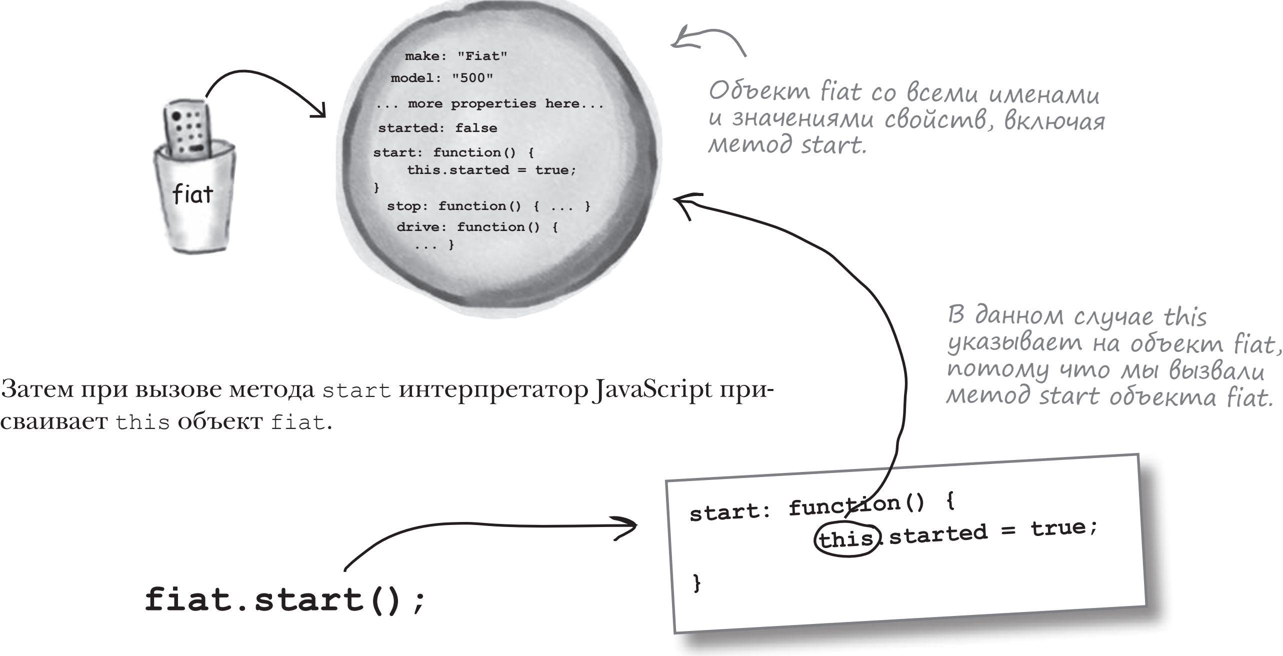
```
var song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,

    play: function() {
        if (!playing) {
            this = true;
            console.log("Playing "
                + name + " by " + artist);
        }
    },

    pause: function() {
        if (playing) {
            this.playing = false;
        }
    }
};

this.play();
this.pause();
```

Как работает this

Представьте себе `this` как переменную, которой присваивается объект, чей метод был только что вызван. Иначе говоря, если вызвать метод `start` объекта fiat в записи `fiat.start()` и использовать `this` в теле метода `start`, это ключевое слово будет ссылаться на объект `fiat`. Давайте повнимательнее разберемся, что происходит при вызове метода `start` объекта `fiat`.

Имеется объект, представляющий машину «Фиат», который присвоен переменной `fiat`:

Затем при вызове метода `start` интерпретатор JavaScript присваивает `this` объект `fiat`.



Чтобы по-настоящему понять смысл `this`, необходимо осознать один важный момент: при вызове любого метода вы можете рассчитывать на то, что `this` в теле этого метода будет указывать на *объект, метод которого был вызван*. Пожалуй, сказанное станет немного яснее, если мы рассмотрим примеры использования `this` с другими объектами...

При вызове метода `start` объекта `chevy` ключевое слово `this` в теле метода будет указывать на объект `chevy`.

```
start: function() {
        this.started = true;
}
```

```
chevy.start();
```

```
make: "Chevy"
model: "Bel Air"
... more properties here...
started: false
start: function() {
 this.started = true;
}
 stop: function() { ... }
  drive: function() {
   ... }
```

В методе `start` объекта `taxi` ключевое слово `this` указывает на объект `taxi`.

```
taxi.start();
```

```
start: function() {
        this.started = true;
}
```

```
make: "Webville..."
model: "Taxi"
... more properties here ...
started: false
start: function() {
  this.started = true;
}
  stop: function() { ... }
   drive: function() {...}
```

Возьми в руку карандаш

Используйте свои новые навыки обращения с **this** и помогите нам закончить этот код. Сверьтесь с ответами в конце главы.

```
var eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                           this.index = ________.index + 1;
                           if (_________.index  >= _________.advice.length) {
                              _________.index = 0;
                           }
                 },
                 look: function() {
                           return _________.advice[_________.index];
                 }
};
eightBall.shake();
console.log(eightBall.look());
```

Повторите несколько раз, чтобы протестировать свой код.

Консоль JavaScript

```
no
maybe
not a chance
```

Часто Задаваемые Вопросы

В: Чем метод отличается от функции?

О: Метод — это функция, присвоенная имени свойства в объекте. Функции вызываются по именам, а методы — точечной записью, включающей имя объекта и имя свойства. Ключевое слово `this` в теле метода указывает на объект, для которого этот метод был вызван.

В: Я заметил, что при использовании ключевого слова function в объекте мы не указываем имя функции. Что с ним случилось?

О: Верно, для вызова методов мы указываем имя свойства (вместо того, чтобы явно назначить имя функции и обращаться к ней по этому имени). Пока будем считать это правилом, но позднее будет рассмотрена тема анонимных функций (то есть функций, вызываемых без явного указания имен).

В: Могут ли методы содержать локальные переменные, как и функции?

О: Да. Метод является функцией. Мы называем эту функцию методом только потому, что она находится внутри объекта. Метод делает все, что делает функция, именно потому, что он и есть функция.

В: Значит, методы могут возвращать значения?

О: Да. Перечитайте предыдущий ответ!

В: Как насчет передачи аргументов методам? Это тоже возможно?

О: Эээ... Вы так и не прочитали тот ответ? Да, возможно!

В: Можно ли добавить метод в объект после того, как он создан (по аналогии с добавлением свойств)?

О: Да. Метод можно рассматривать как функцию, присвоенную свойству, поэтому новый метод можно добавить в любой момент:

```
// Добавление метода engageTurbo
car.engageTurbo =
        function() { ... };
```

В: Если я добавлю метод engageTurbo, ключевое слово `this` по-прежнему будет работать?

О: Да. Вспомните: указатель на объект, метод которого был вызван, присваивается в момент вызова.

В: Когда `this` присваивается ссылка на объект? В момент определения объекта или при вызове метода?

О: Значение `this` присваивается при вызове метода. Таким образом, при вызове fiat.start() `this` присваивается ссылка на fiat, а при вызове chevy.start() присваивается ссылка на chevy. *Кажется*, будто значение `this` задается при определении объекта, потому что в fiat.start `this` всегда присваивается fiat, а в chevy.start всегда присваивается chevy. Но как будет показано позднее, существует веская причина для задания значения `this` при вызове, а не при определении объекта. Это очень важный момент, к которому мы еще неоднократно вернемся.

Допустим, вы скопировали методы start, stop и drive в созданные ранее объекты chevy и cadi. Что необходимо изменить в коде, чтобы эти методы правильно работали?

Ответ: Ничего! this обозначает «текущий объект», то есть объект, метод которого мы вызываем.

Пора уже привести весь автопарк в движение. Добавьте метод drive в каждый объект. Когда это будет сделано, добавьте код методов запуска двигателя, ведения машины и остановки. Сверьтесь с ответами в конце главы.

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};
```

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};
```

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

Не забудьте добавить запятую после mileage при добавлении новых свойств!

Добавьте в каждый объект свойство started и методы. Затем используйте приведенный внизу код для проверки.

```
started: false,

start: function() {
   this.started = true;
},

stop: function() {
   this.started = false;
},

drive: function() {
   if (this.started) {
     alert(this.make + " " +
           this.model + " goes zoom zoom!");
   } else {
     alert("You need to start the engine first.");
   }
}
```

Метод drive немного изменен; будьте внимательны и используйте новую версию кода.

Включите этот код после определений объектов машин, чтобы проверить их в деле.

```
cadi.start();
cadi.drive();
cadi.stop();
chevy.start();
chevy.drive();
chevy.stop();
taxi.start();
taxi.drive();
taxi.stop();
```

Похоже, копирование и вставка методов в разные объекты машин приводит к дублированию кода. Нет ли более эффективного способа?

А вы наблюдательны.

Да, копирование методов `start`, `stop` и `drive` в каждый объект машины определенно приводит у дублированию кода. В отличие от других свойств, которые могут зависеть от конкретного объекта, эти методы остаются неизменными для всех объектов.

Что вы говорите? «Отлично, повторное использование кода в действии»? Нет, не торопитесь. Конечно, код используется повторно, но для этого он копируется, и не однократно, а много раз! Что произойдет, если мы вдруг решим, что на работу нужно ехать по особым правилам? Код придется перерабатывать в каждом объекте машины. Нехорошо. Исправления во многих местах не только неэффективны, но и создают риск ошибок.

Однако вы выявили проблему более серьезную, чем простое копирование: предполагается, что присутствие одинаковых свойств во всех наших объектах делает их всех частными случаями объекта машины. Что если свойство mileage будет случайно опущено в одном из объектов — можно ли будет считать его машиной?

Все эти проблемы вполне реальны. Тем не менее мы займемся ими в следующей главе, посвященной нетривиальному использованию объектов, когда речь пойдет о правильной организации повторного использования кода объектов.

Да, вы можете перебрать все свойства объекта в цикле. Для этого стоит воспользоваться разновидностью циклов, которая нам еще не встречалась, — циклом `for in`. Цикл `for in` перебирает все свойства объекта в произвольном порядке. Перебор свойств объекта `chevy` может происходить так:

Цикл for in перебирает свойства объекта, которые последовательно присваиваются переменной prop.

```
for (var prop in chevy) {
    console.log(prop + ": " + chevy[prop]);
}
```

Переменная prop может использоваться для получения значений свойств (с квадратными скобками).

JavaScript console

```
make: Chevy
model: Bel Air
year: 1957
color: red
passengers: 2
convertible: false
mileage: 1021
```

И тут возникает другая тема: существует другой способ обращения к свойствам. Вы заметили альтернативный синтаксис обращения к свойствам `chevy`? Оказывается, в JavaScript есть два способа обращения к свойствам объектов. Точечная запись вам уже известна:

```
chevy.color
```

Имя объекта, за которым следует точка и имя свойства.

Но существует и другой способ: запись с квадратными скобками:

```
chevy["color"]
```

За именем объекта ставятся квадратные скобки, в которые заключается имя свойства в кавычках. *Немного похоже на обращение к элементам массивов.*

Эти две формы эквивалентны и делают одно и то же. Единственное, чем отличается запись с квадратными скобками — она обладает чуть большей гибкостью и может использоваться в следующей форме:

```
chevy["co" + "lor"]
```

В квадратных скобках может находиться любое выражение — необходимо лишь, чтобы при его вычислении получалось имя свойства, представленное строкой.

Как поведение влияет на состояние

Объекты обладают состоянием и поведением. Свойства объекта позволяют хранить данные текущего состояния — уровень бензина, текущую температуру, или песню, которая сейчас звучит в салоне. Методы объекта могут наделять объекты поведением — запустить машину, включить обогрев или выключить быструю перемотку музыкального трека. Вы заметили, что состояние и поведение связаны? Мы не сможем завести машину, если в баке нет бензина, а уровень топлива уменьшается во время движения. Как в реальном мире, верно?

Давайте еще немного поэкспериментируем с этой концепцией: добавим в автомобиль датчик бензина, а затем соответствующее поведение. Уровень топлива будет представлен новым свойством `fuel`, а заправку будет выполнять новый метод `addFuel`. У метода `addFuel` имеется параметр `amount`, который будет использоваться как приращение уровня топлива в свойстве `fuel`. Итак, добавьте следующие свойства в объект `fiat`:

```
var fiat = {
    make: "Fiat",
    model: "500",
    // Здесь идут другие свойства...
    started: false,
    fuel: 0,

    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
       if (this.started) {
         alert(this.make + " " + this.model + " goes zoom zoom!");
       } else {
         alert("You need to start the engine first.");
       }
    },
    addFuel: function(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

Мы добавили новое свойство fuel, представляющее количество бензина в машине. В начале существования объекта car бензобак пуст.

Также добавим метод addFuel для заправки машины. Количество добавляемого топлива задается параметром amount при вызове метода.

А amount — параметр функции, поэтому использовать this не нужно.

Помните: fuel — свойство объекта, поэтому необходимо использовать ключевое слово this...

Состояние влияет на поведение

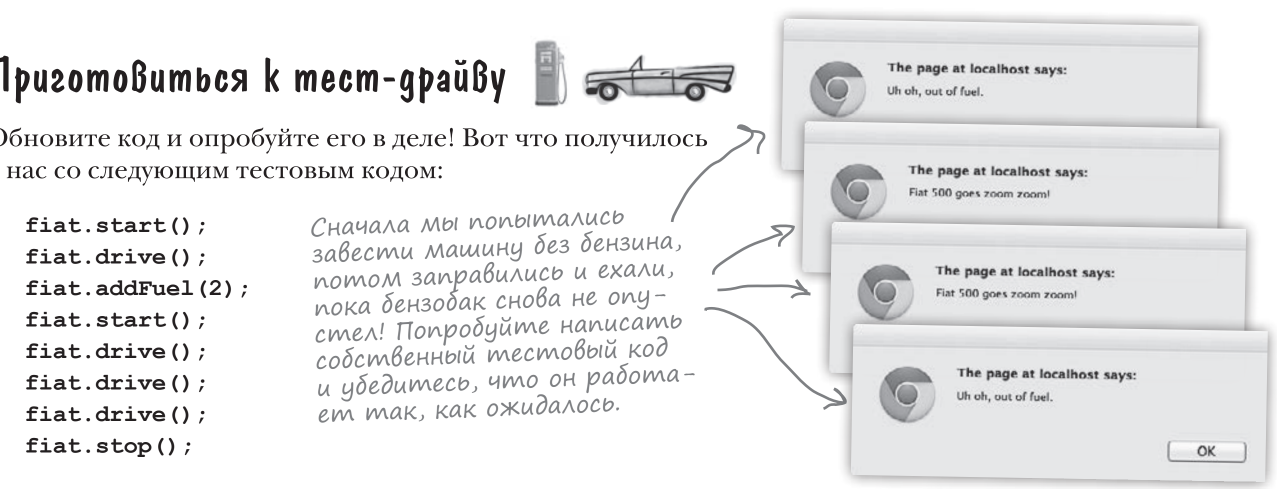
Теперь, после появления датчика топлива, можно переходить к реализации интересного поведения. Например, если топлива нет вообще, машина ехать не должна! Итак, немного изменим метод `drive` и включим в него проверку уровня топлива, а потом будем уменьшать fuel на 1 при каждом вызове drive. Код выглядит так:

```
var fiat = {
    // Другие свойства и методы...
    drive: function() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
               this.fuel = this.fuel - 1;
            } else {
                alert("Uh oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel: function(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

Прежде чем выехать, мы убеждаемся, что в машине есть топливо. И если машина едет, количество бензина должно уменьшаться при каждом вызове.

Если бензина не осталось, мы выводим сообщение и останавливаем двигатель. Чтобы снова завести машину, необходимо добавить бензина и снова запустить двигатель.

Приготовиться к тест-драйву



Обновите код и опробуйте его в деле! Вот что получилось у нас со следующим тестовым кодом:

```
fiat.start();
fiat.drive();
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.drive();
fiat.drive();
fiat.stop();
```

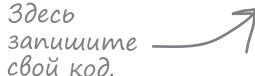
Сначала мы попытались завести машину без бензина, потом заправились и ехали, пока бензобак снова не опустел! Попробуйте написать собственный тестовый код и убедитесь, что он работает так, как ожидалось.

Работа по интеграции свойства fuel в объект машины еще не завершена. Например, можно ли запустить двигатель при отсутствии бензина? Проверим метод start:

```
start: function() {
    this.started = true;
}
```

Конечно, можно — ничто не помешает.

Помогите нам интегрировать свойство fuel в код. Для этого проверьте состояние fuel перед запуском двигателя. Если метод start вызывается при пустом бензобаке, выведите какое-нибудь осмысленное сообщение для водителя, например: **"The car is on empty, fill up before starting!"** Перепишите метод start, добавьте его в свой код и протестируйте. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.



Еще раз взгляните на код объекта fiat. В каких еще местах свойство fuel может влиять на поведение машины (или, наоборот, стоит добавить поведение, изменяющее свойство fuel)? Запишите свои предложения здесь.

Поздравляем с первыми объектами!

Первая глава, посвященная объектам, подходит к концу, а вы готовы двигаться дальше. Помните, как мы начинали работать с JavaScript? Мы мыслили понятиями низкоуровневых чисел, строк, команд, условий, циклов for и т. д. А теперь взгляните, как далеко мы зашли. Вы начинаете мыслить на более высоком уровне — понятиями методов и объектов. Только посмотрите:

```
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.stop();
```

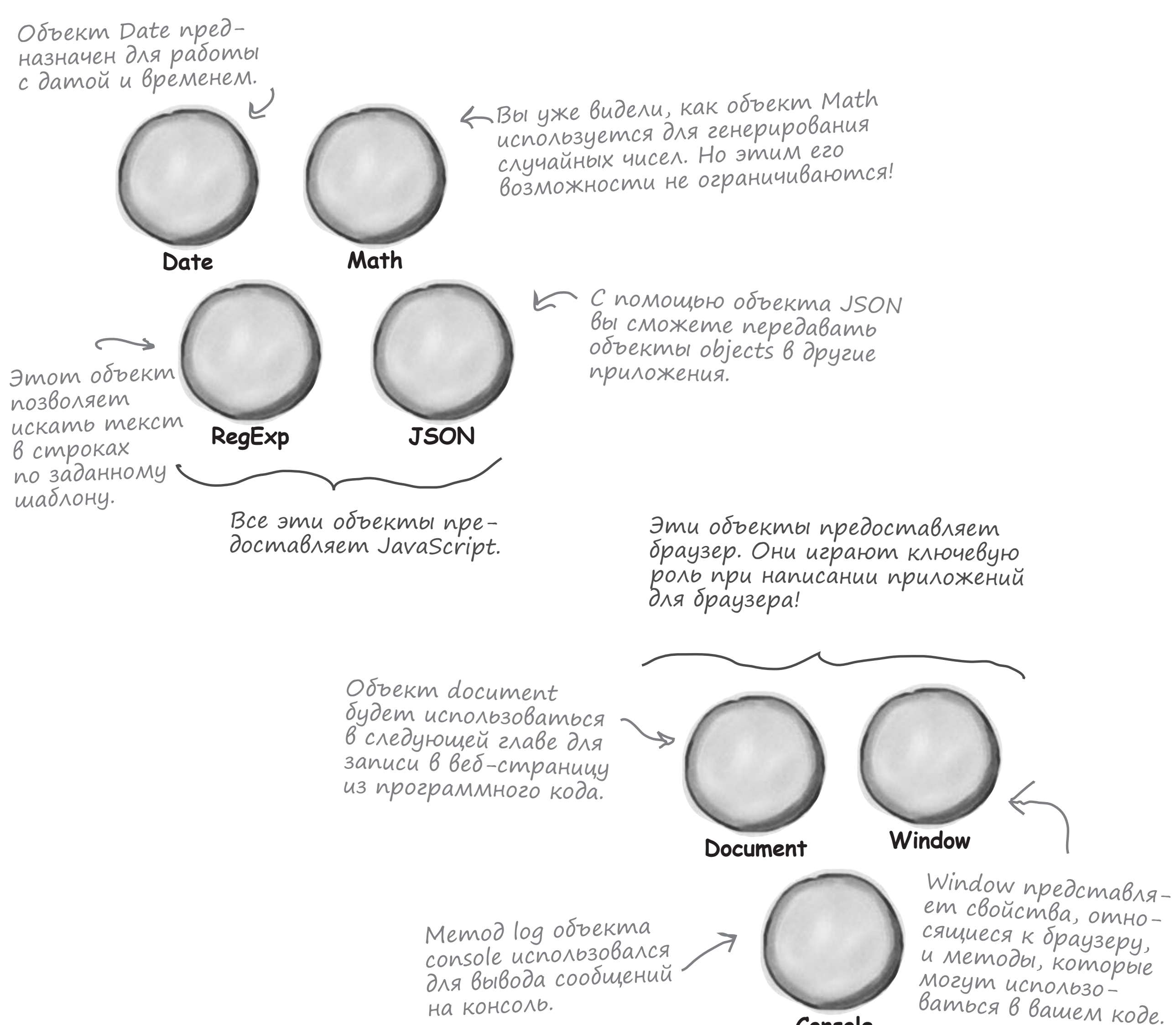
Насколько проще понять, что происходит в этом коде — ведь он описывает мир при помощи объектов, обладающих состоянием и поведением.

Впрочем, это только начало! Мы можем пойти еще дальше и непременно это сделаем. Теперь, когда вы так много узнали об объектах, мы поднимем ваши навыки на новый уровень и начнем писать полноценный объектно-ориентированный код, используя другие возможности JavaScript и многие передовые методы разработки (чрезвычайно важные при работе с объектами).

Но есть еще кое-что, что вы должны знать до того, как мы перейдем к следующей главе...

Представьте, вас окружают сплошные объекты! (и они упрощают вашу работу)

Знакомство с объектами открывает перед вами совершенно новые горизонты, потому что JavaScript предоставляет множество готовых объектов, которые вы можете использовать в своем коде (для математических расчетов, операций со строками, датой и временем... перечислять можно долго). JavaScript также предоставляет объекты, необходимые при написании кода для браузера (мы рассмотрим один из таких объектов в следующей главе). Сейчас же мы в общих чертах представим объекты, к которым еще не раз вернемся к книге.



Откровенно об объекте

Интервью недели:
Объект рассказывает о себе

Head First: Добро пожаловать, Объект, это была потрясающе интересная глава. Переход на объектные представления — это же полный переворот в сознании.

Объект: Что вы, все только начинается.

Head First: Как это?

Объект: Объект — это набор свойств, верно? Одни свойства используются для хранения состояния объекта, а другие представляют собой функции (а вернее, методы), определяющие поведение объекта.

Head First: Да, я понимаю. Вообще-то мы не думали о методах как о свойствах, но в конце концов, метод — всего лишь пара «имя/значение»... если, конечно, можно назвать функцию значением.

Объект: Еще как можно! Более того, эта мысль — настоящее озарение, сознаете вы это или нет. Запомните ее.

Head First: Мы немного отклонились от темы...

Объект: Итак, мы рассматривали объекты с их наборами свойств и создавали множество разных объектов — как, например, множество разных типов машин.

Head First: Верно.

Объект: Но все эти объекты были слишком привязаны к конкретному применению. Чтобы возможности объектов проявились в полной мере, необходимо создать некий шаблон, на основе которого «штампуются» похожие объекты.

Head First: Вы имеете в виду объекты, относящиеся к одному типу?

Объект: В определенном смысле. Вы увидите, что в JavaScript концепция типа весьма любопытна. Однако вы на верном пути. Вы увидите, какие возможности открываются при написании кода для работы с объектами одного типа. Допустим, вы можете написать код, который работает для любого вида транспорта, будь то велосипед, машина или автобус. В этом вся соль.

Head First: Безусловно, это выглядит интересно. Что еще для этого необходимо?

Объект: Немножко лучше разбираться в объектах и иметь механизм создания похожих объектов.

Head First: Ведь мы это уже делали? Все эти машины?

Объект: Они похожи только потому, что написанный нами код создает их похожими. Другими словами, все эти объекты обладают одинаковыми свойствами и методами.

Head First: Точно. И мы, кстати, говорили о дублировании кода в объектах, которое может иметь нежелательные последствия для сопровождения.

Объект: Следующий шаг — научиться создавать объекты, которые будут похожими и будут использовать код, находящийся в одном месте. Это подводит к проектированию объектно-ориентированного кода. Теперь, после знакомства с основами, вы уже готовы к этой теме.

Head First: Безусловно, наши читатели рады это услышать!

Объект: Но здесь есть один важный момент, о котором необходимо знать.

Head First: Вот как?

Объект: Рядом с вами находится множество объектов, которые вы можете использовать в своем коде.

Head First: Неужели? Мы и не заметили, где?

Объект: Как насчет вызова `console.log`? Что такое, по-вашему, `console`?

Head First: Раз уж мы говорим об объектах, наверное, это объект?

Объект: В ТОЧКУ. А `log`?

Head First: Свойство... эээ, метод?

Объект: Снова В ТОЧКУ. А как насчет `alert`?

Head First: Понятия не имею.

Объект: Это тоже связано с объектами, но отложим эту тему на будущее.

Head First: Что ж, вы рассказали нам много полезного об объектах. Надеемся, эта беседа еще будет продолжена.

Объект: Конечно, мы еще встретимся.

Head First: Прекрасно! Тогда до следующего раза.

Взлом кода

В ходе очередной операции по захвату мира доктор Зло случайно открыл доступ к внутренней веб-странице, содержащей текущий пароль к его базе данных. С этим паролем мы наконец-то сможем победить его. Как только доктор Зло обнаружил, что страница доступна в Интернете, он быстро стер ее. К счастью, наши агенты успели сделать копию. Единственная проблема — агенты не знают ни HTML, ни JavaScript. Сможете ли вы подобрать пароль по приведенному коду? Учтите, что ошибка дорого обойдется Всему Прогрессивному Человечеству.

СОВЕРШЕННО СЕКРЕТНО

```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Это код JavaScript.

Похоже, в коде используется объект document.

Какой пароль будет выведен при вызове alert?

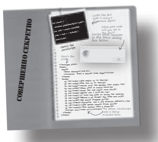
Запишите ответ в диалоговом окне alert.

OK

А это разметка HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

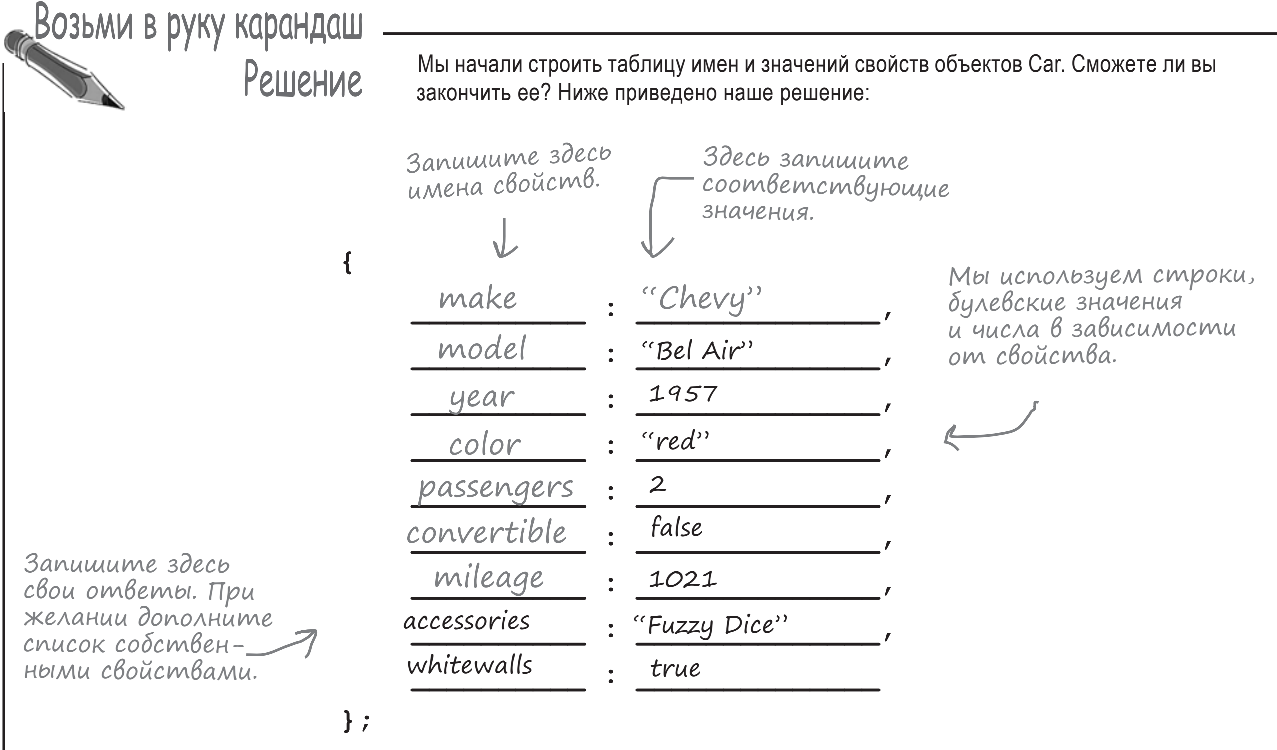
Включение приведенного выше кода JavaScript.



Если вы пропустили предыдущую страницу, вернитесь назад и пройдите испытание. Это будет жизненно важно в главе 6!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Объект представляет собой **набор свойств**.
- Для обращения к свойствам обычно используется **точечная запись**: имя переменной, содержащей объект, затем точка и имя свойства.
- Свойства можно добавить в объект в любой момент. Для этого следует присвоить значение с указанием имени нового свойства.
- Свойства можно удалять из объектов оператором **delete**.
- В отличие от переменных, содержащих примитивные значения (строки, числа и булевские величины), переменная не может содержать объект. Вместо этого в ней хранится **ссылка** на объект. Такие переменные называются ссылочными.
- При передаче объекта функция получает копию ссылки на объект, а не копию объекта. Таким образом, изменение свойств объекта в функции приводит к изменениям в исходном объекте.
- Свойства объекта могут содержать функции. Функция, определенная в объекте, называется методом.
- Методы можно вызывать в **точечной записи**: имя объекта, точка и имя свойства метода, за которым следует пара круглых скобок.
- Метод во всем аналогичен функции, кроме того, что он существует в объекте.
- Методы могут получать аргументы (как и обычные функции).
- При вызове метода объекта ключевое слово **this** указывает на объект, метод которого вызывается.
- Чтобы обратиться к свойствам объекта в методе, необходимо использовать точечную запись с указанием **this** вместо имени объекта.
- В объектно-ориентированном программировании разработчик мыслит понятиями объектов, а не процедур.
- Объект обладает **состоянием** и **поведением**. Состояние может влиять на поведение, а поведение может влиять на состояние.
- Объекты **инкапсулируют**, или скрывают от пользователя, сложность своего состояния и поведения.
- Хорошо спроектированный объект содержит методы, абстрагирующие подробности выполнения операций с объектом, так что вам не приходится беспокоиться о них.
- Наряду с объектами, которые вы создаете, JavaScript содержит много встроенных объектов, которые вы можете использовать в программах. Многие из этих встроенных объектов используются в книге.



Возьми в руку карандаш

Решение

Используйте свои новые навыки обращения с **this** и помогите нам закончить этот код. Ниже приведено наше решение.

```
var eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                            this.index = this.index + 1;
                            if (this.index  >= this.advice.length) {
                               this.index = 0;
                            }
                 },
                 look: function() {
                            return this.advice[this.index];
                 }

};
eightBall.shake();
console.log(eightBall.look());
```

Повторите несколько раз, чтобы протестировать код.

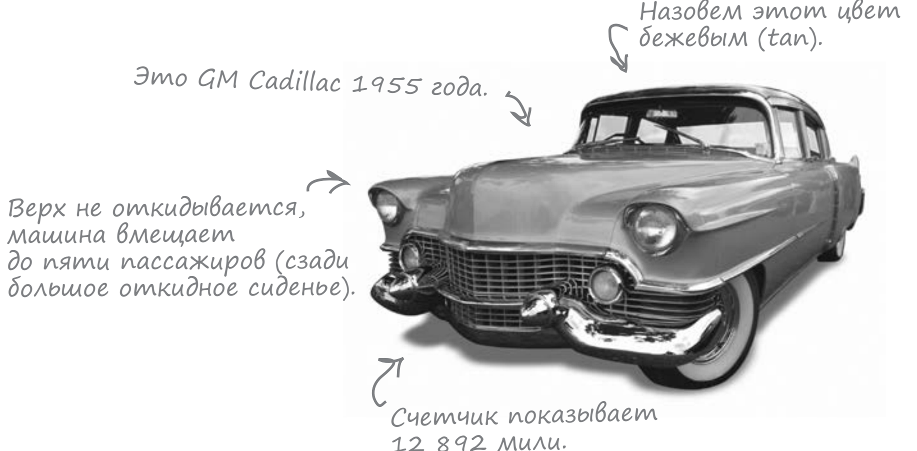
Консоль JavaScript

```
no
maybe
not a chance
```

Никто не заставляет вас ограничиваться всего одним объектом. Настоящая сила объектов (как вы вскоре увидите) проявляется в ситуациях, в которых вы создаете множество объектов и пишете код, способный работать с любым полученным объектом. Попробуйте создать с нуля новый объект — представляющий еще одну машину. Итак, напишите код для создания второго объекта. Ниже приведено наше решение.

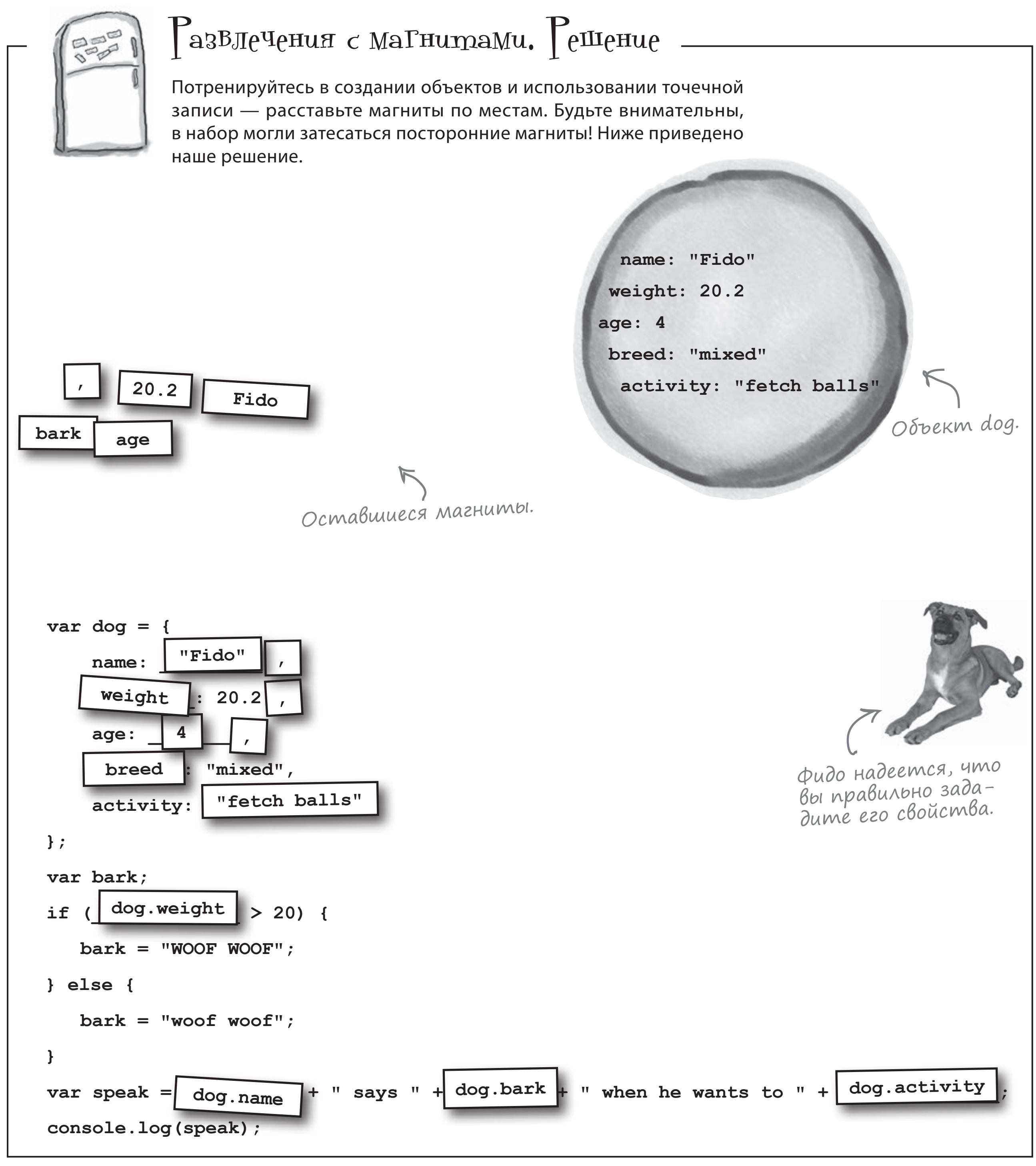
```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};
```

Разместите здесь свойства объекта Cadillac.



Развлечения с магнитами. Решение

Потренируйтесь в создании объектов и использовании точечной записи — расставьте магниты по местам. Будьте внимательны, в набор могли затесаться посторонние магниты! Ниже приведено наше решение.



Возьми в руку карандаш Решение

Ваша очередь. Перед вами еще три объекта; что получится, если передать каждый из них функции `prequal`? Сначала попробуйте определить ответ в уме, а потом напишите код для проверки ответов. Ниже приведено наше решение:

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892
};

prequal(cadi);
```

false

Здесь запишите значение prequal.

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

false

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021
};

prequal(chevy);
```

true

Возьми в руку карандаш
Решение

Вам вручили сверхсекретный файл и две функции, позволяющие читать и записывать содержимое файла, но только при наличии пароля. Первая функция, `getSecret`, возвращает содержимое файла, если пароль указан правильно, и регистрирует все попытки обращения к файлу. Вторая функция, `setSecret`, обновляет содержимое файла и обнуляет счетчик обращений. Заполните пробелы в приведенном ниже коде и протестируйте функции. Ниже приведено наше решение.

```
function getSecret(file, secretPassword) {
    file.opened = file.opened + 1;
    if (secretPassword == file.password) {
        return file.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == file.password) {
        file.opened = 0;
        file.contents = secret;
    }
}

var superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
var secret = getSecret(superSecretFile, 2);
console.log(secret);

setSecret(superSecretFile, 2, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(superSecretFile, 2);
console.log(secret);
```

Объект superSecretFile передается функции getSecret, и ему назначается имя параметра file. Нужно проследить за тем, чтобы при обращении к свойствам объекта (таким, как opened и password) использовалась запись с именем объекта file и точкой.

То же самое и здесь.

Объект supserSecretFile может передаваться функциям getSecret и setSecret.

Стань браузером. Решение

Ниже приведен код JavaScript, содержащий ошибки. Представьте себя на месте браузера и попробуйте найти ошибки в коде. Далее наше решение.

```
var song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,

    play: function() {
        if (!this.playing) {
            this.playing = true;
            console.log("Playing "
                + this.name + " by " + this.artist);
        }
    },

    pause: function() {
        if (this.playing) {
            this.playing = false;
        }
    }
};

this song.play();
this song.pause();
```

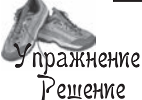
Здесь не хватает this.

А здесь — имени свойства playing.

Ключевое слово this также должно использоваться при обращении к обоим свойствам.

И здесь для обращения к свойству playing должно использоваться ключевое слово this.

Ключевое слово this не используется за пределами метода; чтобы вызвать метод для конкретного объекта, мы указываем имя переменной этого объекта.



Пора уже привести весь автопарк в движение. Добавьте метод drive в каждый объект. Когда это будет сделано, добавьте код методов запуска двигателя, ведения машины и остановки. Ниже приведено наше решение.

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                   this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                   this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                   this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};

cadi.start();
cadi.drive();
cadi.stop();

chevy.start();
chevy.drive();
chevy.stop();

taxi.start();
taxi.drive();
taxi.stop();
```

Не забудьте поставить запятую после каждого из добавляемых свойств.

Код скопирован в каждый объект, так что все машины обладают одинаковыми наборами свойств и методов.

Теперь вы можете запускать/останавливать двигатель и вести машину, используя одни и те же имена методов.

Работа по интеграции свойства fuel в объект машины еще не завершена. Например, можно ли запустить двигатель при отсутствии бензина? Помогите нам интегрировать свойство fuel в код. Для этого проверьте состояние fuel перед запуском двигателя. Если метод start вызывается при пустом бензобаке, выведите какое-нибудь осмысленное сообщение для водителя, например: «**The car is on empty, fill up before starting!**» Перепишите метод start, добавьте его в код и протестируйте. Ниже приведено наше решение.

The page at localhost says:
The car is on empty, fill up before starting!
OK

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    fuel: 0,
    started: false,

    start: function() {
        if (this.fuel == 0) {
            alert("The car is on empty, fill up before starting!");
        } else {
            this.started = true;
        }
    },

    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
                this.fuel = this.fuel - 1;
            } else {
                alert("Uh oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel: function(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

Модель DOM

Вы значительно продвинулись в изучении JavaScript. Фактически вы из новичка в области сценарного программирования превратились в... **программиста.** Впрочем, кое-чего не хватает: для полноценного использования ваших навыков JavaScript необходимо уметь взаимодействовать с веб-страницей, в которой располагается ваш код. Только в этом случае вы сможете писать **динамические** страницы, которые реагируют на действия пользователя и обновляются после загрузки. Как взаимодействовать со страницей? Через объектную модель документа **DOM** (**Document Object Model**). В этой главе мы рассмотрим DOM и общие принципы работы с этой моделью из JavaScript для расширения возможностей страницы.

В предыдущей главе мы предложили вам одну головоломку на «вскрытие кода»

Вы получили разметку HTML с кодом во внешнем файле с сайта доктора Зло. Разметка выглядела так:

Это HTML.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dr. Evel's Secret Code Page</title>
  </head>
  <body>
    <p id="code1">The eagle is in the</p>
    <p id="code2">The fox is in the</p>
    <p id="code3">snuck into the garden last night.</p>
    <p id="code4">They said it would rain</p>
    <p id="code5">Does the red robin crow at</p>
    <p id="code6">Where can I find Mr.</p>
    <p id="code7">I told the boys to bring tea and</p>
    <p id="code8">Where's my dough? The cake won't</p>
    <p id="code9">My watch stopped at</p>
    <p id="code10">barking, can't fly without umbrella.</p>
    <p id="code11">The green canary flies at</p>
    <p id="code12">The oyster owns a fine</p>
    <script src="code.js"></script>
  </body>
</html>
```

Обратите внимание: каждому абзацу присваивается идентификатор (id).

Включение кода JavaScript....

```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

document — глобальный объект.

А getElementById — метод.

Проверьте регистр символов в имени метода getElementById, иначе программа работать не будет!

Обратите внимание на точечную запись: похоже на объект со свойством innerHTML.

И вам предложили определить пароль доктора Зло, для этого нужно было применить свои дедуктивные способности.

Что же делает этот код?

Вы узнаете об объектах document и element в этой главе.

Давайте разберем эту программу и выясним, как доктор Зло генерирует пароли. А когда каждый шаг будет очевиден, вы начнете понимать, как работает код в целом:

1 Сначала программа присваивает переменной `access` результат вызова метода `getElementById` объекта `document`, с передачей аргумента "code9". Метод возвращает объект element.

```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Получение элемента с идентификатором "code9" — вот этого элемента...

```
<p id="code9">My watch stopped at</p>
```

2 Затем мы берем этот элемент (то есть элемент с идентификатором "code9") и используем его свойство `innerHTML` для получения его содержимого, которое присваивается переменной `code`.

```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

Элемент с идентификатором "code9" является элементом абзаца, а содержимое элемента (его «внутренняя разметка HTML») представляет собой текст «My watch stopped at».

3 Код доктора Зло присоединяет строку " midnight" к строке, содержащейся в `code` (то есть «My watch stopped at»). Затем страница отображает окно с паролем, хранящимся в переменной `code`.

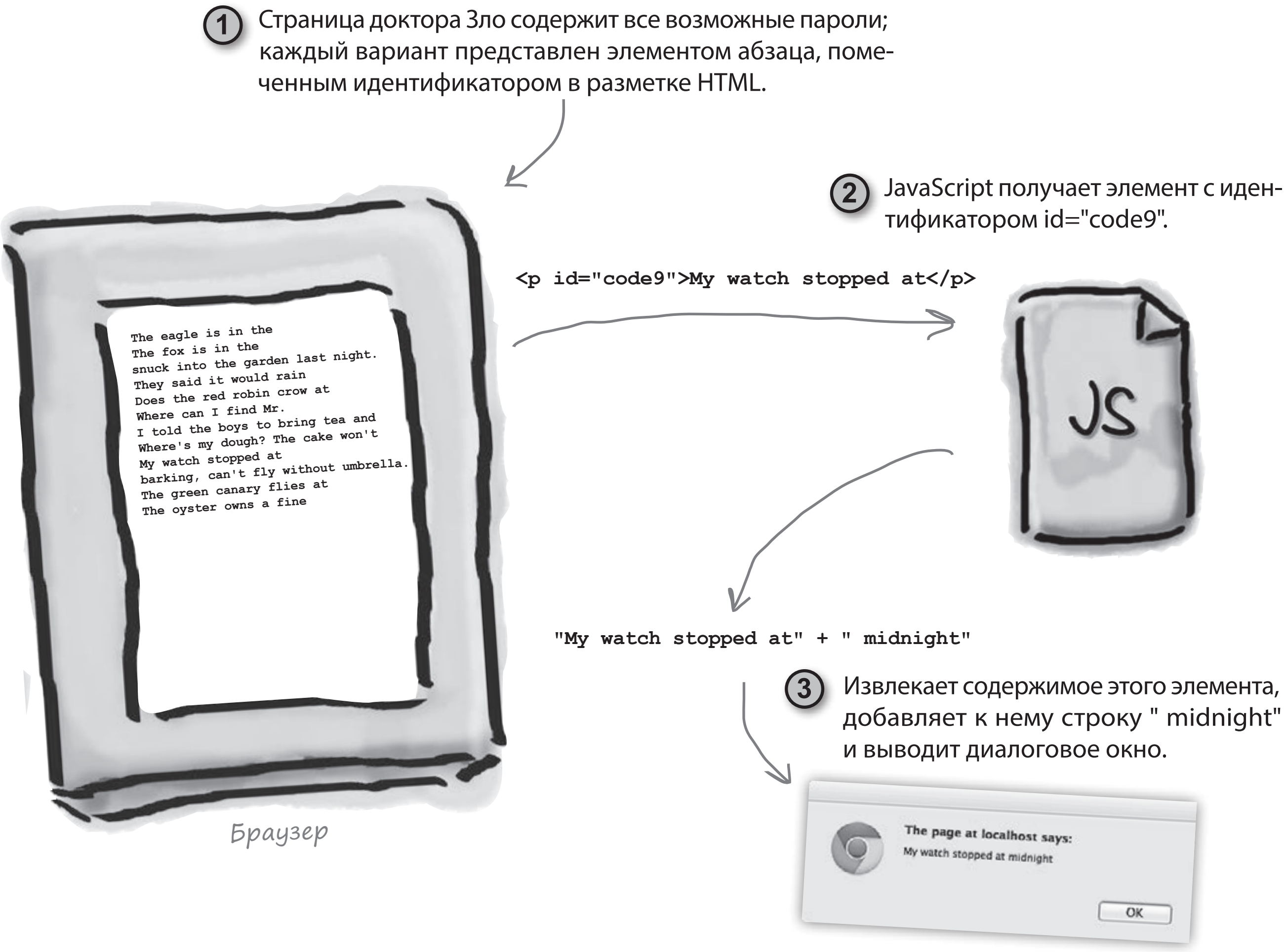
```
var access =
  document.getElementById("code9");
var code = access.innerHTML;
code = code + " midnight";
alert(code);
```

The page at localhost says:
My watch stopped at midnight
OK

Итак, мы добавляем " midnight" к строке «My watch stopped at», получаем строку «My watch stopped at midnight» и выводим ее в диалоговом окне.

В двух словах

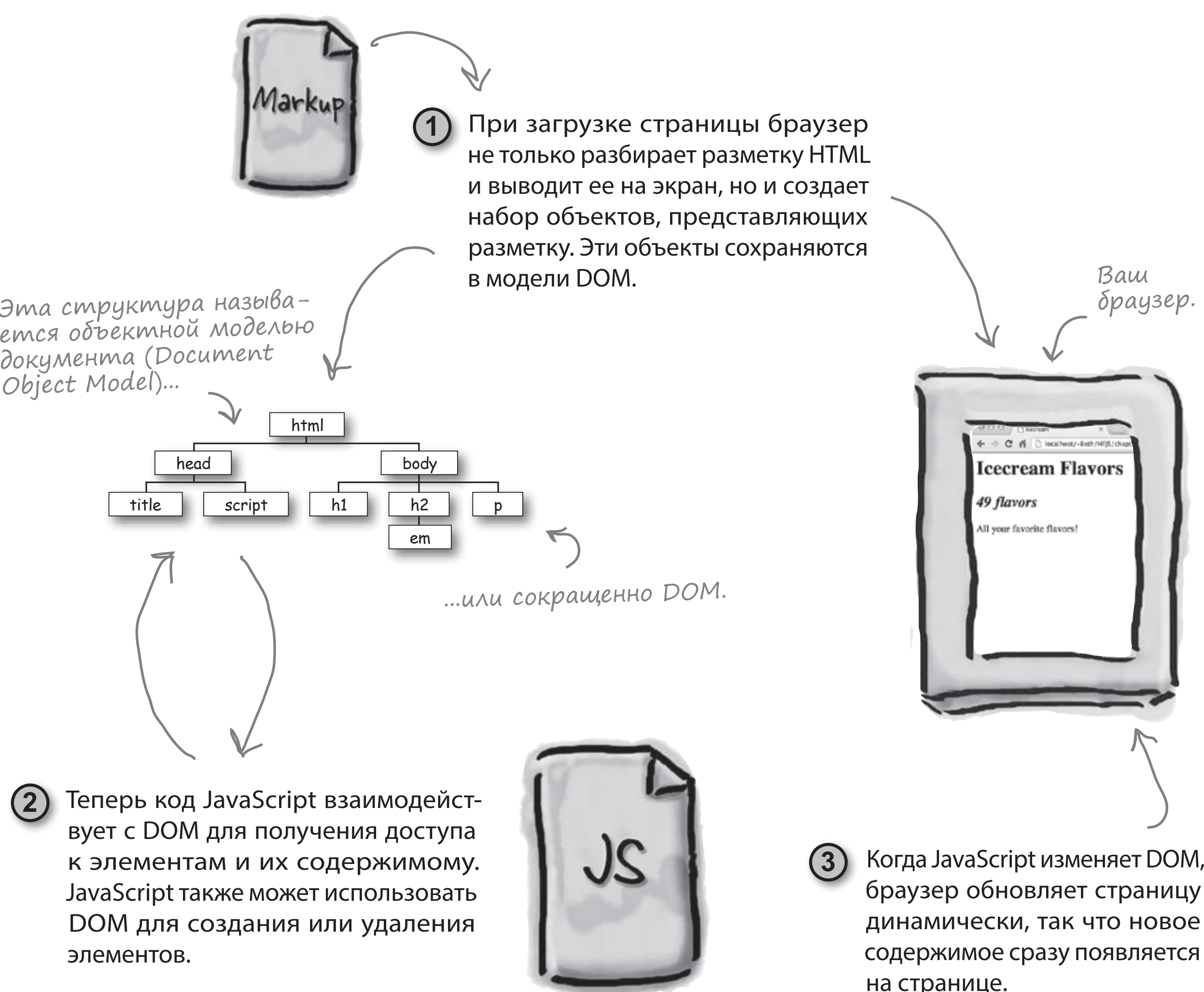
Еще раз — что же произошло? Наш код JavaScript обратился к странице (также называемой *документом*), получил элемент (с идентификатором `"code9"`), взял содержимое этого элемента (строка `"My watch stopped at"`), добавил к ней строку `" midnight"` и вывел результат на экран.



Остается пожелать доктору Зло лучше следить за безопасностью, но для нас сейчас важно другое: веб-страница представляет собой живую, подвижную *структуру данных*, с которой может взаимодействовать ваш код JavaScript. Вы можете обращаться к элементам страницы и читать их содержимое. Или другой вариант: JavaScript можно использовать для изменения структуры страницы. Но для этого необходимо немного отступить и разобраться, как же JavaScript работает с HTML.

Как JavaScript на самом деле взаимодействует со страницей

JavaScript и HTML — две разных сущности: HTML — разметка, а JavaScript — код. Как же они взаимодействуют? Все происходит через представление страницы, называемое *объектной моделью документа*, или сокращенно DOM. Откуда берется модель DOM? Она создается при загрузке страницы браузером. Вот как это происходит:



Как приготовить модель DOM

Давайте возьмем фрагмент разметки и создадим для него модель DOM. Для этого можно воспользоваться простым рецептом:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My blog</title>
  <script src="blog.js"></script>
</head>
<body>
  <h1>My blog</h1>
  <div id="entry1">
    <h2>Great day bird watching</h2>
    <p>
      Today I saw three ducks!
      I named them
      Huey, Louie, and Dewey.
    </p>
    <p>
      I took a couple of photos...
    </p>
  </div>
</body>
</html>
```

Ингредиенты

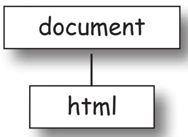
Страница HTML5 в правильном формате — 1 шт., Современный браузер, запущенный и готовый к работе, — 1 шт.

Инструкции

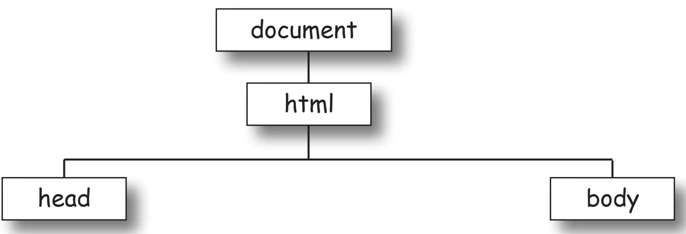
1. Начните с создания узла document на верхнем уровне.

document

2. Возьмите элемент страницы HTML верхнего уровня (в нашем случае <html>) и добавьте как дочерний элемент по отношению к document (этот элемент становится текущим).



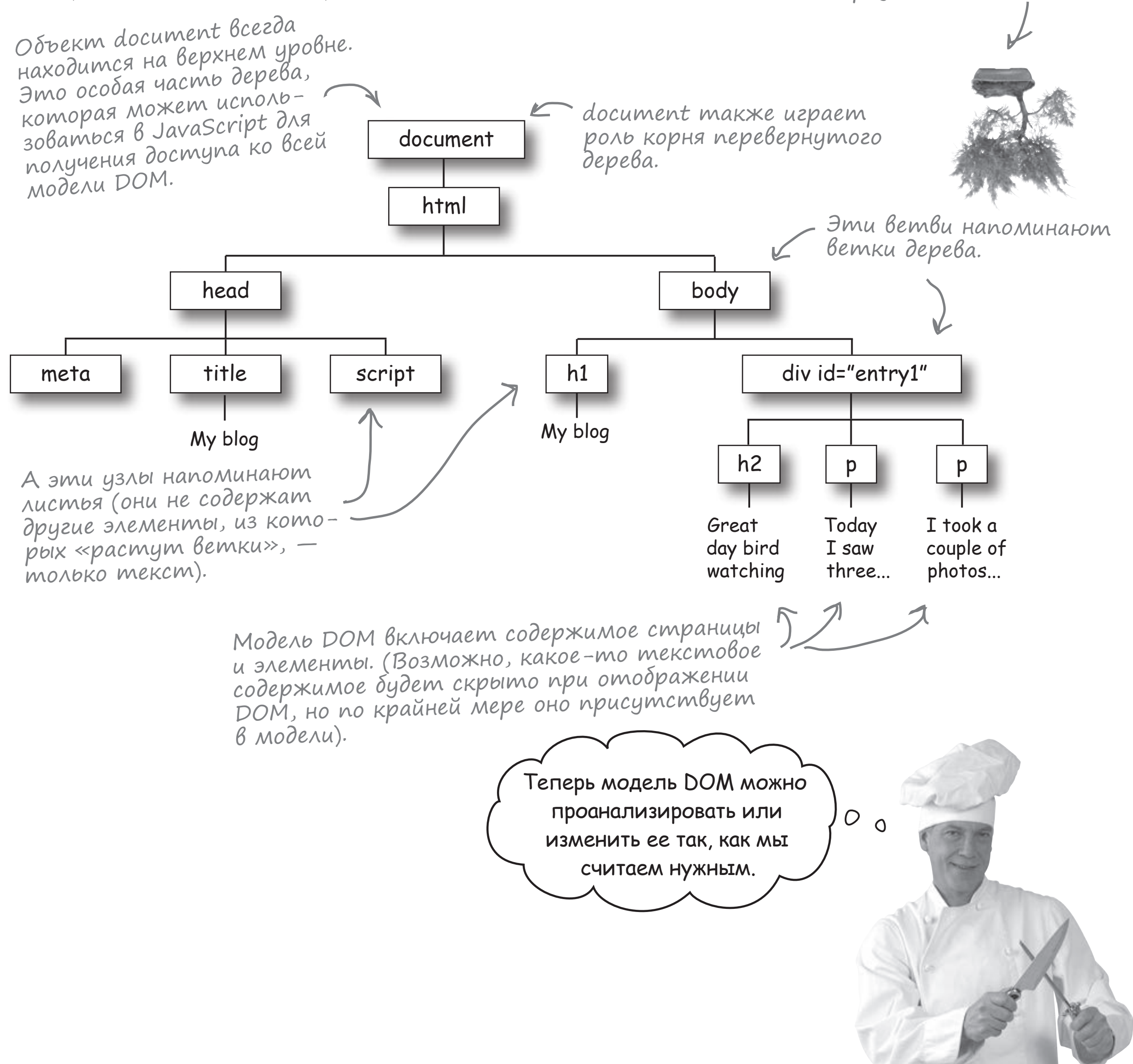
3. Для каждого элемента, вложенного в текущий элемент, добавьте этот элемент как дочерний по отношению к текущему элементу в модели DOM.



4. Вернитесь к пункту 3 для каждого добавленного элемента. Повторяйте до тех пор, пока все элементы не будут обработаны.

DOM: первые впечатления

Если вы последовали рецепту создания DOM, то у вас получилась структура наподобие той, что изображена ниже. Корнем каждой модели DOM является объект document, а под ним располагается дерево с ветвями и листовыми узлами для всех элементов разметки HTML. Пожалуй, на нее стоит взглянуть поближе.



СТАНЬ браузером

Представьте себя на месте браузера. Ваша задача — разобрать разметку HTML и построить для нее модель DOM. Используйте разметку, приведенную справа, и нарисуйте свою модель DOM внизу. Мы уже начали рисовать ее за вас.

Сравните свой ответ с нашим решением, приведенным в конце главы.

Movies

Movie Showtimes

Plan 9 from Outer Space

Playing at 3:00pm, 7:00pm. Special showing tonight at *midnight*!

Forbidden Planet

Playing at 5:00pm, 9:00pm.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Movies</title>
  </head>
  <body>
    <h1>Movie Showtimes</h1>
    <h2 id="movie1">Plan 9 from Outer Space</h2>
    <p>Playing at 3:00pm, 7:00pm.
      <span>
        Special showing tonight at <em>midnight</em>!
      </span>
    </p>
    <h2 id="movie2">Forbidden Planet</h2>
    <p>Playing at 5:00pm, 9:00pm.</p>
  </body>
</html>
```

document

html

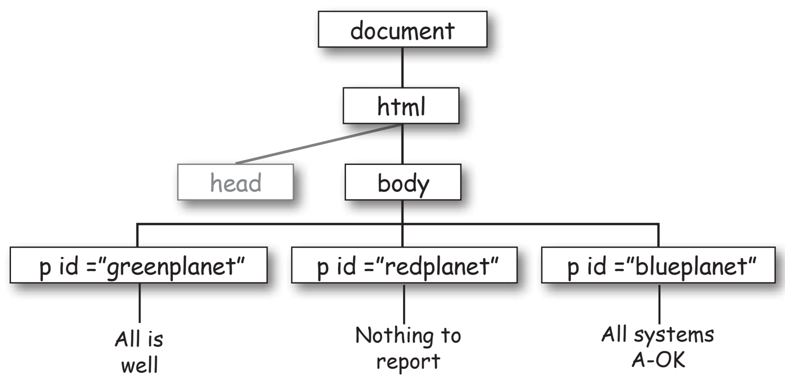
Здесь нарисуйте свою модель DOM.

Как две разные терминологии связали свою судьбу.

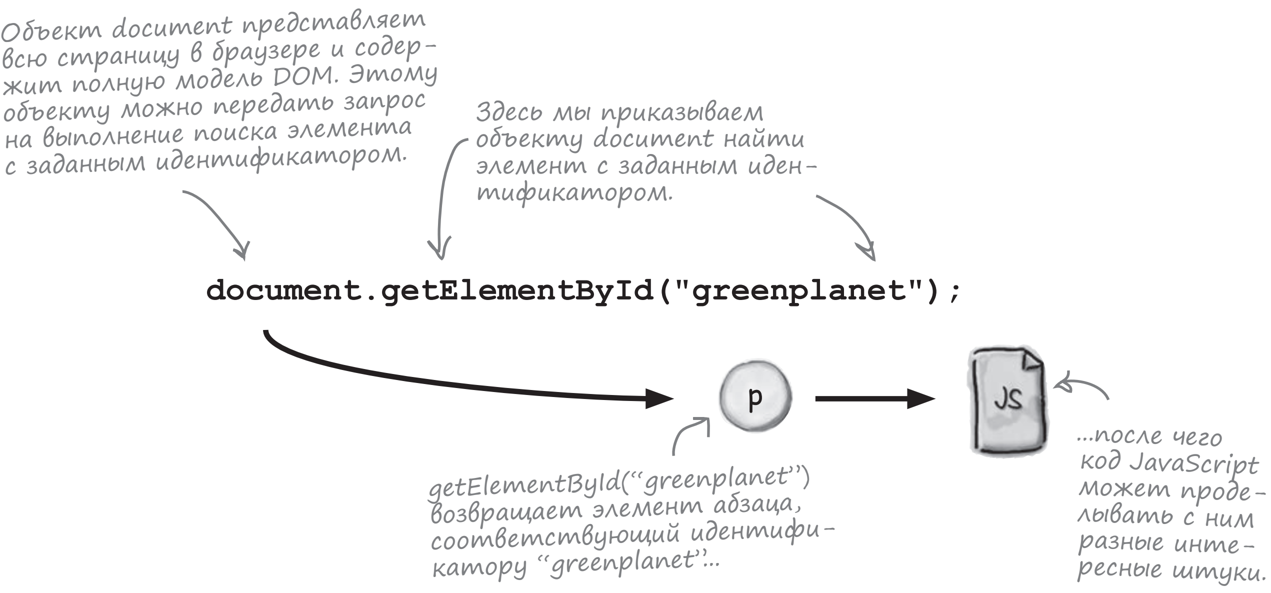
Безусловно, HTML и JavaScript — обитатели разных планет. Нужны доказательства? ДНК HTML состоит из декларативной разметки для описания набора вложенных элементов, образующих страницы. С другой стороны, JavaScript состоит из алгоритмического генетического материала, предназначенного для описания вычислений.

Может, они настолько различаются, что не могут даже общаться? Конечно, нет! Ведь у них есть нечто общее: DOM. Через модель DOM код JavaScript может взаимодействовать со страницей, и наоборот. Такое взаимодействие осуществляется несколькими способами, но сейчас мы сосредоточимся на одном — небольшой «лазейке», через которую код JavaScript может получить доступ к любому элементу страницы. Эту лазейку открывает метод *getElementById*.

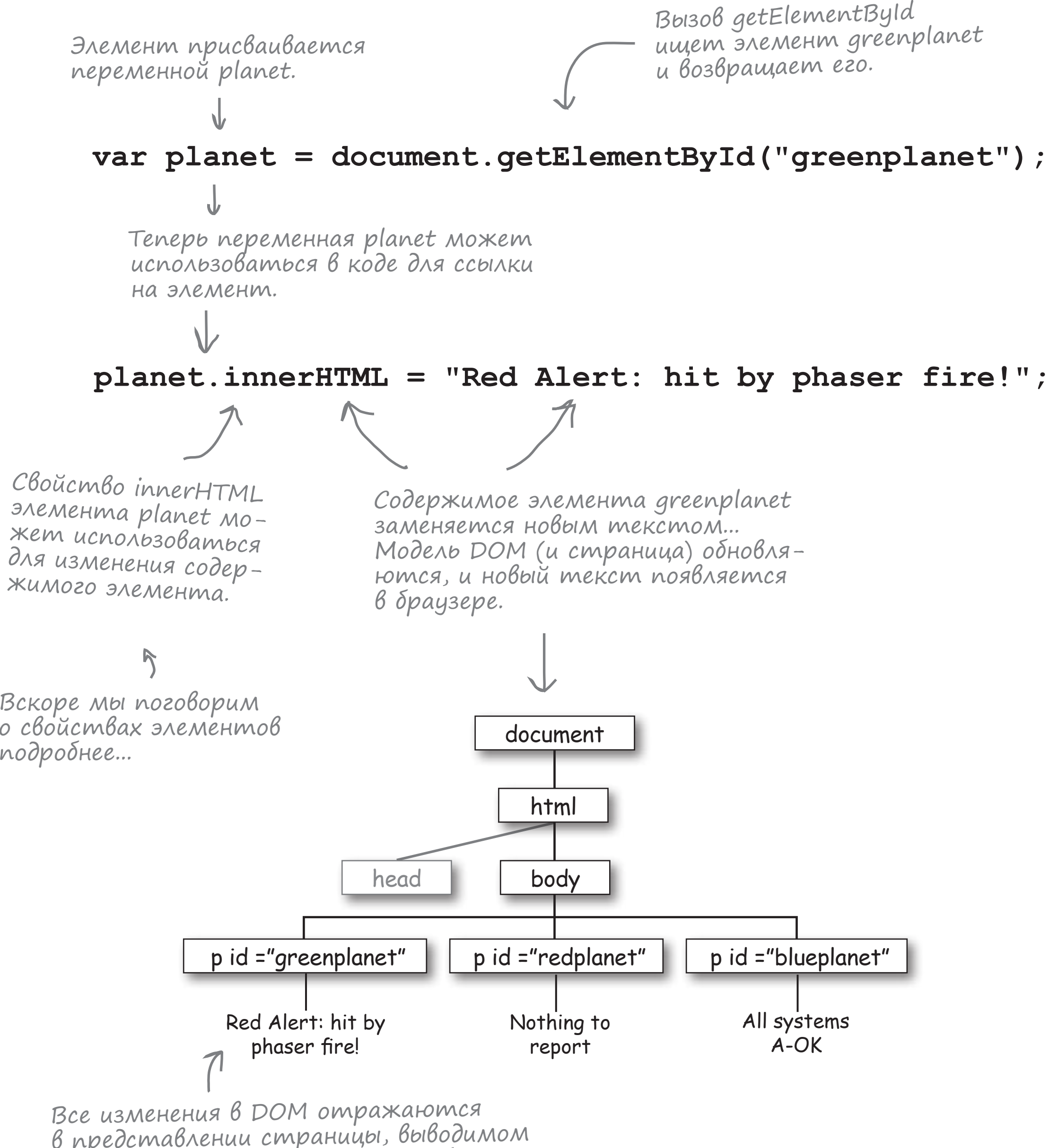
Начнем с модели DOM. Ниже изображена простая модель DOM; она состоит из нескольких абзацев HTML, каждый из которых снабжен идентификатором id. Каждый абзац содержит некоторый текст. Конечно, элемент <head> тоже присутствует, но мы опустили лишние подробности, чтобы не усложнять ситуацию.



А теперь используем JavaScript для того, чтобы сделать разметку более интересной. Допустим, текст с идентификатором greenplanet ("All is well") нужно заменить на "Red Alert: hit by phaser fire!", например, из-за действий пользователя или на основании данных, полученных от веб-службы. К этой теме мы еще вернемся, а пока просто обновим текст greenplanet. Для этого нам понадобится элемент с идентификатором "greenplanet". Следующий код выполняет обновление:



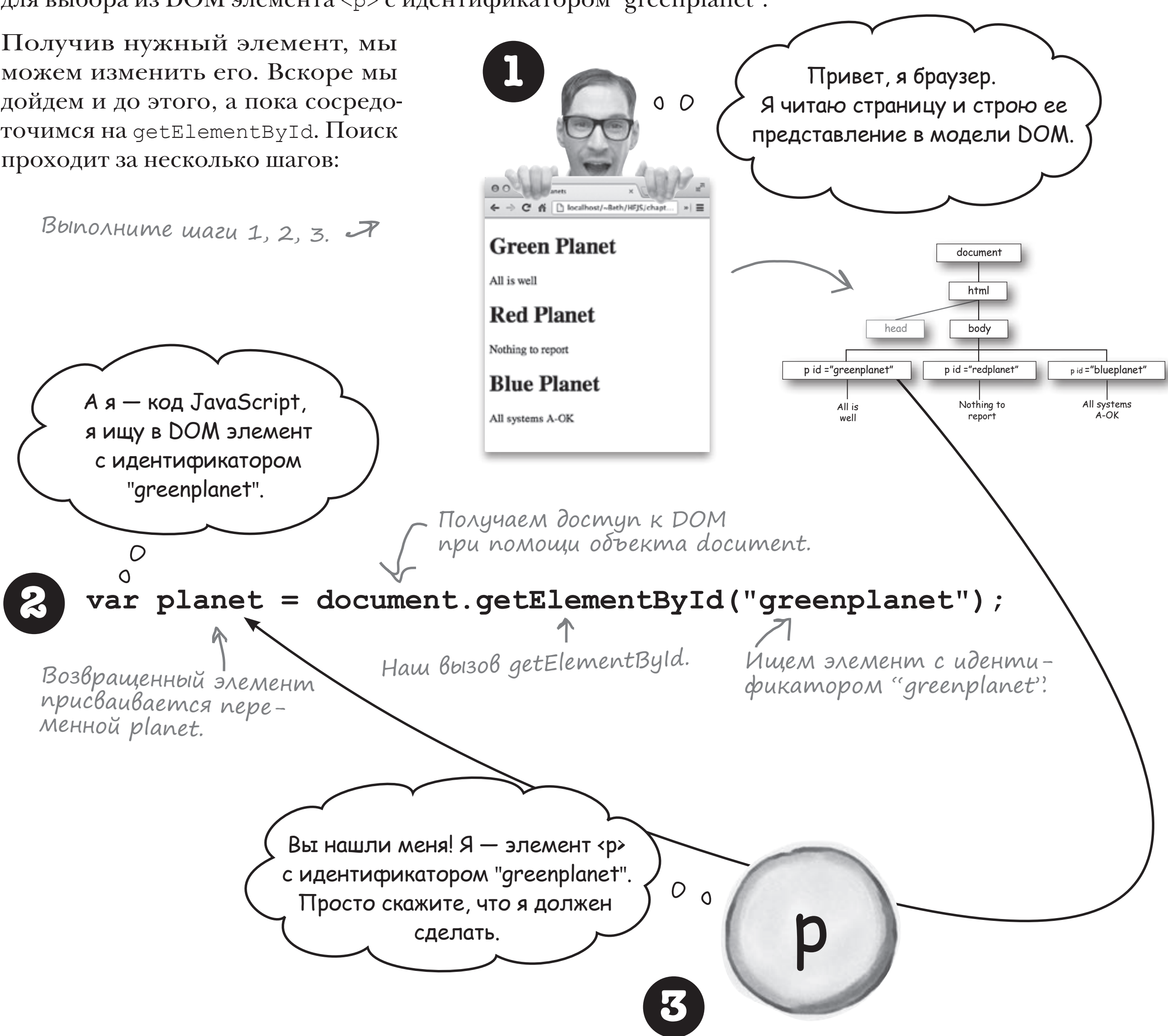
Получив элемент от getElementById, вы можете выполнить с ним какую-нибудь операцию (например, заменить его текст). Для этого элемент обычно присваивается переменной, чтобы на него можно было ссылаться в коде. Давайте сделаем это, а потом изменим текст:



Получение элемента методом getElementById

Что же мы только что проделали? Давайте разберемся. Мы используем объект `document` для получения доступа к DOM из нашего кода. Встроенный объект `document` содержит целый набор свойств и методов, в том числе и `getElementById`, который может использоваться для получения элементов из DOM. Метод `getElementById` получает идентификатор и возвращает соответствующий элемент. Возможно, в прошлом вы использовали идентификаторы для выбора и стилевого оформления элементов средствами CSS. Но здесь идентификатор используется для выбора из DOM элемента `<p>` с идентификатором "greenplanet".

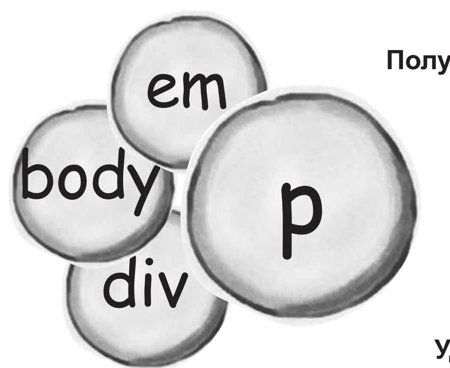
Получив нужный элемент, мы можем изменить его. Вскоре мы дойдем и до этого, а пока сосредоточимся на `getElementById`. Поиск проходит за несколько шагов:



Что именно мы получаем от DOM?

Запрашивая элемент из DOM методом `getElementById`, вы получаете объект *element*, который может использоваться для чтения, изменения или замены содержимого или атрибутов элемента. Тут-то и происходит настоящее волшебство: при изменении элемента также изменяется отображаемая страница.

Впрочем, обо всем по порядку. Для начала повнимательнее присмотримся к элементу, извлеченному из DOM. Мы знаем, что данный объект element представляет элемент `<p>` нашей страницы с идентификатором "greenplanet" и что этот элемент содержит текст "All is well". Объект element, как и все остальные разновидности объектов JavaScript, обладает свойствами и методами. Свойства и методы объекта element позволяют читать и изменять элемент. Несколько операций, которые могут выполняться с объектами element:



Получение содержимого (текст или HTML).

Изменение содержимого.

Чтение атрибута.

Добавление атрибута.

Изменение атрибута.

Удаление атрибута.

Операции, которые могут выполняться с объектом element.

Операция, которую мы хотим выполнить со своим элементом (напомним, что это элемент `<p>` с идентификатором "greenplanet") — замена содержимого «All is well» на «Red Alert: hit by phaser fire!». Объект element присвоен переменной `planet` в нашем коде; воспользуемся этой переменной для изменения одного из свойств объекта, а именно `innerHTML`:

Переменная planet содержит объект element для элемента <p> с идентификатором "greenplanet".

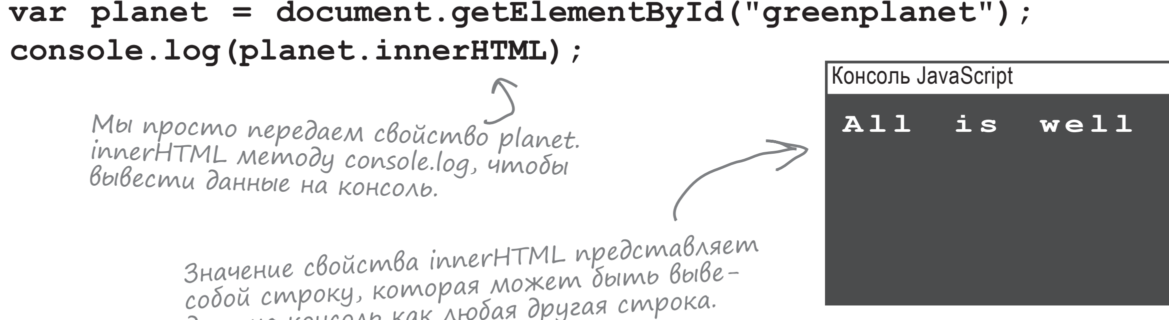
```
var planet = document.getElementById("greenplanet");

planet.innerHTML = "Red Alert: hit by phaser fire!";
```

Свойство innerHTML объекта element может использоваться для изменения содержимого элемента!

В поисках внутреннего HTML

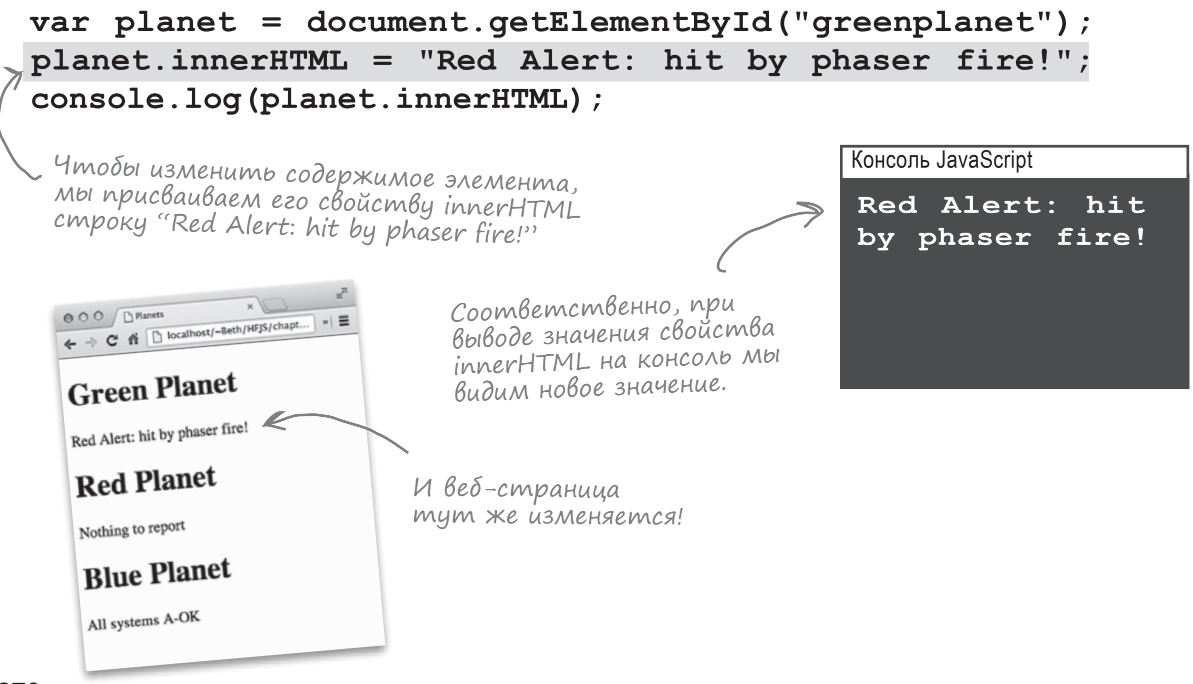
Важное свойство `innerHTML` используется для чтения и замены содержимого элемента. В `innerHTML` возвращается внутреннее содержимое элемента, без тегов элемента HTML. Давайте в порядке эксперимента выведем на консоль содержимое объекта элемента `planet`, используя свойство `innerHTML`. Вот что мы получим:

```
var planet = document.getElementById("greenplanet");
console.log(planet.innerHTML);
```



Теперь попробуем изменить значение свойства `innerHTML`. Это приводит к изменению содержимого элемента `<p>` с идентификатором "greenplanet" на странице, поэтому вы увидите, что страница тоже изменяется!

```
var planet = document.getElementById("greenplanet");
planet.innerHTML = "Red Alert: hit by phaser fire!";
console.log(planet.innerHTML);
```



Освежаем воспоминания

Пожалуй, стоит ненадолго отвлечься от основной темы. Возможно, вы говорите себе: «Погоди, что-то такое было с идентификаторами и классами, хотя я и не помню подробностей... Кажется, они имели какое-то отношение к CSS?» Нет проблем — давайте немного передохнем, освежим в памяти кое-какую полезную информацию — и вы в два счета будете готовы продолжить путь...

В HTML атрибут id предоставляет возможность однозначной идентификации элементов и их выбора в CSS для стилевого оформления. И как вы уже видели, выбор элементов по идентификаторам реализован и в JavaScript.

Давайте посмотрим на пример:

```
<div id="menu">
 ...
</div>
```

Этому элементу <div> присваивается уникальный идентификатор "menu". На странице не должно быть других элементов с идентификатором "menu".

После того как элемент будет определен, его можно выбрать в CSS для применения стилевого оформления. Это делается так:

div#menu является селектором id.

```
div#menu {
    background-color: #aaa;
}
```

div#menu выбирает элемент <div> с идентификатором menu, чтобы мы могли применить стиль к этому (и только этому!) элементу.

К элементу можно обратиться по идентификатору из кода JavaScript:

```
var myMenu = document.getElementById("menu");
```

Однако следует помнить, что существует другой способ пометки элементов: с использованием классов. Классы предназначены для пометки целых групп элементов:

```
<h3 class="drink">Strawberry Blast</h3>
<h3 class="drink">Lemon Ice</h3>
```

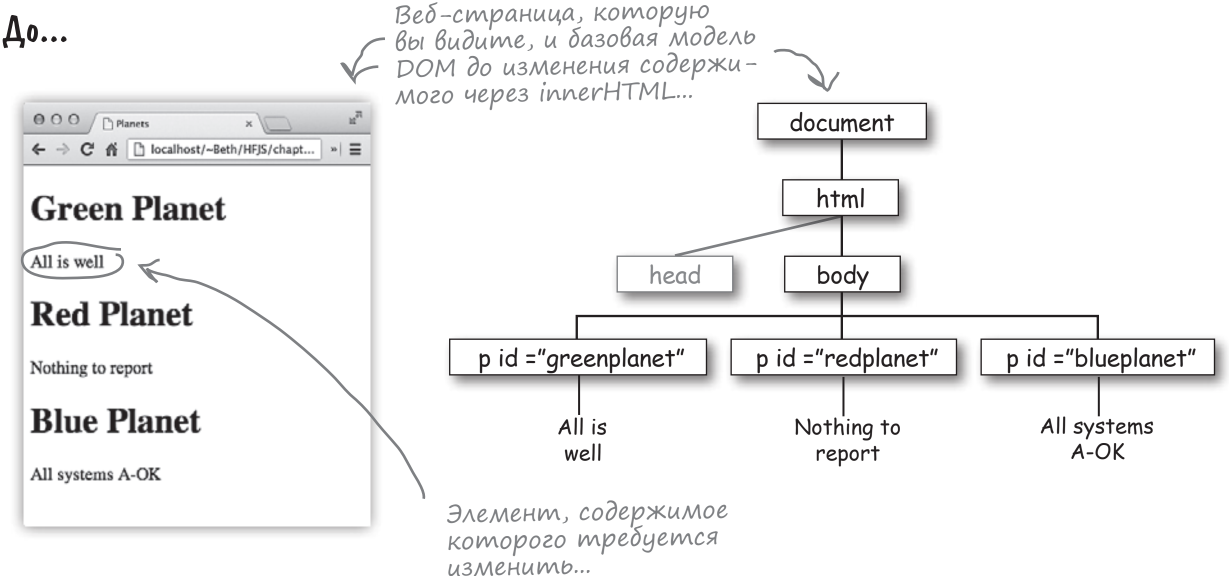
Оба элемента <h3> относятся к классу "drink". Класс — это группа, которая может состоять из нескольких элементов.

Элементы можно выбирать по классам — как в CSS, так и в JavaScript. Вскоре вы научитесь работать с классами в JavaScript. А если этого краткого напоминания окажется недостаточно — обратитесь к главе 7 книги «*Изучаем HTML, XHTML и CSS*», или к вашему любимому учебнику по HTML и CSS.

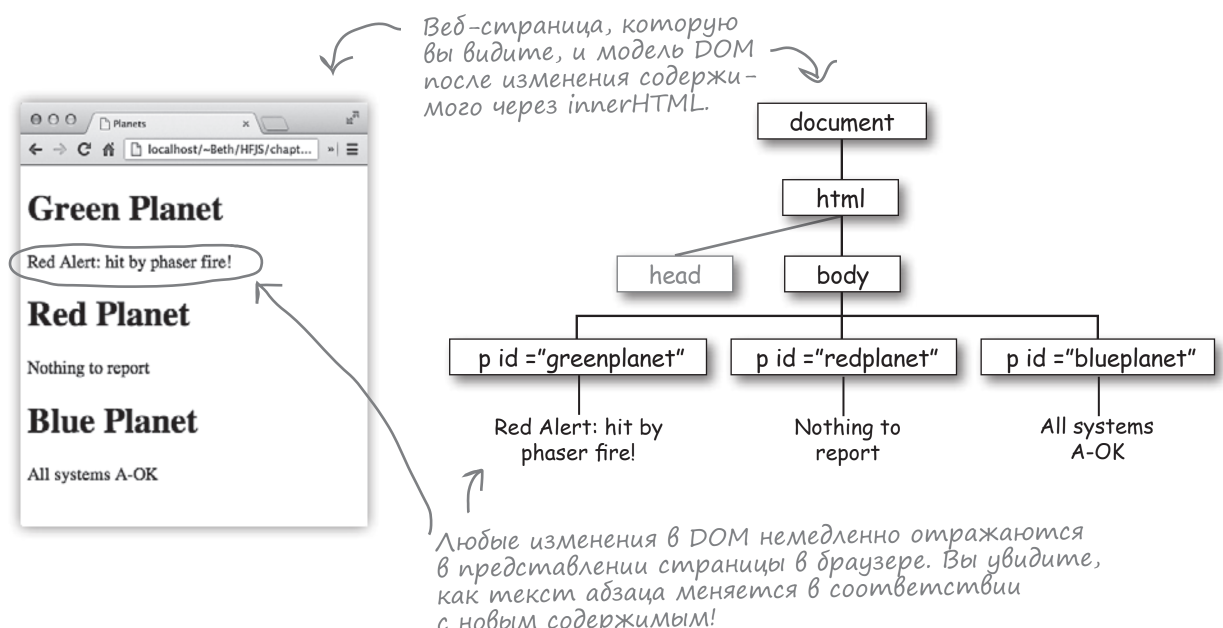
Что происходит при внесении изменений в DOM

Итак, что же именно происходит при изменении содержимого элемента с использованием `innerHTML`? Фактически вы изменяете непосредственное содержимое веб-страницы «на ходу». Изменения, вносимые в содержимое в DOM, немедленно появляются на веб-странице.

До...



...и после.



В: А что произойдет, если при вызове document.getElementById передается несуществующий идентификатор?

О: Если вы пытаетесь получить элемент из DOM по идентификатору, отсутствующему в модели, вызов getElementById возвращает значение null. При использовании getElementById стоит выполнить проверку на null перед тем, как пытаться обращаться к свойствам элемента. Значение null более подробно рассматривается в следующей главе.

В: Можно ли использовать document.getElementById для выбора элементов по классу — скажем, если я хочу получить группу элементов класса “planets”?

О: Нет, но вы мыслите в правильном направлении. Метод getElementById позволяет выбирать элементы по идентификатору. Однако существует другой метод DOM — getElementsByClassName, который может использоваться для получения элементов по имени класса. С этим методом вы получаете набор элементов, принадлежащих классу (так как один класс может содержать множество элементов). Кроме этого существует метод, возвращающий набор элементов — getElementsByTagName. Он возвращает все элементы с заданным именем тега. Мы рассмотрим метод getElementsByTagName немного позднее. Тогда же вы увидите, как обрабатывается возвращаемая им группа элементов.

В: Что конкретно представляет собой объект element?

О: Отличный вопрос. Объект element является внутренним представлением элемента разметки, который вы включаете в свой файл HTML, например <p>some text</p>. В процессе загрузки и разбора файла HTML браузер создает объект element для каждого элемента страницы и добавляет все эти объекты в модель DOM. Таким образом, модель DOM представляет собой дерево объектов element. Помните, что объекты element, как и любые другие объекты, содержат свойства (такие, как innerHTML) и методы. Некоторые свойства и методы объектов element будут более подробно описаны далее.

В: Логично было бы включить в объект element свойство с именем “content” или хотя бы “html”. Почему вместо этого используется имя innerHTML?

О: Мы согласны, имя выбрано странное. Свойство innerHTML представляет все содержимое элемента, включая вложенные элементы (например, абзац помимо текста абзаца может включать элементы <em> и <img>). Существует ли парное свойство outerHTML? Да! Оно возвращает всю разметку HTML внутри элемента, а также сам элемент. На практике outerHTML используется довольно редко, тогда как свойство innerHTML сплошь и рядом применяется для обновления содержимого элементов.

В: Итак, присваивая что-то innerHTML, я могу заменить содержимое любого элемента новым содержимым. А что будет, если я, допустим, заменю содержимое элемента <body>?

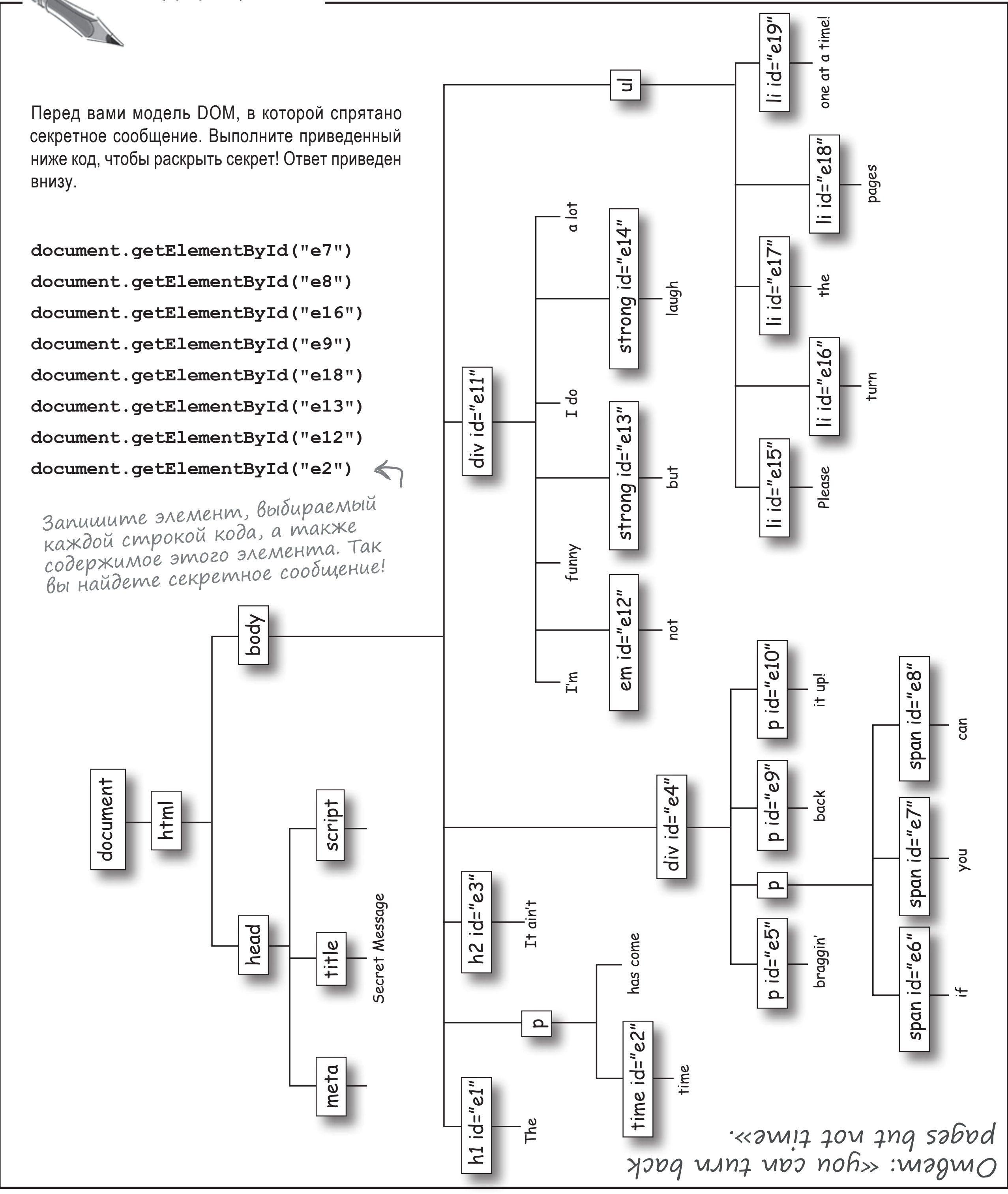
О: Верно, свойство innerHTML предоставляет удобные средства для замены содержимого элемента. И вы правы, innerHTML может использоваться для замены содержимого элемента <body>, в результате чего вся страница будет заменена совершенно новой страницей.

Возьми в руку карандаш

Перед вами модель DOM, в которой спрятано секретное сообщение. Выполните приведенный ниже код, чтобы раскрыть секрет! Ответ приведен внизу.

```
document.getElementById("e7")
document.getElementById("e8")
document.getElementById("e16")
document.getElementById("e9")
document.getElementById("e18")
document.getElementById("e13")
document.getElementById("e12")
document.getElementById("e2")
```

Запишите элемент, выбираемый каждой строкой кода, а также содержимое этого элемента. Так вы найдете секретное сообщение!



Ответ: «you can turn back pages but not time».

Планетарный тест-драйв

Теперь вы знаете, как использовать метод `document.getElementById` для обращения к элементам, а свойство `innerHTML` — для изменения содержимого элемента. Давайте применим новые знания на практике.

Ниже приведена разметка HTML нашего примера с планетами. В секцию `<head>` включен элемент `<script>` с кодом, а в теле документа определяются три абзаца. Если вы не сделали этого ранее, введите разметку HTML и код JavaScript для обновления DOM:

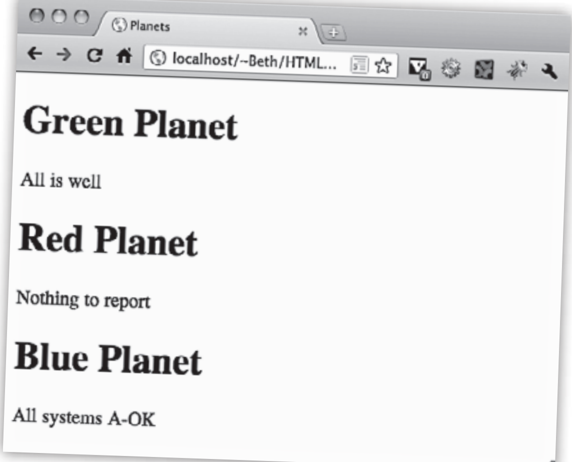
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <script>
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

Элемент script с кодом.

Как было показано ранее, мы получаем элемент <p> с идентификатором “greenplanet” и изменяем его содержимое.

Элемент <p>, который будет изменяться из кода JavaScript.

После того как вся разметка будет введена, загрузите страницу в браузере и посмотрите, как работает волшебство DOM.



СТОП! Хьюстон, у нас проблемы. В абзаце “greenplanet” по-прежнему выводится текст «All is well». Что не так?

Я трижды проверил
и перепроверил разметку
и код — все равно не работает.
Никакие изменения на странице
не отражаются.

Да, мы забыли упомянуть один важный момент. При работе с DOM очень важно выполнять код только *после полной загрузки* страницы. Если это условие будет нарушено, существует большая вероятность того, что модель DOM не будет построена к моменту выполнения кода.

Давайте подумаем, что только что произошло: код включен в раздел `<head>` страницы, поэтому он начинает выполняться до того, как браузер увидит остальную часть страницы. И это создает большую проблему, потому что элемент абзаца с идентификатором “greenplanet” еще не существует.

Что же именно произошло? Вызов `getElementById` возвращает null вместо искомого элемента. Браузер добросовестно двигается дальше и все равно выводит страницу, но без изменения содержимого абзаца с идентификатором “greenplanet”.

Как решить проблему? Можно переместить код в конец раздела `<body>`, но существует более надежный способ гарантировать выполнение этого кода в правильное время – приказать браузеру: «Выполни этот код только после полной загрузки страницы и построения модели DOM». Давайте посмотрим, как это делается.

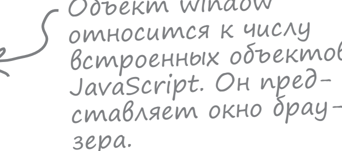
Проверьте консоль при загрузке страницы — эта ошибка выводится в большинстве браузеров. Консоль также является полезным инструментом отладки.

```
Uncaught TypeError:
Cannot set property
'innerHTML' of null
```

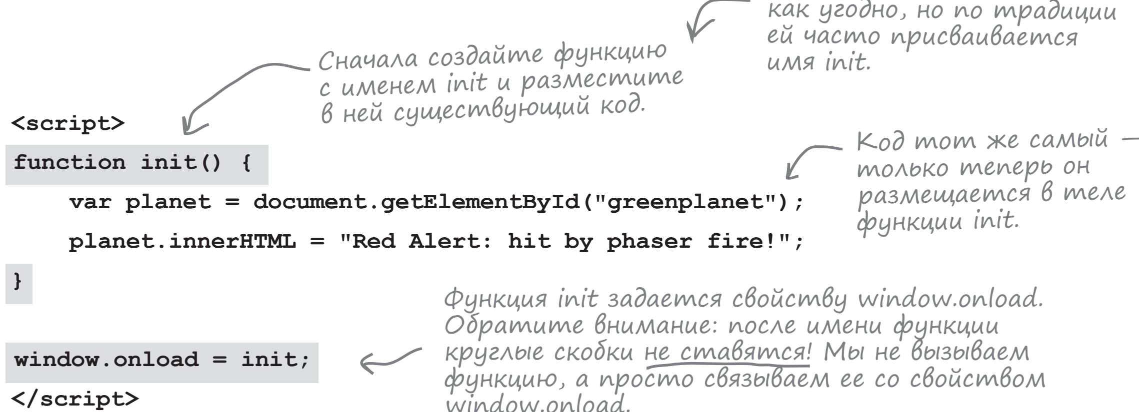
И не вздумайте выполнять мой код до того, как страница будет загружена!

Да, но как? Кроме перемещения кода в конец тела страницы, существует и другое — и пожалуй, более элегантное решение.

Вот как это делается: сначала создается функция с кодом, который должен быть выполнен только после полной загрузки страницы. Эта функция присваивается свойству `onload` объекта window.



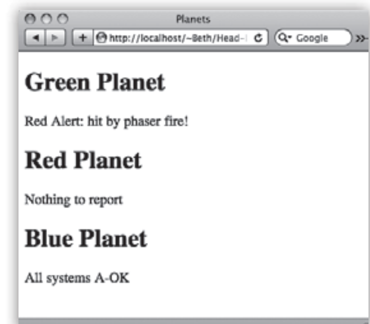
Как это работает? Объект window вызовет любую функцию, связанную со свойством `onload`, но только после того, как страница будет полностью загружена. Спасибо разработчикам объекта window, предусмотревшим механизм определения кода, гарантированно вызываемого после завершения загрузки страницы. Посмотрите:




```
<script>
function init() {
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
}

window.onload = init;
</script>
```

Пробуем снова...

Перезагрузите страницу с новой функцией `init` и свойством `onload`. На этот раз браузер полностью загружает страницу, строит полную модель DOM и только после этого вызывает вашу функцию `init`.



Порядок: текст абзаца изменился, как и предполагалось.

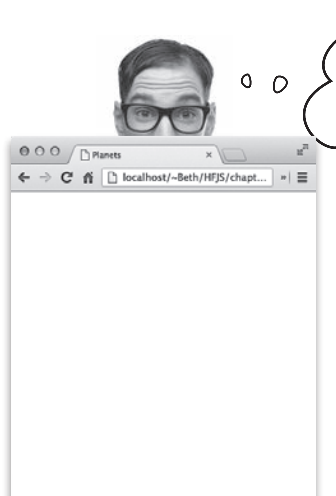
«Обработчик события», или «функция обратного вызова»

В работе `onload` стоит разобраться подробнее, потому что этот стандартный прием программирования еще не раз встретится вам в JavaScript.

Предположим, должно случиться какое-то большое и важное событие, и вам *совершенно необходимо* о нем узнать. Скажем, речь идет о событии «страница полностью загружена». В таких ситуациях обычно применяются *функции обратного вызова*, также называемые *обработчиками событий*.

Работа функций обратного вызова основана на том, что вы передаете эту функцию объекту, который знает о событии. При возникновении события объект вызывает полученную функцию, то есть оповещает вас о наступлении события. Это стандартное решение применяется в JavaScript для самых разных событий.

Функция обратного вызова, или обработчик события, — выбирайте, что больше нравится.



Получил!
Я сохраняю init
в свойстве onload, чтобы
не забыть имя функции...

Прошло немного времени

Уф, пришлось
изрядно потрудиться,
но страница все-таки загружена.
Давайте посмотрим... Мне нужно
вызвать функцию, на которую ссылается
свойство onload... Так, это функция init.
Вызываю init!

Green Planet

All is well

Red Planet

Nothing to report

Blue Planet

All systems A-OK

метод init вызывается и выполняет свою работу

Эй, браузер, — спасибо,
что не забыл вызвать init!
Все прошло без проблем!

Green Planet

Red Alert: hit by phaser fire!

И после вызова init
содержимое страницы
обновляется!

Red Planet

Nothing to report

Blue Planet

All systems A-OK

Интересно. Значит, я могу использовать функции для упаковки кода, который должен вызываться при наступлении некоторого события. Какие еще события можно обрабатывать подобным образом?

Все верно — существует много разных событий, которые вы можете обрабатывать при желании. Одни события — такие, как load, — генерируются браузером. Другие события генерируются при взаимодействии пользователя со страницей или даже инициируются кодом JavaScript.

Мы уже знаем событие «загрузка страницы завершена», которое обрабатывается функцией из свойства `onload` объекта window. Вы также можете писать обработчики для таких операций, как «вызывать эту функцию каждые пять секунд», или «от веб-службы поступили данные, которые необходимо обработать», или «пользователь нажал кнопку, нужно получить данные от формы»... и многих, многих других. Все эти типы событий интенсивно используются при создании страниц, которые больше напоминают приложения, чем статические документы. Сейчас мы в самых общих чертах познакомились с обработчиками событий, но позднее уделим им гораздо больше времени, учитывая их важную роль в программировании JavaScript.

Возьми в руку карандаш

Перед вами разметка HTML со списком песен — изначально этот список пуст. Ваша задача — дописать приведенный ниже код JavaScript, чтобы он добавлял песни в список. Заполните пробелы программными конструкциями JavaScript, которые будут решать эту задачу. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
<!doctype html>
<html lang="en">
<head>
  <title>My Playlist</title>
  <meta charset="utf-8">
  <script>
  ____________ addSongs() {
    var song1 = document.______________("________");
    var _____ = _______________________("________");
    var _____ = ________.getElementById("________");

    ________.innerHTML = "Blue Suede Strings, by Elvis Pagely";
    __________________ = "Great Objects on Fire, by Jerry JSON Lewis";
    song3.____________ = "I Code the Line, by Johnny JavaScript";
  }
  window.___________ = ____________;
  </script>
</head>
<body>
  <h1>My awesome playlist</h1>
  <ul id="playlist">
    <li id="song1"></li>
    <li id="song2"></li>
    <li id="song3"></li>
  </ul>
</body>
</html>
```

Разметка HTML-страницы.

Это наш сценарий. Программа должна заполнять список песен ниже в элементе <ul>.

Заполните пропуски в коде так, чтобы он выводил список песен.

Пустой список песен. Приведенный выше код должен добавлять текст в каждый из элементов <li> списка.

Так должна выглядеть веб-страница после загрузки, когда код JavaScript заработает.

My Playlist

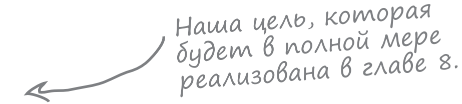
http://localhost/~Beth/Head-

My awesome playlist

- Blue Suede Strings, by Elvis Pagely
- Great Objects on Fire, by Jerry JSON Lewis
- I Code the Line, by Johnny JavaScript

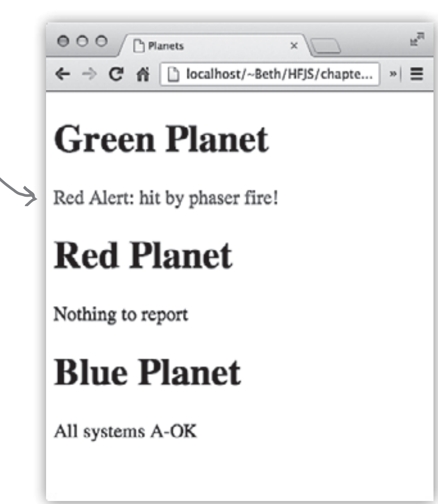
Зачем останавливаться на этом? Сделаем следующий шаг

Задумайтесь на минутку над тем, что мы сейчас сделали: мы взяли статическую веб-страницу и динамически изменили содержимое одного из ее абзацев из программного кода. На первый взгляд вроде бы просто, но на самом деле это первый шаг на пути к созданию действительно интерактивных страниц.



Сделаем второй шаг: теперь, когда вы умеете выбирать элементы из DOM, зададим атрибут элемента из программного кода.

Для чего это может быть нужно? Взять хотя бы наш простой пример с планетами. При назначении тревожного текста «Red Alert» мы можем назначить абзацу красный цвет текста. Безусловно, это поможет более наглядно передать смысл сообщения.



И вот как это будет сделано:

1. Мы определим для класса "redtext" правило CSS, которое назначает абзацу красный цвет текста. Любой элемент, добавленный в этот класс, будет выводиться красным цветом.

2. Затем мы добавим код, который берет элемент абзаца "greenplanet" и добавляет к нему класс "redtext".

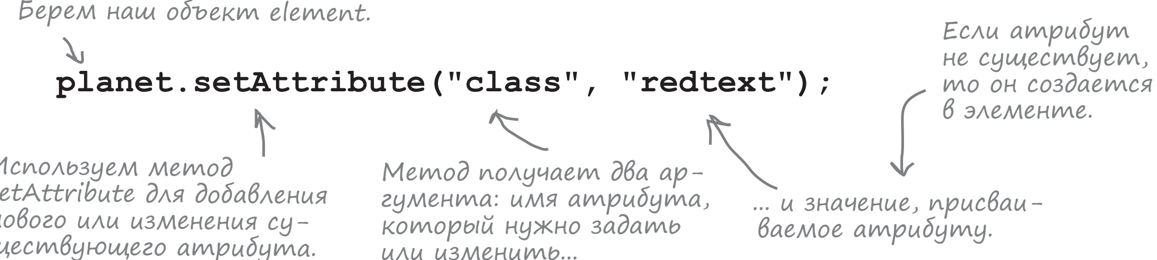
Вот и все. Осталось совсем немного — научиться задавать атрибуты элемента, и можно переходить к написанию кода.

Возьми в руку карандаш

Нет ли желания поработать другой частью мозга? Нам понадобится стиль CSS для класса "redtext", который назначает тексту абзаца красный цвет ("red"). Если вы справляетесь с такими задачами с закрытыми глазами — что ж, прекрасно. Если с тех пор, как вы писали CSS, прошло много времени, не огорчайтесь; все равно попробуйте это сделать. В любом случае, ответ приведен в конце главы.

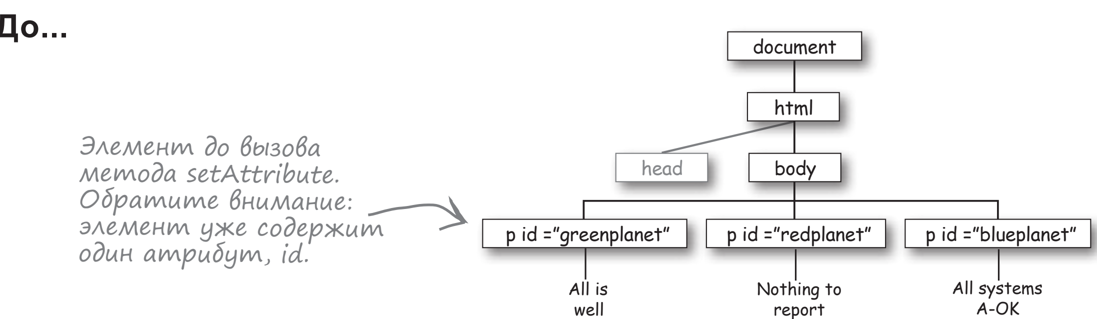
Как задать атрибут методом setAttribute

Объекты element содержат метод `setAttribute`, который может вызываться для задания атрибутов элементов HTML. Метод `setAttribute` выглядит так:

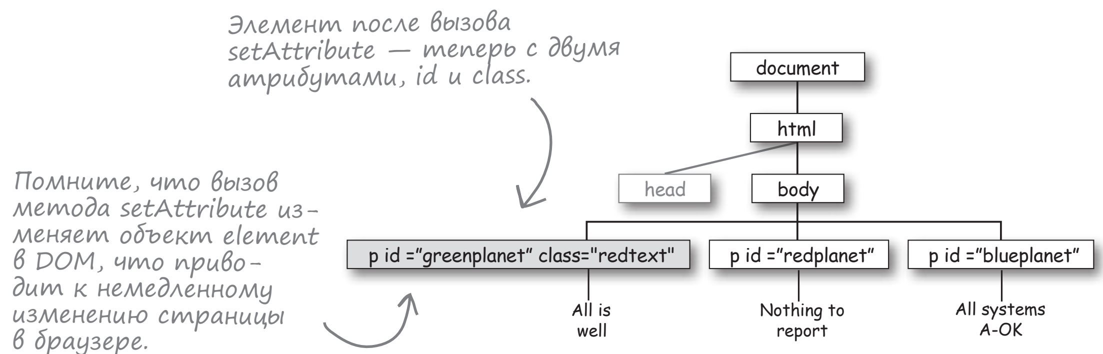


Вызов `setAttribute` для любого объекта element изменяет значение существующего атрибута, а если атрибут не существует — добавляет новый атрибут в элемент. Для примера давайте посмотрим, как выполнение приведенного выше кода отражается на модели DOM.

До...



...и после.



Веселье с атрибутами продолжается!
(значения атрибутов можно ЧИТАТЬ)

Хотите узнать значение атрибута элемента? Запросто — воспользуйтесь методом `getAttribute`, который вызывается для получения значения атрибута элемента HTML.

```
var scoop = document.getElementById("raspberry");
var altText = scoop.getAttribute("alt");
console.log("I can't see the image in the console,");
console.log(" but I'm told it looks like: " + altText);
```

Получаем ссылку на элемент методом getElementById, после чего используем метод getAttribute элемента для получения атрибута.

Передайте имя атрибута, значение которого вы хотите получить.

Что произойдет, если мой атрибут не существует в элементе?

Помните, что происходит при вызове `getElementById`, если идентификатор не существует в DOM? Вы получаете null. То же происходит и с `getAttribute`. Если атрибут не существует, вы получите null. Проверка выполняется так:

```
var scoop = document.getElementById("raspberry");
var altText = scoop.getAttribute("alt");
if (altText == null) {
    console.log("Oh, I guess there isn't an alt attribute.");
} else {
    console.log("I can't see the image in the console,");
    console.log(" but I'm told it looks like " + altText);
}
```

Проверяем, что вызов действительно вернул значение атрибута.

Если значение атрибута не определено, делаем это...

... а если определено — выводим текстовое содержимое на консоль.

И не забывайте, что getElementById тоже может вернуть null!

Каждый раз, когда вы запрашиваете некоторое значение, стоит убедиться в том, что вы получили то, что просили...

Вызов getElementById может вернуть null, если элемент с заданным идентификатором не существует в DOM. По общепринятым правилам после получения элементов стоит проверить полученную ссылку на null. Мы могли бы соблюдать это правило, но тогда книга стала бы на 1000 страниц длиннее.

А тем временем в ~~далекой~~ Галактике...

Пришло время собрать воедино весь код и провести последний тест-драйв.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <style>
    .redtext { color: red; }
  </style>
  <script>
    function init() {
        var planet = document.getElementById("greenplanet");
        planet.innerHTML = "Red Alert: hit by phaser fire!";
        planet.setAttribute("class", "redtext");
    }
    window.onload = init;
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

Вся разметка HTML, CSS и код JavaScript нашей программы.

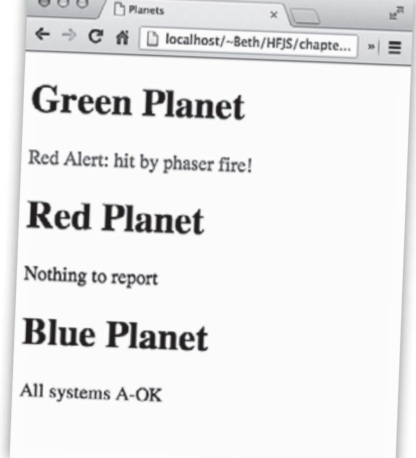
Здесь включается класс redtext, чтобы при добавлении "redtext" как значения атрибута class в нашем коде текст выводился красным цветом.

Напоминаем: мы получаем элемент "greenplanet" и сохраняем значение в переменной planet. Затем мы изменяем содержимое элемента, и, наконец, добавляем атрибут class, который окрашивает текст элемента в красный цвет.

Функция init вызывается только после того, как страница будет полностью загружена!

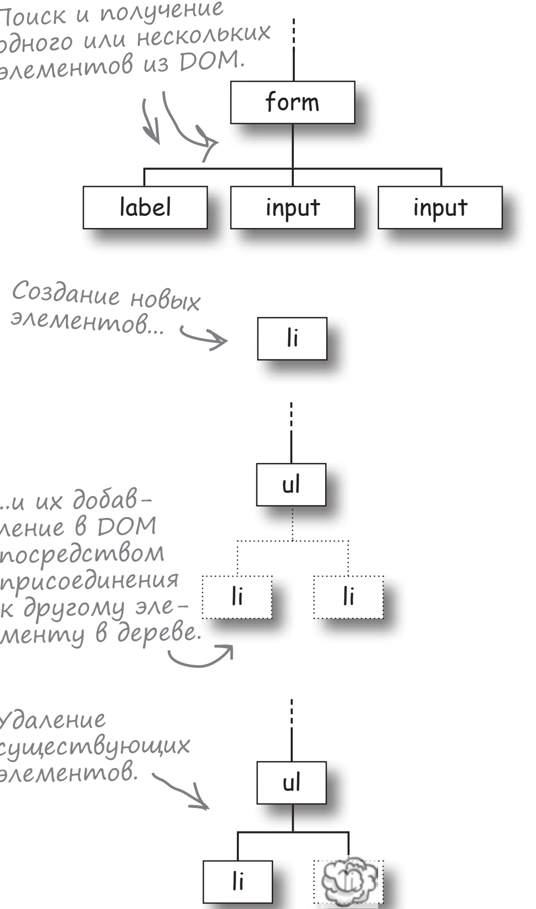
Последний тест-драйв...

Загрузите страницу в браузере. В абзаце с идентификатором «greenplanet» сообщение выводится ярко-красным шрифтом, так что пользователь наверняка обратит на него внимание!



Что еще можно сделать с моделью DOM?

Возможности модели DOM далеко не ограничиваются тем, что мы видели. Некоторые из них будут описаны позднее, а пока мы в общих чертах пройдемся по списку, чтобы вы хотя бы примерно представляли себе, что можно делать с DOM:



Получение элементов из DOM.

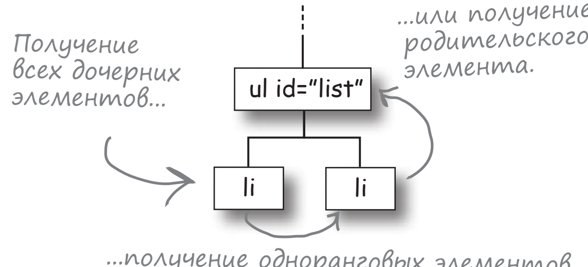
Конечно, вы уже знаете это, потому что мы использовали метод document.getElementById, но существуют и другие способы получения элементов; более того, имена тегов, имена классов и атрибуты могут использоваться для получения не только одного элемента, но целого набора (допустим, всех элементов с классом «on_sale»). Также вы можете получить значения, введенные пользователем на форме, — допустим, текст из элемента ввода.

Создание и добавление элементов в DOM.

Вы можете создавать новые элементы и добавлять их в DOM. Конечно, любые изменения, вносимые в DOM, немедленно отразятся на представлении DOM в браузере (что нам, собственно, и нужно!).

Удаление элементов из DOM.

Также из DOM можно удалять любые элементы: вы берете родительский элемент и удаляете любые из его потомков. И снова удаленный элемент исчезает из окна браузера сразу же, как только он будет удален из DOM.



Перебор элементов в DOM.

Получив доступ к элементу, вы сможете найти все его дочерние элементы, получить «соседей», то есть одноранговые элементы, а также узнать родителя. По своей структуре DOM сильно напоминает генеалогическое дерево.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Объектная модель документа**, или DOM, — внутреннее представлением веб-страницы в браузере.
- Браузер создает модель DOM страницы в процессе загрузки и разбора разметки HTML.
- Для получения доступа к DOM в коде JavaScript используется объект document.
- Объект document обладает свойствами и методами, которые могут использоваться для получения доступа и модификации DOM.
- Метод **document.getElementById** получает элемент из DOM по идентификатору.
- Метод document.getElementById method возвращает **объект element**, представляющий элемент страницы.
- Объект element обладает свойствами и методами, при помощи которых вы сможете прочитать содержимое элемента и изменить его.
- Свойство **innerHTML** содержит текстовое содержимое элемента, а также всю его вложенную разметку HTML.
- Чтобы изменить содержимое элемента, вы изменяете значение его свойства innerHTML.
- При изменении элемента (посредством изменения его свойства innerHTML) изменения немедленно отражаются на веб-странице.
- Метод **getAttribute** позволяет получать значения атрибутов элементов.
- Метод **setAttribute** позволяет задавать значения атрибутов элементов.
- Если ваш код размещается в элементе <script> в секции <head> страницы, проследите за тем, чтобы он не пытался изменять DOM до того, как страница будет полностью загружена.
- Свойство **onload** объекта window может использоваться для назначения **обработчика события** (или функции обратного вызова) для загрузки страницы.
- Обработчик события свойства onload объекта window вызывается сразу же после того, как страница будет полностью загружена.
- Существует много разных событий, которые могут обрабатываться из кода JavaScript при помощи функций-обработчиков.

Стань браузером
Решение

Представьте себя на месте браузера. Ваша задача — разобрать разметку HTML и построить для нее модель DOM. Используйте разметку, приведенную справа, и нарисуйте свою модель DOM внизу. Мы уже начали рисовать ее за вас.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Movies</title>
  </head>
  <body>
    <h1>Movie Showtimes</h1>
    <h2 id="movie1" >Plan 9 from Outer Space</h2>
    <p>Playing at 3:00pm, 7:00pm.
      <span>
        Special showing tonight at <em>midnight</em>!
      </span>
    </p>
    <h2 id="movie2">Forbidden Planet</h2>
    <p>Playing at 5:00pm, 9:00pm.</p>
  </body>
</html>
```

```
Movie Showtimes
Plan 9 from Outer Space
Playing at 3:00pm, 7:00pm. Special showing tonight at midnight!
Forbidden Planet
Playing at 5:00pm, 9:00pm.
```

Наша модель DOM.

- document
  - html
    - head
      - meta
      - title
    - body
      - h1
      - h2 id="movie1"
      - p
        - span
          - em
      - h2 id="movie2"
      - p

Возьми в руку карандаш Решение

Перед вами разметка HTML со списком песен — изначально этот список пуст. Ваша задача — дописать приведенный код JavaScript, чтобы он добавлял песни в список. Заполните пробелы программными конструкциями JavaScript, которые будут решать эту задачу. Ниже приведено наше решение.

Разметка HTML-страницы.

Это наш сценарий. Программа должна заполнять список песен ниже в элементе <ul>.

Заполните пропуски в коде так, чтобы он выводил список песен.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My Playlist</title>
  <script>
  function addSongs() {
    var song1 = document.getElementById("song1");
    var song2 = document.getElementById("song2");
    var song3 = document.getElementById("song3");

    song1.innerHTML = "Blue Suede Strings, by Elvis Pagely";
    song2.innerHTML = "Great Objects on Fire, by Jerry JSON Lewis";
    song3.innerHTML = "I Code the Line, by Johnny JavaScript";
  }
  window.onload = addSongs;
  </script>
</head>
<body>
  <h1>My awesome playlist</h1>
  <ul id="playlist">
    <li id="song1"></li>
    <li id="song2"></li>
    <li id="song3"></li>
  </ul>
</body>
</html>
```

Пустой список песен. Приведенный выше код должен добавлять текст в каждый из элементов <li> списка.

Так должна выглядеть веб-страница после загрузки, когда код JavaScript заработает.

My Playlist

http://localhost/~Beth/Head-

My awesome playlist

- Blue Suede Strings, by Elvis Pagely
- Great Objects on Fire, by Jerry JSON Lewis
- I Code the Line, by Johnny JavaScript

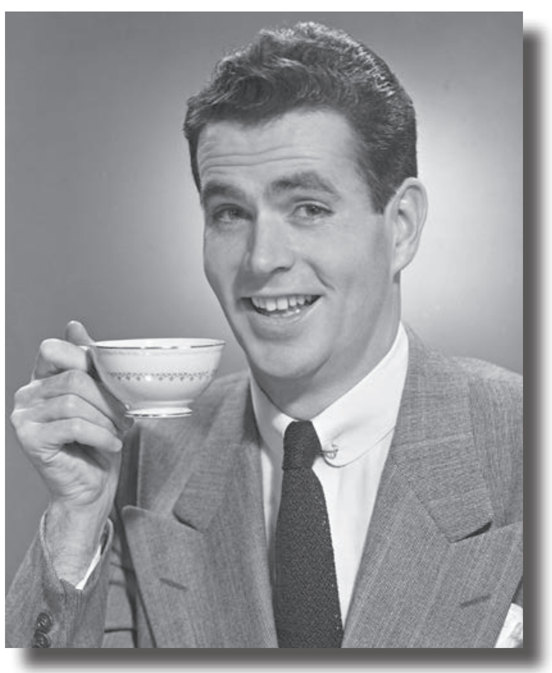
Возьми в руку карандаш
Решение

Нет ли желания поработать другой частью мозга? Нам понадобится стиль CSS для класса “redtext”, который назначает тексту абзаца красный цвет (“red”). Если вы справляетесь с такими задачами с закрытыми глазами — что ж, прекрасно. Если с тех пор, как вы писали CSS, прошло много времени, не огорчайтесь; все равно попробуйте это сделать. Ниже приведено наше решение.

```
.redtext { color: red; }
```

7 типы, равенство, преобразования и все такое

Серьезные типы



Настало время серьезно поговорить о типах. Одна из замечательных особенностей JavaScript заключается в том, что начинающий может достаточно далеко продвинуться, не углубляясь в подробности языка. Но чтобы действительно **овладеть языком**, получить повышение по работе и заняться тем, чем действительно стоит заниматься, нужно хорошо разбираться в **типах**. Помните, что мы говорили о JavaScript, — что у этого языка не было такой роскоши, как академическое определение, прошедшее экспертную оценку? Да, это правда, но отсутствие академической основы не помешало Стиву Джобсу и Биллу Гейтсу; не помешало оно и JavaScript. Это означает, что система типов JavaScript... ну, скажем так — не самая продуманная, и в ней найдется немало **странностей**. Но не беспокойтесь, в этой главе мы все разберем, и вскоре вы научитесь благополучно обходить все эти неприятные моменты с типами.

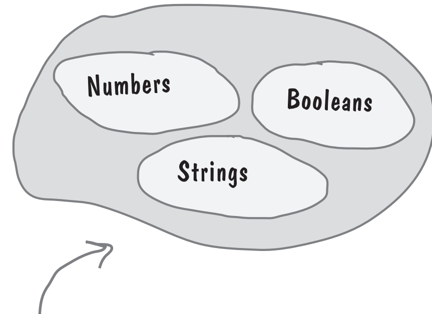
Истина где-то рядом...

У вас уже есть немалый опыт работы с типами JavaScript — это и примитивы с числами, строками и булевскими значениями, и многочисленные объекты, одни из которых предоставляет JavaScript (как, например, объект Math), другие предоставляет браузер (как объект document), а третьи вы написали самостоятельно. Казалось бы, дальше можно просто нежиться под теплым сиянием простой, мощной и последовательной системы типов JavaScript.

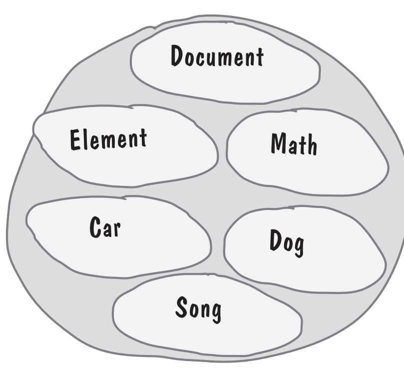
Низкоуровневые базовые типы для чисел, строк и булевских значений.

Высокоуровневые объекты, используемые для представления сущностей из пространства задачи.

Примитивные типы



Объекты



Предоставляются JavaScript.

JavaScript предоставляет большой набор полезных объектов, но вы также можете создавать собственные объекты или использовать объекты, написанные другими разработчиками.

В конце концов, чего вы ожидали от официального языка веб-программирования? Будь вы простым сценарным программистом, вы могли бы откинуться в кресле, потягивая мартини, и наслаждаться желанным отдыхом...

Но вы не простой сценарный программист, и вы чувствуете, что здесь что-то не так. У вас возникает тревожное чувство, что за заборами Объектвиля происходит что-то странное. До вас доносились слухи о строках, которые ведут себя как объекты; вы читали в блогах о типа null (возможно, небезопасном для здоровья); люди шепотом говорят, что интерпретатор JavaScript в последнее время выполняет какие-то странные преобразования типов. Что все это значит? Мы не знаем, но истина где-то рядом, и мы откроем ее в этой главе. А когда это произойдет, ваши представления о том, что есть истина, а что есть ложь, могут резко измениться.

Группа значений JavaScript и незваных гостей, облачившись в маскарадные костюмы, развлекаются игрой «Кто я?». Они дают подсказки, а вы должны угадать их по тому, что они говорят о себе. Предполагается, что участники всегда говорят о себе правду. Соедините линией каждое высказывание с именем соответствующего участника. Мы уже провели одну линию за вас. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

Если упражнение покажется вам слишком сложным — не отчаивайтесь и подсмотрите ответ.

Я возвращаюсь из функции при отсутствии команды return.

Я считаюсь значением переменной, которой еще не было присвоено значение.

Я — значение элемента, не существующего в разреженном массиве. → undefined

Я — значение несуществующего свойства.

Я — значение удаленного свойства.

Я — значение, которое не может быть задано свойству при создании объекта.

Сегодняшние участники:

нуль

пустой объект

null

undefined

NaN

infinity

area 51

...---...

{}

[]

Будьте осторожны: undefined иногда появляется совершенно неожиданно...

Каждый раз, когда что-то идет не так, вы используете переменную, которая еще не была инициализирована, пытаетесь получить значение несуществующего (или удаленного) свойства, обращаетесь к отсутствующему элементу массива — вы сталкиваетесь с `undefined`.

Что это такое, спросите вы? Ничего сложного. Считайте, что `undefined` — это значение, которое присваивается тому, что еще не имеет значения (другими словами, не было инициализировано).

Какая польза от undefined? Оно позволяет проверить, было ли присвоено значение переменной (или свойству, или элементу массива). Рассмотрим пару примеров начиная с неинициализированной переменной.

```
var x;

if (x == undefined) {
    // Переменная x не инициализирована! Принять меры!
}
```

Мы можем проверить, была ли инициализирована переменная. Просто сравните ее с undefined.

Обратите внимание: здесь используется значение undefined. Не путайте его со строкой "undefined".

Теперь проверяем свойство объекта:

```
var customer = {
    name: "Jenny"
};
if (customer.phoneNumber == undefined) {
    // Получить телефон клиента
}
```

Чтобы проверить свойство на неопределенность, также сравните его со значением undefined.

Часто задаваемые вопросы

В: Когда нужно проверять переменную (свойство, элемент массива) на неопределенность?

О: Это зависит от структуры кода. Если ваш код написан так, что свойство или переменная может не иметь значения при выполнении некоторого блока, то проверка на undefined даст вам возможность обработать эту ситуацию вместо того, чтобы выполнять вычисления с неопределенными значениями.

В: Если undefined — это значение, то есть ли у него тип?

О: Да, есть — значение undefined относится к типу Undefined. Почему? Вероятно, логика выглядит примерно так: это не объект, не число, не строка и не булевское значение... И вообще ничто определенное. Так почему бы не назначить этой неопределенности «неопределенный» тип? Это одна из тех странностей JavaScript, с которыми нужно просто смириться.

В лаборатории

В своей лаборатории мы любим разбирать, заглядывать «под капот», экспериментировать, подключать диагностические инструменты и выяснять, что же происходит на самом деле. Сегодня, в ходе исследования системы типов JavaScript, мы обнаружили маленький диагностический инструмент для анализа переменных — **typeof**. Наденьте лабораторный халат и защитные очки, заходите и присоединяйтесь к нам.

types

Оператор **typeof** встроен в JavaScript. Он используется для проверки типа операнда (того, к чему применяется оператор). Пример:

```
var subject = "Just a string";

var probe = typeof subject;
console.log(probe);
```

Оператор typeof получает операнд и определяет его тип.

В данном случае выводится тип string. Обратите внимание: для представления типов typeof использует строки вида "string", "boolean", "number", "object", "undefined" и т. д.

Консоль JavaScript

```
string
```

Теперь ваша очередь. Соберите данные по следующим экспериментам:

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Тестовые данные и тесты.

Консоль JavaScript

Запишите здесь результаты. Какие-нибудь сюрпризы?

Да, здесь часто возникает путаница. Во многих языках существует концепция значения, представляющего «отсутствие объекта». И это вполне нормально — возьмем хотя бы метод `document.getElementById`. Он ведь должен возвращать объект, верно? А что произойдет, если он не сможет вернуть объект? Тогда он должен вернуть какой-то признак, означающий: «Здесь мог бы быть объект, но, к сожалению, его нет». Именно этот смысл заложен в `null`.

Переменной также можно явно присвоить `null`:

```
var killerObjectSomeday = null;
```

Что может означать присваивание `null` переменной? Например, «Когда-нибудь в будущем мы присвоим объект этой переменной, но пока еще не присвоили».

Если в этот момент вы недоумеваете и спрашиваете: «Хмм, а почему они не использовали `undefined`?» — знайте, что вы не одни. Это решение было принято в самом начале существования JavaScript. Разработчики хотели иметь одно значение для переменных, которые еще не были инициализированы, и другое для обозначения отсутствия объекта. Может, решение не самое красивое и немного избыточное, но дело обстоит именно так. Просто запомните смысл каждого из значений (`undefined` и `null`) и знайте, что null чаще всего используется там, где объект еще не создан или не найден, а `undefined` — для неинициализированных переменных, отсутствующих свойств объектов или отсутствующих значений в массивах.

Снова в лабораторию

Стоп, мы забыли включить null в свои тестовые данные. Недостающий тест:

```
var test10 = null;

console.log(typeof test10);
```

Запишите здесь свой результат.

Консоль JavaScript

Как использовать null

Существует великое множество функций и методов, возвращающих объекты. Часто бывает нужно убедиться в том, что вы получаете полноценный объект, а не `null`, — на случай, если функция не смогла найти или создать возвращаемый объект. Примеры уже встречались нам при работе с DOM:

```
var header = document.getElementById("header");
if (header == null) {
    // Заголовка нет — какая-то серьезная проблема!
}
```

Ищем элемент header, без которого никак не обойтись.

Ну и ну, заголовок не существует! Спасайся, кто может!

Следует учитывать, что получение `null` не всегда означает, что что-то пошло не так. Это может означать, что объект еще не существует и его нужно создать или его нужно пропустить при обработке. Пользователь может открыть или закрыть погодный виджет на сайте. Если виджет открыт, то в DOM существует элемент `<div>` с идентификатором "weatherDiv", а если закрыт — такого элемента нет. Так польза от `null` становится очевидной:

Проверим, существует ли элемент с идентификатором "weatherDiv".

```
var weather = document.getElementById("weatherDiv");
if (weather != null) {
    // Создание содержимого для погодного виджета
}
```

При помощи null можно проверить, существует объект или нет.

Если результат getElementById отличен от null, значит, на странице существует такой элемент. Создадим для него погодный виджет (который, как предполагается, загружает местную сводку погоды).

Блейн, штат Миссури

Всегда жарко, вероятность дождя 40%.

Помните: значение null используется для представления несуществующих объектов.

Хотите верьте, хотите нет!

Число которое не является числом

Хотите верьте, хотите нет, но существуют числовые значения, которые **невозможно представить** в JavaScript! В JavaScript эти значения не выражаются, поэтому для них используется специальное значение:

NaN

Легко написать команду JavaScript, результатом которой является неопределенное числовое значение.

Несколько примеров:

```
var a = 0/0;
```

В математике это выражение не имеет однозначного результата — так откуда его возьмет JavaScript?

```
var b = "food" * 1000;
```

Мы не знаем, как должен выглядеть результат, но это безусловно не число!

```
var c = Math.sqrt(-9);
```

Квадратный корень из отрицательного числа — это комплексное число, а в JavaScript такие числа не имеют представления.

В JavaScript используется значение NaN (сокращение от "Not a Number", то есть «не число») для представления числовых результатов... не имеющих представления. Для примера возьмем 0/0. Результат 0/0 не имеет собственного представления на компьютере, поэтому в JavaScript он представляется специальным значением NaN.

ВОЗМОЖНО, NaN — САМОЕ СТРАННОЕ ЗНАЧЕНИЕ В МИРЕ. Оно не только представляет все числовые значения, не имеющие собственного представления; это единственное значение в JavaScript, не равное самому себе!

Да, вы поняли правильно. Если сравнить NaN с NaN, они не будут равны!

NaN != NaN

Работа с NaN

Может показаться, что вам очень редко придется иметь дело со значением NaN, но если ваш код обрабатывает числовые данные, вы удивитесь, как часто оно будет встречаться на вашем пути. Вероятно, самой частой операцией будет проверка числа на NaN. С учетом того, что вы узнали о JavaScript, решение может показаться очевидным:

```
if (myNum == NaN) {
    myNum = 0;
}
```

Вроде бы должно работать... Но не работает.

Логично предположить, что проверять переменную на хранение значения NaN нужно именно так... Но это решение не работает. Почему? Потому что значение NaN не равно ничему, даже самому себе, значит, любые проверки равенства с NaN исключаются. Вместо этого приходится использовать специальную функцию isNaN. Это делается так:

```
if (isNaN(myNum)) {
    myNum = 0;
}
```

Функция isNaN возвращает true, если переданное ей значение не является числом.

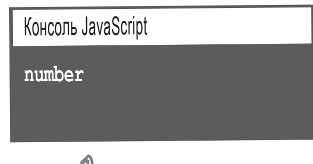
ПРАВИЛЬНО!

А дальше еще удивительнее

Итак, давайте немного подумаем. Если NaN означает «не число», то что это? Разве не проще было бы выбрать имя по тому, чем оно является (а не по тому, чем оно не является)? А как вы думаете, чем оно является? Тип NaN можно проверить оператором typeof с совершенно неожиданным удивительным результатом:

```
var test11 = 0 / 0;
console.log(typeof test11);
```

Вот что мы получаем.



Если у вас голова не идет кругом, вероятно, вы чего-то не поняли из этой книги.

Что вообще творится? NaN имеет числовой тип? Как то, что не является числом, может иметь числовой тип? Спокойно, дышите глубже. Считайте, что имя NaN просто выбрано неудачно. Вероятно, вместо «не число» стоило использовать что-то вроде «число, не имеющее представления» (хотя, конечно, красивого сокращения уже не получится). Лучше всего рассматривать NaN именно так — как число, которое невозможно представить (по крайней мере на компьютере).

В общем, ваш список странностей JavaScript расширяется.

Часто задаваемые вопросы

В: Если передать isNaN строку, которая не является числом, вернет ли функция true?

О: Несомненно, как и следовало ожидать. Для переменной, содержащей значение NaN или любое другое значение, не являющееся числом, функция isNaN вернет true (а в противном случае возвращается значение false). У этого правила есть несколько неочевидных аспектов; мы рассмотрим их, когда займемся преобразованиями типов.

В: Но почему значение NaN не равно само себе?

О: Если вы хотите глубоко разобраться в этом вопросе, обратитесь к спецификации вычислений с плавающей запятой IEEE. Упрощенно говоря, тот факт, что NaN представляет непредставимое число, не означает, что два таких непредставимых числа равны. Например, возьмем sqrt(-1) и sqrt(-2): результаты явно не равны, но в обоих случаях используется значение NaN.

В: Деление 0/0 дает результат NaN, но я попытался разделить 10/0 и получил `Infinity`. Чем это значение отличается от `NaN`?

О: Значение The Infinity (или -Infinity) в JavaScript представляет все значения, выходящие за границы представления чисел с плавающей точкой, то есть 1.7976931348623157E+10308 (или -1.7976931348623157E+10308 для -Infinity). Значение Infinity имеет числовой тип. Используйте проверку на Infinity, если подозреваете, что ваше значение оказалось слишком большим:

```
if (tamale == Infinity) {
    alert("That's a big tamale!");
}
```

В: Я в шоке от этого “NaN имеет числовой тип”. Что-нибудь еще интересное в этом роде?

О: Странно, что вы спрашиваете. Хорошо, вот вам: Infinity минус Infinity равно... вы не поверите... NaN. Чтобы понять это, вам лучше обратиться к хорошему математику.

В: Просто для полноты картины: а к какому типу относится null?

О: Это очень легко узнать: примените оператор typeof к null. Вы получите “object”. В принципе это разумно, ведь null используется для представления отсутствующих объектов. Однако в последнее время по этому поводу шли яростные споры, и в последней спецификации null определяется с типом null. Возможно, в этом отношении реализация JavaScript вашего браузера будет расходиться со спецификацией, но на практике вам вряд ли потребуется использовать тип null в программном коде.

Упражнение

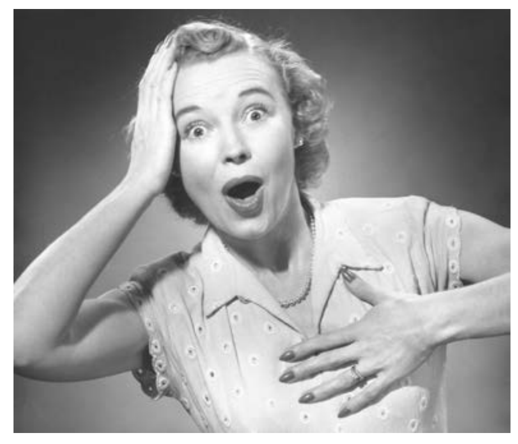
В этой главе мы рассматривали некоторые... ммм... интересные значения. А теперь рассмотрим интересное поведение. Попробуйте добавить приведенный ниже код под элементом <script> обычной веб-страницы и посмотрите, что будет выведено на консоль при загрузке страницы. Причины вам пока неизвестны, но попробуйте хотя бы предположить, что при этом происходит.

```
if (99 == "99") {
    console.log("A number equals a string!");
} else {
    console.log("No way a number equals a string");
}
```

Консоль JavaScript

Здесь запишите результат.

Мы должны вам признаться...



Есть один аспект JavaScript, о котором мы намеренно не упоминали. В принципе о нем можно было сказать сразу, но тогда он бы не выглядел так логично, как сейчас.

Не думайте, будто мы намеренно вводили вас в заблуждение — скорее, придержали часть информации. О чем идет речь? Взгляните сами:

Переменной присваивается значение — в данном случае число 99.

```
var testMe = 99;
```

Позднее оно сравнивается с числом при проверке условия.

```
if (testMe == 99) {
    // Происходит что-то хорошее
}
```

Тривиально, верно? Еще бы, что может быть проще? Однако возможна и другая ситуация — более того, мы уже проделали нечто подобное минимум один раз, хотя вы, возможно, и не заметили.

Переменной присваивается значение, в данном случае строка "99".

Вы заметили, что на этот раз используется строка?

```
var testMe = "99";
```

Позднее она сравнивается с числом в условии.

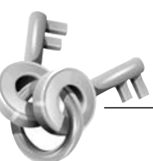
```
if (testMe == 99) {
    // Происходит что-то
хорошее
}
```

На этот раз строка сравнивается с числом.

Что же произойдет при сравнении строки с числом? Вселенский хаос? Цепная реакция в компьютере? Беспорядки на улицах?

Нет, JavaScript хватает ума понять, что в практическом контексте 99 и "99" почти не отличаются друг от друга. Но что именно происходит «за кулисами», чтобы такое сравнение работало? Давайте посмотрим.

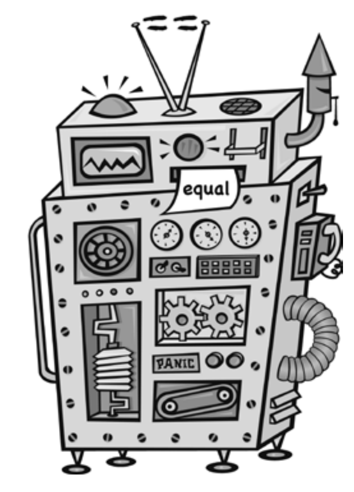
КЛЮЧЕВЫЕ МОМЕНТЫ



Краткое напоминание о том, чем присваивание отличается от проверки равенства:

- `var x = 99;`
  = — это оператор присваивания. Он используется для изменения значения переменной.
- `x == 99`
  == — это оператор проверки равенства. Он сравнивает одно значение с другим и выясняет, равны ли они между собой.

Оператор проверки равенства (также известный как ==)

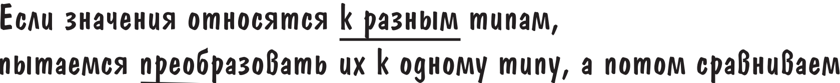


Казалось бы, равенство — очень простая концепция. Мы понимаем, что 1 == 1, "guacamole" == "guacamole", а true == true. Но в случае "99" == 99 дело этим не ограничивается. Что может происходить в операторе проверки равенства, чтобы такие сравнения стали возможными?

Оказывается, оператор == при сравнении учитывает типы своих операндов (то есть двух значений, которые вы сравниваете). Возможны два варианта:

Если два значения относятся к одному типу, просто сравниваем их

Если два сравниваемых значения имеют одинаковый тип (скажем, два числа или две строки), то сравнение работает как обычно: два значения просто сравниваются друг с другом. Если значения одинаковые, то результат равен true. Все просто.



Если значения относятся к разным типам, пытаемся преобразовать их к одному типу, а потом сравниваем

Это более интересный случай. Допустим, сравниваются два значения с разными типами, например число и строка. JavaScript пытается преобразовать строку в число, после чего сравнивает два значения:

Преобразование действует временно, только для выполнения сравнения.

```
99 == "99"
   ↓
99 == 99
```

Когда вы сравниваете число со строкой, JavaScript преобразует строку в число (если это возможно)...

... а затем пытается выполнить сравнение повторно. Теперь, если значения равны, то результат будет true, а если нет — false.

На уровне здравого смысла выглядит разумно, но какие правила действуют при преобразовании? Что будет, если сравнить булевское значение с числом, null с undefined или с другими комбинациями значений? Как узнать, что и во что преобразуется? И почему бы не преобразовать число в строку или воспользоваться другой схемой для проверки равенства? Относительно простой набор правил в спецификации JavaScript определяет, как должно выполняться преобразование при сравнении двух значений разных типов. Это одна из тех областей, которые необходимо предельно твердо усвоить, но зато после этого в вашей карьере программиста JavaScript уже никогда не будет проблем со сравнениями.

Кроме того, вы будете выгодно выделяться на фоне своих коллег и вам будет проще на следующем собеседовании.

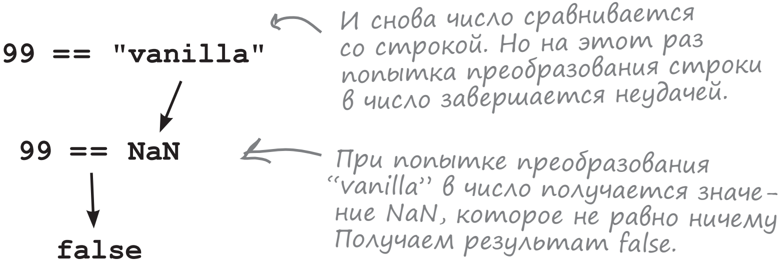
Как происходит преобразование операндов (все не так страшно, как может показаться)

Итак, мы знаем, что при сравнении двух значений разных типов JavaScript преобразует один тип к другому, чтобы сравнить их. Если у вас есть опыт программирования на других языках, это может показаться странным: обычно такие операции программируются явно, а не происходят автоматически. Не волнуйтесь, обычно эта особенность JavaScript полезна — *при условии, что вы понимаете, как и где она работает*. Именно это нам и нужно выяснить: где выполняются преобразования и как они выполняются.

Рассмотрим четыре простых случая.

СЛУЧАЙ №1: Сравнение числа со строкой.

При сравнении строки с числом всегда происходит одно и то же: строка преобразуется в число, после чего сравниваются два числа. Преобразование не всегда проходит гладко, потому что не все строки можно преобразовать в числа. Давайте посмотрим, что происходит в этой ситуации:



СЛУЧАЙ №2: Сравнение булевского значения с любым другим типом.

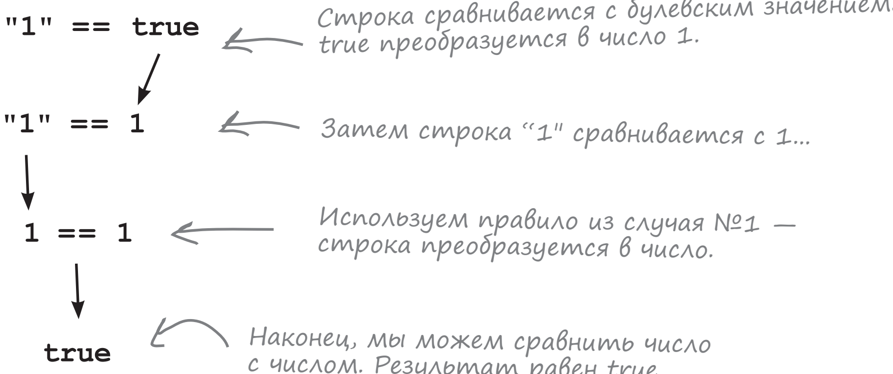
В этом случае булевское значение преобразуется в число и два числа сравниваются. Выглядит немного странно, но происходящее становится более понятным, если просто запомнить, что true преобразуется в 1, а false преобразуется в 0. Также стоит помнить, что данная ситуация не всегда ограничивается одним преобразованием типа. Рассмотрим несколько примеров:

```
1 == true
1 == 1
true
```

Сравниваем число с булевским значением. Значение true преобразуется в число 1.

1 сравнивается с 1, получаем true.

Другой пример: на этот раз булевское значение сравнивается со строкой. В данном случае потребуется больше шагов.



СЛУЧАЙ №3: Сравнение null с undefined.

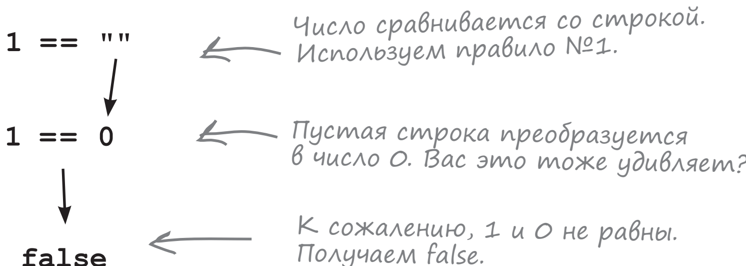
Сравнение этих значений дает результат true. Выглядит немного странно, но таковы правила. Чтобы происходящее стало более понятным, эти значения фактически представляют «отсутствие значения» (то есть переменная или объект, не имеющие значения), поэтому они считаются равными.

```
undefined == null
      ↓
     true
```

Undefined и null всегда равны.

СЛУЧАЙ №4: ...Постойте, а случая №4 нет.

Вот и все. По этим правилам можно определить результат практически любой проверки равенства. Конечно, граничные случаи и особые ситуации временами встречаются. Во-первых, нужно разобраться со сравнением объектов; мы сделаем это немного позднее. Во-вторых, некоторые преобразования могут застать вас врасплох. Один из примеров:



Ах, если бы два значения можно было проверить на **равенство,** не беспокоясь о преобразовании типов. Чтобы два значения считались равными только в том случае, если они содержат одинаковые значения *и* относятся к одному типу. Чтобы не нужно было помнить все эти правила и ошибки, которые могут из-за них возникнуть. Это было бы замечательно... Но это, конечно, только мечты.

Как проверить строгое равенство

Я к своим сравнениям отношусь гораздо строже.

Раз уж мы начали признаваться, откроем еще одну тайну: в JavaScript не один, а два оператора проверки равенства. Вы уже видели оператор == (равенство), но еще существует оператор === (строгое равенство).

Все верно, три знака =. Вы всегда можете использовать === вместо ==, но сначала нужно убедиться в том, что вы понимаете, чем они отличаются.

Сложные правила, связанные с преобразованием (разнотипных) операндов для оператора ==, вам уже известны. У оператора === правила еще сложнее.

...Шутка. Вообще-то для === правило *всего одно*:

От редактора: Дуг Крокфорд, один из создателей JavaScript, разрешил нам использовать свою фотку.

Два значения считаются строго равными только в том случае, если они относятся к одному типу и содержат одно значение.

Перечитайте еще раз. Это означает, что если два значения относятся к одному типу, мы переходим к их сравнению. Если нет — результат равен false в любом случае: никаких преобразований, никаких сложных правил, ничего. Вам нужно лишь запомнить, что с оператором === два значения считаются равными, *только если они относятся к одному типу и содержат одинаковые значения*.

Возьми в руку карандаш

Для каждого сравнения запишите его результат (true или false) при использовании операторов == и ===:

	==	===	
`"42" == 42`	true	______	`"42" === 42`
`"0" == 0`	______	______	`"0" === 0`
`"0" == false`	______	______	`"0" === false`
`"true" == true`	______	______	`"true" === true`
`true == (1 == "1")`	______	______	`true === (1 === "1")`

Трудный вопрос! Трудный вопрос!

В: Что произойдет, если сравнить число (например, 99) со строкой, которая не преобразуется в число (“ninety-nine”)?

О: JavaScript попытается преобразовать “ninety-nine” в число. Попытка окажется неудачной, получится NaN. Два значения окажутся неравными, и результат будет false.

В: Как JavaScript преобразует строки в числа?

О: Строка последовательно разбирается на символы, и каждый символ преобразуется в компонент числа. Таким образом, для строки “34” сначала берется символ “3”, который преобразуется в цифру 3, затем символ “4” преобразуется в 4. Строки вида “1.2” преобразуются в вещественные числа: JavaScript понимает, что такая строка может интерпретироваться как число.

В: А если я попробую выполнить проверку “true” == true?

О: Здесь строка сравнивается с булевским значением, поэтому по правилам JavaScript сначала преобразует true в 1, а затем сравнит “true” с 1. Для этого JavaScript попытается преобразовать “true” в число, попытка окажется неудачной, и вы получите false.

В: Раз существуют операторы == и ===, доступны ли аналогичные операторы <=, <==, >= и >==?

О: Нет. Операторы <== и >== не существуют, можно использовать только <= и >=. Эти операторы умеют сравнивать только строки и числа (true <= false не имеет смысла), поэтому при сравнении любых значений, кроме двух строк или двух чисел (или строки с числом), JavaScript пытается выполнить преобразование типов по описанным ранее правилам.

В: И как же будет вычисляться результат 99 <= “100”?

О: Действуйте по правилам: “100” преобразуется в число, после чего сравнивается с 99. Так как 99 меньше либо равно 100 (в данном случае меньше), мы получаем true.

В: Существует ли оператор !==?

О: Да. И по аналогии с тем, как === является «строгой» версией ==, оператор !== является «строгой» версией !=. Оператор !== использует те же правила, что и ===, но проверяет неравенство вместо равенства.

В: Будут ли те же правила использоваться при сравнении, допустим, булевского значения и числа операторами < и >, например 0 < true?

О: Да! И в этом случае true преобразуется в 1, и вы получаете true, ведь 0 меньше 1.

В: Строка может быть равна другой строке, это понятно. Но как строка может быть меньше или больше другой строки?

О: Хороший вопрос. Как определить результат сравнения “banana” < “mango”? Чтобы определить, будет ли одна строка меньше или больше другой, можно воспользоваться алфавитным порядком. Так как строка “banana” начинается с “b”, а “mango” начинается с “m”, “banana” меньше “mango”, потому что “b” предшгествует “m” в алфавите. А строка “mango” меньше “melon”, потому что первые буквы совпадают, но при сравнении вторых букв “a” предшествует “e”.

Впрочем, алфавитное сравнение порой преподносит неожиданные сюрпризы; например, условие “Mango” < “mango” истинно, хотя, казалось бы, буква “M” в верхнем регистре должна быть больше “m” в нижнем регистре. Порядок сортировки строк определяется порядком значений Юникода, используемых для представления каждого символа на компьютере (Юникод — стандарт цифрового представления символов), а этот порядок может отличаться от ваших ожиданий. За подробностями обращайтесь к Google. Тем не менее в большинстве случаев отношения «больше/меньше» между строками определяются простым алфавитным порядком.

Беседа у камина

Сегодня в студии: операторы равенства и строгого равенства выясняют, кто главнее.

==

Смотрите, кто пришел — мистер Бескомпромиссность.

===

Учти, что, по мнению некоторых ведущих экспертов JavaScript, разработчикам стоит использовать меня, и только меня. Они считают, что тебя лучше вообще убрать из языка.

==

А ты сравни, как часто встречаются == и === в коде по всему миру. Боюсь, ты сильно отстаешь в популярности. Даже сравнивать смешно.

===

Возможно, ты прав, но люди постепенно начинают вникать в суть дела, и цифры меняются.

==

Вряд ли. Я предоставляю ценные услуги. Кому из нас не приходится, скажем, сравнивать строку, введенную пользователем, с числом?

===

Но к полезной функциональности прилагаются все правила, которые необходимо помнить просто для использования ==. Не усложняйте код и жизнь; используйте ===. А если вам потребуется преобразовать ввод пользователя в число, для этого существуют свои методы.

==

Ага, а еще ты каждый день ходил в начальную школу пешком, по снегу и в гору — притом в обе стороны... Сам подумай, зачем усложнять себе жизнь на ровном месте?

===

Очень смешно. Нет ничего плохого в четкой и строгой семантике сравнений. Иначе приходится постоянно держать в голове правила, и если вдруг забудешь о них — происходят самые неожиданные и неприятные вещи.

==

Короче, я не только выполняю те же преобразования, что и ты, но еще и добавляю к ним полезные преобразования типов.

===

Каждый раз, когда я смотрю на твои правила, меня начинает подташнивать. Чтобы сравнить булевское значение с чем угодно, необходимо сначала преобразовать его в число? Где логика?

==

А ты что будешь делать? Сразу сдашься, вернешь false и отправишься домой?

==

Пока все работает. Посмотри, сколько кода уже написано — причем часто его пишут программисты, как бы это сказать... рядового уровня.

В смысле — принять душ после одной из бесед с тобой?

Хмм... А откупиться не думал? Я охотно проведу свою жизнь на пляже, нежась на солнышке с «Маргаритой» в руке.

Все эти споры «== versus ===» надоели! В жизни есть и более интересные занятия.

Слушай, просто признай: люди не перестанут использовать ==. Иногда это очень удобно. И его можно использовать разумно, извлекая пользу из его особенностей там, где это имеет смысл. Как в примере с обработкой пользовательского ввода — почему бы не использовать ==?

А я считаю так: если разработчики хотят использовать тебя — прекрасно. А я остаюсь на своем месте на случай, если буду им нужен... В мире существует еще достаточно старого кода с == — и я всегда буду востребован.

===

Нет, но в твоих сложных правилах легко запутаться.

Это хорошо, но страницы становятся все более сложными и запутанными. Не пора ли поступать так, как рекомендуют профессионалы?

Нет, ограничиться использованием to ===. Код становится более понятным, исчезает риск неожиданных граничных ситуаций.

Неожиданно — я думал, ты будешь защищать свою роль НАСТОЯЩЕГО оператора проверки равенства до самого конца. Что происходит?

Даже не знаю, что сказать.

Я же говорил — никогда не знаешь, что может произойти.

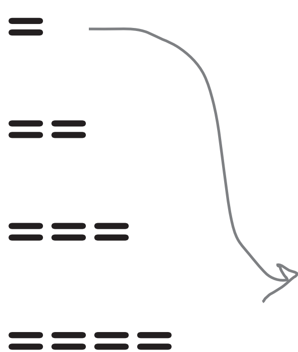
Отличично! Если кому-то не хватало проблем — пожалуйста, два оператора равенства. Какой из них использовать?

Спокойнее. По этой теме идут яростные споры, и разные эксперты высказывают разные мнения. Вот что мы думаем об этом: традиционно программисты использовали в основном == (равенство)... просто потому, что мало кто знал о двух операторах и различиях. В наши дни народ пошел более просвещенный, для большинства практических целей оператор === работает достаточно хорошо, к тому же он более безопасен, потому что вы точно знаете, что получите. Конечно, с оператором == результат тоже точно просчитывается, но со всеми преобразованиями иногда бывает трудно учесть все возможности.

В некоторых ситуациях оператор == удобен (например, при сравнении чисел со строками), и разумеется, никто не запрещает вам использовать == в таких случаях — особенно, если вы, в отличие от многих программистов, точно знаете, как работает ==. Теперь, когда мы рассмотрели ===, в этой книге мы поменяем курс и будем в основном использовать ===. Тем не менее мы не будем фанатично цепляться за этот выбор в тех случаях, когда == упрощает нашу жизнь и не создает проблем.

Разработчики также называют оператор === (строгое равенство) оператором «тождественности».

Мы подобрали описания для всех операторов, но они как-то перепутались. Помогите нам разобраться, кто что делает. Будьте внимательны: мы не уверены в том, сколько возможных описаний может иметь каждый претендент — нуль, одно или более. Один вариант мы уже нашли, он обозначен внизу:



Сравнивает значения и проверяет их на равенство. Он настолько внимателен, что пытается выполнить преобразование типов, чтобы узнать, действительно ли значения равны.

Сравнивает значения и проверяет их на равенство. При этом значения, относящиеся к разным типам, попросту не рассматриваются.

Присваивает значение переменной.

Сравнивает ссылки на объекты и возвращает true, если они равны, и false в противном случае.

Еще больше преобразований типов...

Условные команды — не единственная область, в которой вы столкнетесь с преобразованиями типов. Есть еще несколько операторов, которые любят преобразовывать типы, если им представится такая возможность. Обычно такие преобразования упрощают работу программиста, но желательно точно понимать, где и когда они выполняются. Давайте разберемся.

Снова о конкатенации и суммировании

Вероятно, вы уже поняли, что с числами оператор + выполняет сложение, а со строками — конкатенацию. Но что произойдет, если операнды + относятся к разным типам?

При попытке сложения числа со строкой JavaScript преобразует число в строку и выполняет конкатенацию (поведение противоположно тому, что происходит при проверке равенства):

```
var addi = 3 + "4";
```

При суммировании строки с числом выполняется конкатенация, а не суммирование.

Переменной result присваивается строка "34" (не 7).

```
var plusi = "4" + 3;
```

То же самое... Получается "43".

Если на первом месте стоит строка, а к ней оператором + прибавляется число, то происходит то же самое: число преобразуется в строку, и операнды объединяются посредством конкатенации.

А что с другими арифметическими операторами?

Что касается других арифметических операторов — умножения, деления и вычитания, — JavaScript рассматривает их как арифметические, а не строковые операции.

```
var multi = 3 * "4";
```

Здесь JavaScript преобразует строку "4" в число 4, и умножает его на 3, получается 12.

```
var divi = 80 / "10";
```

Строка "10" преобразуется в число 10. Затем 80 делится на число 10, получается 8.

```
var mini = "10" — 5;
```

С вычитанием "10" преобразуется в число 10; получается 10 минус 5, то есть 5.

Часто задаваемые вопросы

В: Оператор + всегда интерпретируется как конкатенация, если один из операндов является строкой?

О: Да. Но поскольку оператор + обладает так называемой левосторонней ассоциативностью, в следующей ситуации:

```
var order = 1 + 2 + " pizzas";
```

вы получите строку “3 pizzas”, а не "12 pizzas", потому что при обработке выражения слева направо сначала 1 прибавляется к 2 (оба значения являются числами); получается 3. Затем число 3 суммируется со строкой, поэтому 3 преобразуется в строку и объединяется с "pizza" посредством конкатенации. Чтобы избежать всякой неоднозначности, вы всегда можете использовать круглые скобки для обеспечения нужного порядка выполнения операторов:

```
var order = (1 + 2) + " pizzas";
```

В этом случае получается строка “3 pizzas”, а в этом:

```
var order = 1 + (2 + " pizzas");
```

результатом будет строка “12 pizzas”.

В: Это все? Или бывают другие преобразования?

О: Есть и другие области, в которых происходят преобразования. Например, унарный оператор «–» (для создания отрицательных чисел) преобразует –true в –1. А при конкатенации булевского значения со строкой создается строка (скажем, true + “ love” дает “true love”). Эти случаи относительно редки, и мы никогда не сталкивались с ними на практике, но знать об их существовании все же стоит.

В: А если я хочу, чтобы JavaScript преобразовал строку в число и сложил ее с другим числом, как это сделать?

О: Для этого существует функция с именем Number (да, символ N в верхнем регистре). Она используется примерно так:

```
var num = 3 + Number("4");
```

В результате переменной num присваивается значение 7. Функция Number получает аргумент, и если возможно, преобразует его в число. Если преобразовать аргумент в число не удается, Number возвращает... вы не поверите... NaN.

Возьми в руку карандаш

Пришло время проверить, насколько вы понимаете преобразования. Запишите результат каждого выражения справа от него. Одно задание мы уже выполнили за вас. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
Infinity - "1"                        ____________
"42" + 42                             “4242”
2 + "1 1"                             ____________
99 + 101                              ____________
"1" - "1"                             ____________
console.log("Result: " + 10/2)        ____________
3 + " bananas " + 2 + " apples"       ____________
```

Но вы еще ни слова не сказали о том, как равенство работает для объектов. И вообще, какие два объекта можно считать равными?

Хорошо, что вы подумали об этом. Что касается равенства объектов, есть два ответа: один простой, а другой длинный и глубокий. Простой ответ относится к вопросу: равен ли этот объект тому объекту? Иначе говоря, если у меня есть две переменные, содержащие ссылки на объекты, ссылаются ли они на один объект? Эта тема будет рассмотрена на следующей странице. Сложный вопрос связан с типами объектов и тем, принадлежат ли два объекта к одному типу. Мы уже видели, что вы можете создавать объекты, которые выглядят как однотипные, но как удостовериться, что это действительно так? На этот важный вопрос мы попытаемся ответить в следующей главе.

Как проверить два объекта на равенство

Первый вопрос: о каком сравнении мы говорим, == или ===? Хорошая новость: при сравнении двух объектов это неважно! Иначе говоря, если оба операнда являются объектами, вы можете использовать как ==, так и ===, потому что они работают абсолютно одинаково. Вот что происходит при проверке двух объектов на равенство.

При проверке равенства двух объектных переменных сравниваются ссылки на эти объекты

Вспомните: в переменных хранятся ссылки на объекты, поэтому каждый раз, когда мы сравниваем два объекта, сравниваются ссылки на эти объекты.

```
if (var1 === var2) {
    // Да это же один объект!
}
```

Не в этом случае!

Не важно, что хранится в этих объектах. Если ссылки не одинаковые, то и объекты не равны.

Две ссылки равны только в том случае, если они ссылаются на один объект

Проверка равенства двух переменных, содержащих ссылки на объекты, возвращает true только в том случае, если две ссылки указывают на один объект.

```
if (var1 === var3) {
    // Да это же один объект!
}
```

Наконец-то — две ссылки на объекты равны.

Эта ссылка...

var3 var1 var2

...равна этой ссылке. Они относятся к одному объекту, а следовательно, равны!

Возьми в руку карандаш

Эта маленькая программа предназначена для поиска машин на стоянке. Проанализируйте код и запишите значения переменных loc1-loc4 внизу.

```
function findCarInLot(car) {
    for (var i = 0; i < lot.length; i++) {
        if (car === lot[i]) {
            return i;
        }
    }
    return -1;
}
var chevy = {
    make: "Chevy",
    model: "Bel Air"
};
var taxi = {
    make: "Webville Motors",
    model: "Taxi"
};
var fiat1 = {
    make: "Fiat",
    model: "500"
};
var fiat2 = {
    make: "Fiat",
    model: "500"
};

var lot = [chevy, taxi, fiat1, fiat2];

var loc1 = findCarInLot(fiat2);    ____________
var loc2 = findCarInLot(taxi);     ____________
var loc3 = findCarInLot(chevy);    ____________
var loc4 = findCarInLot(fiat1);    ____________
```

Ваши ответы.

Эрл, владелец стоянки.

Псевдоистина где-то рядом...

Да, все верно — «псевдоистина», а не истина. И еще есть «псевдоложь». Что вообще имеется в виду? В некоторых языках используются точные, формальные определения true и false. В JavaScript дело обстоит иначе. Собственно, в JavaScript эти понятия размыты: существуют значения, которые не являются ни true, ни false, но при этом интерпретируются как true или false в условных выражениях. Мы называем эти выражения «псевдоистиной» и «псевдоложью», потому что с технической точки зрения они не являются булевскими значениями, но ведут себя как булевские значения (опять же — в условных выражениях).

Как разобраться во всех этих тонкостях? Существует простое правило: *сначала выясните, что считается псевдоложью, а все остальное будет псевдоистиной*. Рассмотрим несколько примеров:

```
var testThis;
if (testThis) {
   // ...
}
```

Да, выглядит странно — в условии эта переменная заведомо содержит undefined. Будет ли это выражение работать? Является ли оно корректным кодом JavaScript? (Ответ: да.)

```
var element = document.getElementById("elementThatDoesntExist");
if (element) {
   // ...
}
```

Значение element равно null. Что произойдет?

```
if (0) {
   // ...
}
```

Проверяем 0?

Выполняем условную проверку для пустой строки. Что будет? Рискнете сделать ставку?

```
if ("") {
   // Этот код когда-нибудь будет выполнен? Принимаем ставки.
}
```

Секунду, мы используем NaN в булевском выражении? И как это значение будет интерпретировано?

```
if (NaN) {
   // Хм, а что NaN делает в логическом условии?
}
```

Что JavaScript считает «псевдоложью»

Еще раз: чтобы запомнить, какие значения считаются псевдоистинными, а какие псевдоложными, следует запомнить псевдоложные. Остальные значения относятся к псевдоистинным. Итак, псевдоложные значения:

undefined

null

0

пустая строка

NaN

Просто запомните пять псевдоложных значений — undefined, null, 0, "" и NaN. Все остальное JavaScript рассматривает как псевдоистину.

Следовательно, при вычислении всех условий на предыдущей странице будет получен результат false. Мы уже говорили, что все остальные значения псевдоистинны (кроме false, разумеется)? Несколько примеров псевдоистинных значений:

```
if ([]) {
   // Будет выполняться
}

var element = document.getElementById("elementThatDoesExist");
if (element) {
   // Это тоже
}

if (1) {
   // И это
}

var string = "mercy me";
if (string) {
   // И это
}
```

Это массив. Он не является undefined, null, zero, "" или NaN. А значит, интерпретируется как true!

На этот раз мы имеем дело с объектом element. Он не относится к псевдоложным значениям, а значит, является псевдоистинным.

В набор псевдоложных значений входит только 0, все остальные числа истинны.

Псевдоложна только пустая строка, все остальные строки псевдоистинны.

Возьми в руку карандаш

А вы умеете управляться с детектором лжи? Определите, сколько ложных ответов дал подозреваемый, а также виновен ли он по предъявленным обвинениям (три и более ложных ответа). Прежде чем двигаться дальше, сверьтесь с ответами в конце главы. И конечно, если захотите — самостоятельно проведите проверку в браузере!

```
function lieDetectorTest() {
    var lies = 0;

    var stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    var car = {
        keysInPocket: null
    };
    if (car.keysInPocket) {
        console.log("Uh oh, guess you stole the car!");
        lies++;
    }
    if (car.emptyGasTank) {
        console.log("You drove the car after you stole it!");
        lies++;
    }
    var foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    var yourName = " ";
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
var numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

Строка из одного пробела.

Как вы думаете, что делает этот код? Не видите ли вы в нем каких-то странностей — особенно с учетом того, что вам известно о примитивных типах?

```
var text = "YOU SHOULD NEVER SHOUT WHEN TYPING";
var presentableText = text.toLowerCase();
if (presentableText.length > 0) {
     alert(presentableText);
}
```

Тайная жизнь строк

Типы всегда относятся к одной из двух категорий: они являются либо примитивными типами, либо объектами. Примитивы ведут относительно простую жизнь, тогда как объекты обладают состоянием и поведением (или, если говорить иначе, обладают свойствами и методами). Так?

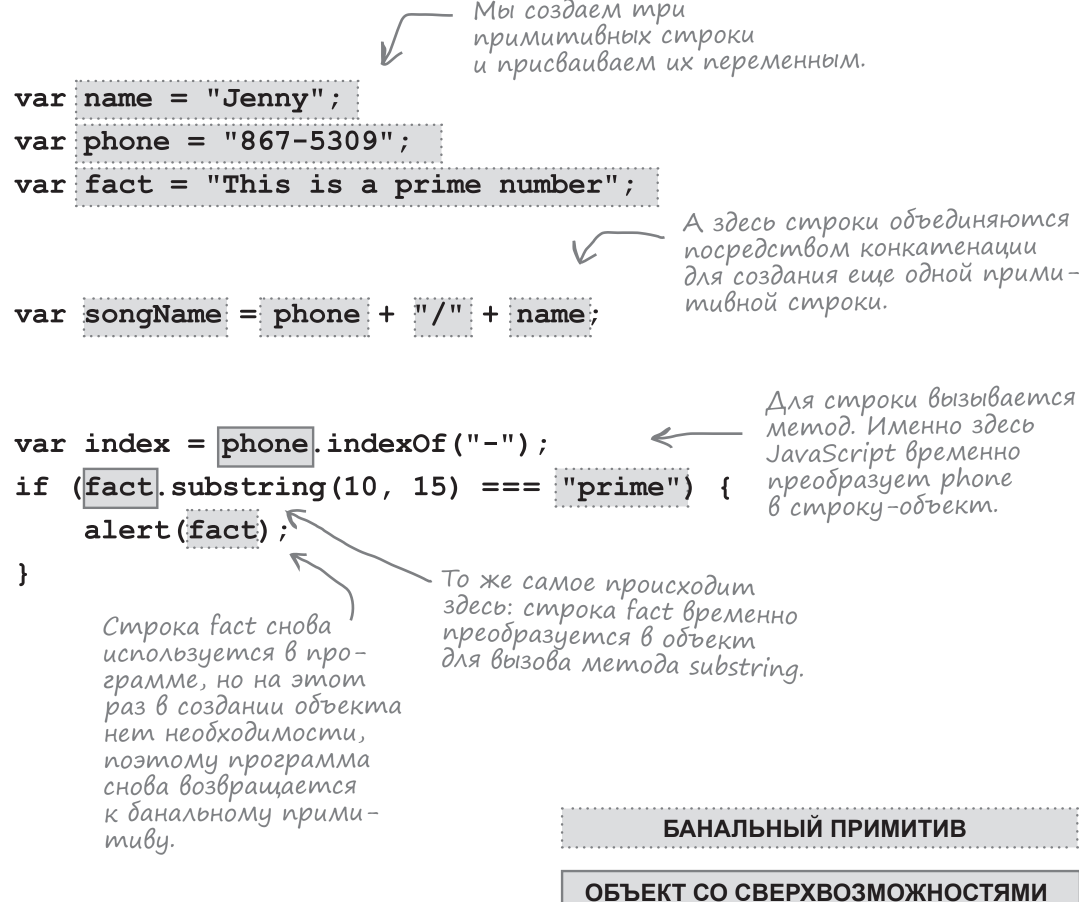
Безусловно, все сказанное — правда, но это не вся картина. Со строками дело обстоит сложнее. Рассмотрим следующий код:

```
var emot = "XOxxOO";          // Обычная, примитивная строка..
var hugs = 0;
var kisses = 0;

emot = emot.trim();           // Секундочку! Вызываем метод для строки?
emot = emot.toUpperCase();

for(var i = 0; i < emot.length ; i++) {     // Строка со свойством?
    if (emot.charAt(i) === "X") {          // Снова методы?
       hugs++;
    } else if (emot.charAt(i) == "O") {
       kisses++;
    }
}
```

Строка может выглядеть и как примитив, и как объект

Получается, что строка выглядит и как примитив, и как объект. Почему? Потому что JavaScript поддерживает обе возможности. Иначе говоря, в JavaScript можно создать как строку-примитив, так и строку-объект (с поддержкой множества полезных методов обработки строк). Ранее мы не упоминали о том, как создать строку-объект, и в большинстве случаев вам не придется это делать явно, потому что интерпретатор JavaScript *создает строки-объекты за вас* при необходимости.

Но где и почему может возникнуть такая необходимость? Рассмотрим жизненный цикл строки:



Просто голова идет кругом. Значит, строка преобразуется туда-сюда между примитивом и объектом? И как мне уследить за всем этим?

А вам и не нужно. Обычно вы можете просто рассматривать свои строки как объекты с множеством полезных методов, упрощающих манипуляции с текстом. JavaScript позаботится обо всех подробностях. Взгляните на это так: вы теперь гораздо лучше понимаете, что происходит «за кулисами» JavaScript, но в повседневном программировании можно просто положиться на то, что JavaScript сделает все правильно за вас (так обычно и происходит).

Часто Задаваемые Вопросы

В: На всякий случай уточню: мне когда-нибудь придется следить за тем, где моя строка представляет собой примитив, а где — объект?

О: В большинстве случаев — нет. Интерпретатор JavaScript выполнит все преобразования за вас. Вы просто пишете свой код, предполагая, что строка поддерживает свойства и методы объекта, и все работает так, как положено.

В: Почему в JavaScript поддерживается двойственная природа строк (как примитив и как объект)?

О: Вы пользуетесь эффективностью простых примитивных строковых типов (при условии, что вы ограничиваетесь базовыми строковыми операциями — сравнениями, конкатенациями, записью строк в DOM и т. д. Но если вам потребуются более сложные строковые операции, тогда в вашем распоряжении мгновенно появляется объект строки.

В: Имеется произвольная строка. Как узнать, чем она является — объектом или примитивом?

О: Строка всегда является примитивом, если только она не была создана специальным образом (с использованием конструктора). Конструкторы объектов будут рассмотрены позднее. Вы всегда можете применить к своей переменной оператор typeof, чтобы узнать, относится ли она к типу string или object.

В: Могут ли другие примитивы вести себя как объекты?

О: Да, числа и булевские значения тоже способны вести себя как объекты. Однако ни те ни другие не обладают таким количеством полезных свойств, как строки, поэтому для них эта функциональность используется намного реже, чем для строк. И помните: все это происходит «за кулисами», так что вам не придется активно следить за происходящим. Просто используйте свойство, если понадобится, и предоставьте JavaScript выполнить временное преобразование за вас.

В: Как получить полный список всех методов и свойств, доступных для объектов String?

О: Воспользуйтесь хорошим справочником. Существует множество полезных электронных документов, а если вы предпочитаете книгу — в *JavaScript: The Definitive Guide* содержится справочник с полным перечнем свойств и методов строк в JavaScript. Впрочем, о Google тоже не стоит забывать.

Краткий обзор методов (и свойств) строк

Раз уж мы занялись строками и вы узнали, что строки также поддерживают методы, давайте ненадолго отвлечемся от типов с их странностями и рассмотрим несколько типичных методов строк, которые вам наверняка пригодятся. Некоторые методы и свойства строк практически постоянно используются в программах, и вы не пожалеете о времени, потраченном на их изучение. За дело!

Конечно, мы могли написать целую главу обо всех строках и методах, поддерживаемых строками. От этого книга разрослась бы до 2000 страниц, но на данный момент это и не нужно, вы уже разбираетесь в основах использования методов и объектов, и если вам потребуется изучить обработку строк более подробно, достаточно будет хорошего справочника.

Свойство length

Свойство length содержит количество символов в строке. Его удобно использовать при переборе символов строки.

Свойство length используется для последовательного перебора символов строки.

```
var input = "jenny@wickedlysmart.com";
for(var i = 0; i < input.length; i++) {
    if (input.charAt(i) === "@") {
        console.log("There's an @ sign at index " + i);
    }
}
```

А метод charAt получает символ с конкретным индексом в строке.

Консоль JavaScript

```
There's an @ sign at index 5
```

Метод charAt

Метод charAt получает целое число от 0 до длины строки (минус 1) и возвращает строку с одним символом, находящимся в заданной позиции. Строку можно рассматривать как массив, элементами которого являются отдельные символы, а индексы начинаются с 0 (как и в обычных массивах). Если переданный индекс больше либо равен длине строки, возвращается пустая строка.

Учтите, что в JavaScript не существует символьного типа, поэтому символы возвращаются в виде новых строк, содержащих всего один символ.

a	b	c	d	e	f

charAt(0) содержит "a".

charAt(5) содержит "f".

Метод indexOf

Метод получает строку-аргумент и возвращает индекс первого символа первого вхождения аргумента в своей строке.

Строка, для которой будет вызываться метод indexOf.

```
var phrase = "the cat in the hat";
```

Мы хотим найти позицию первого вхождения "cat" в исходной строке.

```
var index = phrase.indexOf("cat");
console.log("there's a cat sitting at index " + index);
```

Возвращается индекс первого вхождения "cat".

```
Консоль JavaScript
There's a cat sitting at index 4
```

Также можно передать второй аргумент — индекс начальной позиции поиска.

```
index = phrase.indexOf("the", 5);
console.log("there's a the sitting at index " + index);
```

Так как поиск начинается с индекса 5, первое вхождение "the" пропускается, и обнаруживается второе вхождение "the" с начальным индексом 11.

```
Консоль JavaScript
There's a the sitting at index 11
```

```
index = phrase.indexOf("dog");
console.log("there's a dog sitting at index " + index);
```

Если строку найти не удалось, метод возвращает -1.

```
Консоль JavaScript
There's a dog sitting at index -1
```

Метод substring

Метод substring получает два индекса, после чего извлекает и возвращает заключенную между ними строку.

```
var data = "name|phone|address";
var val = data.substring(5, 10);
console.log("Substring is " + val);
```

Строка, для которой будет вызываться substring.

Метод возвращает строку, начинающуюся с индекса 5 и заканчивающуюся индексом 10 (не включая его).

Мы получаем новую строку, состоящую из символов с индексами от 5 до 10.

Консоль JavaScript

```
Substring is phone
```

Второй индекс можно не указывать, в этом случае substring извлекает строку, которая начинается с первого индекса и продолжается до конца исходной строки.

```
val = data.substring(5);
console.log("Substring is now " + val);
```

Консоль JavaScript

```
Substring is now phone|address
```

Метод split

Метод split получает символ-ограничитель и разбивает строку на части по позиции ограничителя.

```
var data = "name|phone|address";
var vals = data.split("|");
console.log("Split array is ", vals);
```

Метод split использует ограничитель для разбиения исходной строки на части, которые возвращаются в массиве.

Консоль JavaScript

```
Split array is ["name", "phone", "address"]
```

Обратите внимание: при вызове console.log передаются два аргумента, разделенных запятой. В этом случае массив vals не преобразуется в строку перед выводом на консоль.

Строковый суп

toLowerCase

Возвращает строку, в которой все символы верхнего регистра преобразуются к нижнему регистру.

replace

Находит подстроки и заменяет их другой строкой.

lastIndexOf

Аналогичен indexOf, но находит последнее вхождение (вместо первого).

slice

Возвращает новую строку, из которой удалена часть исходной строки.

concat

Соединяет строки.

match

Ищет совпадения регулярного выражения в строке.

toUpperCase

Возвращает строку, в которой все символы нижнего регистра преобразуются к верхнему регистру.

substring

Возвращает часть строки.

trim

Удаляет пропуски с обоих концов строки. Метод удобен при обработке данных, вводимых пользователем.

Возможности работы со строками широки. Вот еще несколько методов, которые вы можете использовать в своих программах. Сейчас достаточно самого поверхностного знакомства, а когда потребуется — вы сможете поискать дополнительную информацию...

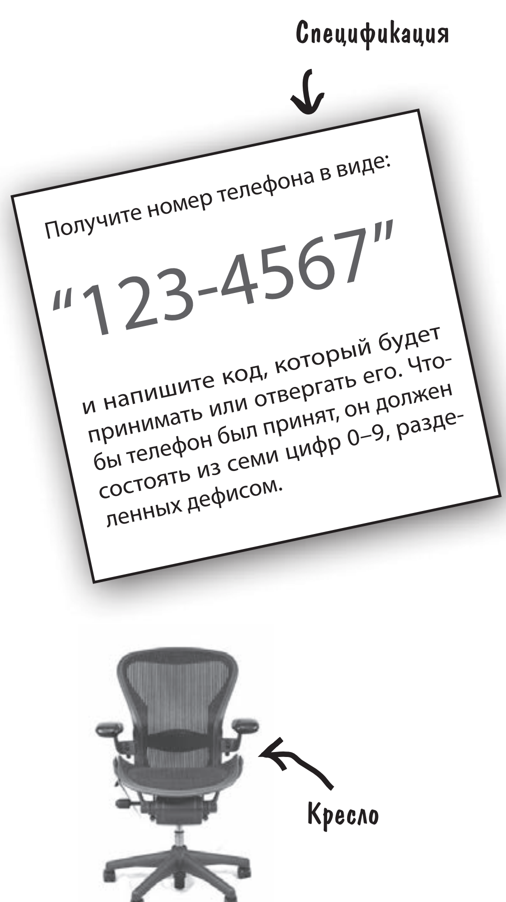
Битва за кресло

(или как хорошее знание типов может изменить вашу жизнь)

Однажды двум программистам дали одну и ту же спецификацию. Противный Руководитель Проекта заставил двух программистов соревноваться, он пообещал, что тот, кто первым сдаст работоспособное решение, получит крутое офисное кресло, как у всех серьезных людей в Кремниевой Долине. Брэд, ветеран сценарного программирования, и Ларри, недавний выпускник, решили, что задача не так уж сложна.

Ларри, сидя за компьютером, подумал: «Что должен делать этот код? Проверить, что строка имеет достаточную длину, что в середине стоит дефис и что все остальные символы являются цифрами. Я могу воспользоваться свойством `length` и умею обращаться к символам строки методом `charAt`».

Брэд откинулся на спинку кресла в кафе и принялся размышлять над тем же вопросом: «Что должен делать этот код?» Сначала он подумал: «Строка — это объект. Существует множество методов, которые я могу использовать для проверки введенного телефона. Я воспользуюсь ими, и все сработает. В конце концов, объект есть объект». Читайте дальше, и вы узнаете, как Брэд и Ларри строили программы, а заодно получите ответ на главный вопрос: ***кому досталось кресло?***



В комнате Ларри

Ларри хочет написать код с использованием методов строк:

```
function validate(phoneNumber) {
    if (phoneNumber.length !== 8) {
       return false;
    }
    for (var i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.charAt(i) !== '-') {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

При помощи свойства length объекта строки Ларри узнает, сколько символов содержит строка.

Он использует метод charAt для проверки каждого символа строки.

Сначала он убеждается в том, что символ с индексом 3 содержит дефис.

А затем проверяет, что каждый символ с индексами 0–2 и 4–6 содержит цифру.

В комнате Брэда

Брэд написал код для проверки двух чисел и дефиса:

```
function validate(phoneNumber) {
    if (phoneNumber.length !== 8) {
        return false;
    }
    var first = phoneNumber.substring(0,3);
    var second = phoneNumber.substring(4);
    if (phoneNumber.charAt(3) !== "-" || isNaN(first) || isNaN(second)) {
        return false;
    }
    return true;
}
```

Брэд начинает так же, как Ларри...

Но он использует свое знание строковых методов.

Метод substring создает строку, содержащую символы с нулевого по третий.

И другую подстроку, начиная с символа с индексом 4 и до конца строки.

Затем он проверяет все критерии правильности телефона в одном условии.

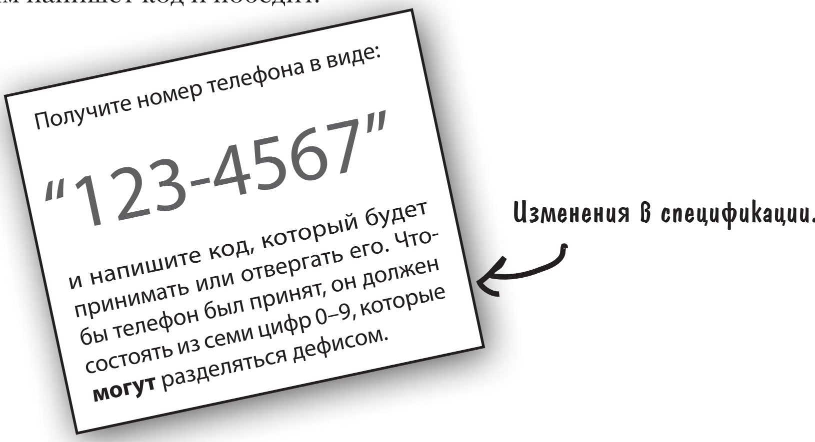
Интересный момент: сознательно или нет, Брэд зависит от преобразований типа для преобразования строки в число, а затем проверяет, что это именно число, при помощи функции isNaN. Умно!

Погодите! Спецификация изменилась

«Формально Ларри был первым, потому что Брэду пришлось искать описания всех этих методов, — сказал Руководитель Проекта, — но в спецификации кое-что изменилось. Конечно, это не создаст проблем для таких опытных программистов, как вы».

«Если бы мне давали монетку каждый раз, когда я это слышу», — подумал Ларри, хорошо знавший, что изменений спецификации без проблем не бывает. «Но Брэд выглядит подозрительно спокойным. Что бы это значило?» Тем не менее Ларри все еще полагает, что способ Брэда — обычный выпендреж. И что в следующем раунде он снова первым напишет код и победит.

А вы не видите каких-нибудь ошибок, которые может вызвать использование isNaN в программе Брэда?



Изменения в спецификации.

Снова в комнате Ларри

Ларри подумал, что большую часть существующего кода можно использовать повторно; нужно лишь разобраться с особым случаем — возможным отсутствием дефиса в номере. Телефон состоит либо только из семи цифр, либо из восьми с дефисом в третьей позиции. Ларри быстро запрограммировал изменения (конечно, программу пришлось потестировать, чтобы она правильно заработала):

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
           phoneNumber.length < 7) {
       return false;
    }
    for (var i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.length === 8 &&
                   phoneNumber.charAt(i) !== '-') {
              return false;
           } else if (phoneNumber.length === 7 &&
                   isNaN(phoneNumber.charAt(i))) {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Ларри внес несколько изменений в логику. Кода немного, но разобраться в нем стало посложнее.

Брэд с ноутбуком на пляже

Брэд улыбнулся, отхлебнул из бокала и быстро внес изменения. Он просто определил вторую часть числа по длине телефонного номера без 4 (начальная точка подстроки) — вместо того, чтобы жестко фиксировать начальную позицию при наличии дефиса. Работа была почти выполнена, хотя проверку дефиса нужно было переписать, потому что она применяется только для телефонного номера из восьми цифр.

Почти столько же изменений, как у Ларри, но код Брэда по-прежнему лучше читается.

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
        phoneNumber.length < 7) {
        return false;
    }
    var first = phoneNumber.substring(0,3);
    var second = phoneNumber.substring(phoneNumber.length - 4);

    if (isNaN(first) || isNaN(second)) {
      return false;
    }
    if (phoneNumber.length === 8) {
      return (phoneNumber.charAt(3) === "-");
    }
    return true;
}
```

Теперь Брэд получает вторую часть, определяя начальную позицию по общей длине номера.

А дефис проверяется только в том случае, если строка содержит восемь символов.

Здесь возвращается результат вычисления условия — true или false.

Кажется, программа Брэда все еще содержит ошибку. А вы сможете ее найти?

Как бы вы переписали код Брэда с использованием метода split?

Ларри чуть-чуть опередил Брэда

Но усмешка на лице Ларри мгновенно растаяла, когда Противный Руководитель Проекта сказал: «Брэд, твой код отлично читается, и с сопровождением не будет проблем. Отличная работа».

И все же Ларри не стоит отчаиваться; ведь мы знаем, что элегантность кода — еще не все. Код должен пройти проверку качества, а мы не уверены, что код Брэда будет работать во всех случаях. А что скажете вы? Кто, как вы думаете, заслужил кресло?

Снова эта проклятая неопределенность. Кому же досталось кресло?

Эми со второго этажа.

(Руководитель проекта в тайне от всех дал задание *трем* программистам.)

Ого, всего одна строка!

Код Эми.

```
function validate(phoneNumber) {
   return phoneNumber.match(/^\d{3}-?\d{4}$/);
}
```

Снова в лаборатории

Ученые из лаборатории продолжают исследовать JavaScript с помощью оператора **typeof**, и им удалось обнаружить нечто интересное. В процессе исследований они открыли новый оператор **instanceof**, с которым наконец-то состоится научный прорыв. Наденьте лабораторный халат и защитные очки и посмотрите, удастся ли вам расшифровать этот код JavaScript и результаты. *Предупреждение: это определенно самый странный код из всего, что вы видели ранее.*

Это программа. Прочитайте ее, запустите, внесите изменения, посмотрите, что она делает...

Как странно. Какая-то смесь функции и объекта?

```
function Duck(sound) {
    this.sound = sound;
    this.quack = function() {console.log(this.sound);}
}

var toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Хмм, "new". Мы еще не видели такую конструкцию. Можно предположить, что она создает новый объект Duck и присваивает его переменной toy.

Если это выглядит как объект... Давайте проверим.

Порядок, и мы видим экземпляр...

Обязательно сверьте свой результат с ответами в конце главы. Но что это все значит? Ответ на этот вопрос вы получите через пару глав. А пока, если вы еще не заметили — вы постепенно становитесь крутым программистом JavaScript. Это очень серьезная тема!

Консоль JavaScript

Запишите здесь свои результаты. Какие-нибудь сюрпризы?

КЛЮЧЕВЫЕ МОМЕНТЫ

- В JavaScript существуют две категории типов: **примитивы** и объекты. Любое значение, которое не относится к примитивному типу, является **объектом**.
- Примитивы: числа, строки, булевские значения, null и undefined. Все остальное — объекты.
- **undefined** означает, что переменная (или свойство, или элемент массива) еще не была инициализирована значением.
- **null** означает «нет объекта».
- NaN означает «не число», хотя правильнее рассматривать **NaN** как число, которое не может быть представлено в JavaScript. NaN относится к числовому типу.
- Значение NaN никогда не равно любому другому значению, даже самому себе, поэтому для проверки на NaN следует использовать функцию **isNaN**.
- Два значения проверяются на равенство операторами == или ===.
- Если два операнда имеют разные типы, оператор проверки равенства (==) пытается преобразовать один из операндов к другому типу перед выполнением проверки.
- Если два операнда имеют разные типы, оператор строгого равенства (===) возвращает false.
- Используйте ===, если хотите избежать проверки типов. Однако помните, что преобразования типа == иногда бывают полезны.
- Преобразование типа также используется с другими операторами, например арифметическими и оператором конкатенации строк.
- В JavaScript существуют пять **псевдоложных** значений: undefined, null, 0, “” (пустая строка) и false. Все остальные значения являются **псевдоистинными**.
- Строки иногда ведут себя как объекты. Если использовать свойство или метод для примитивной строки, JavaScript временно преобразует строку в объект, использует свойство, а затем преобразует результат обратно в примитивную строку. Это происходит автоматически, так что вам даже не придется задумываться о преобразовании.
- Объекты строк поддерживают много методов, полезных для операций со строками.
- Два объекта равны только в том случае, если переменные, в которых хранятся ссылки, указывают на один объект.

Группа значений JavaScript и незваных гостей, облачившись в маскарадные костюмы, развлекаются игрой «Кто я?». Они дают подсказки, а вы должны угадать их по тому, что они говорят о себе. Предполагается, что участники всегда говорят о себе правду. Соедините линией каждое высказывание с именем соответствующего участника. Мы уже провели одну линию за вас.

Ниже приведено наше решение.

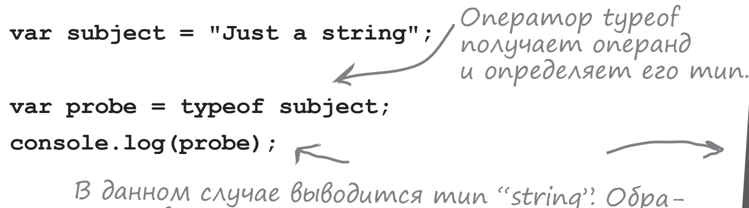
Сегодняшние участники:

Я возвращаюсь из функции при отсутствии команды return. → undefined	нуль
Я считаюсь значением переменной, которой еще не было присвоено значение. → undefined	пустой объект
Я — значение элемента, не существующего в разреженном массиве. → undefined	null
Я — значение несуществующего свойства. → undefined	undefined
Я — значение удаленного свойства. → undefined	NaN
Я — значение, которое не может быть задано свойству при создании объекта. → undefined	infinity
	area 51
	...---...
	{}
	[]

В ЛАБОРАТОРИИ РЕШЕНИЕ

В своей лаборатории мы любим разбирать, заглядывать «под капот», экспериментировать, подключать диагностические инструменты и выяснять, что же происходит на самом деле. Сегодня в ходе исследования системы типов JavaScript мы обнаружили маленький диагностический инструмент для анализа переменных — **typeof**. Наденьте лабораторный халат и защитные очки, заходите и присоединяйтесь к нам.

Оператор **typeof** встроен в JavaScript. Он используется для проверки типа операнда. Пример:




```
var subject = "Just a string";

var probe = typeof subject;
console.log(probe);
```

Консоль JavaScript

```
string
```

Теперь ваша очередь. Соберите данные по следующим экспериментам:

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Тестовые данные и тесты.

Консоль JavaScript

```
string
number
boolean
object
object
undefined
object
object
function
```

Наши результаты.

Снова в лаборатории РЕШЕНИЕ

Стоп, мы забыли включить null в тестовые данные. Недостающий тестовый пример:

```
var test10 = null;

console.log(typeof test10);
```

Наш результат.

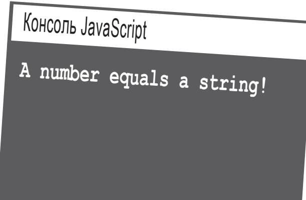
Консоль JavaScript

```
object
```

Упражнение Решение

В этой главе мы рассматривали некоторые... ммм... интересные значения. А теперь рассмотрим интересное поведение. Попробуйте добавить приведенный ниже код под элементом <script> обычной веб-страницы и посмотрите, что будет выведено на консоль при загрузке страницы. Причины вам пока неизвестны, но попробуйте хотя бы предположить, что при этом происходит.

```
if (99 == "99") {
    console.log("A number equals a string!");
} else {
    console.log("No way a number equals a string");
}
```



То, что получилось у нас.

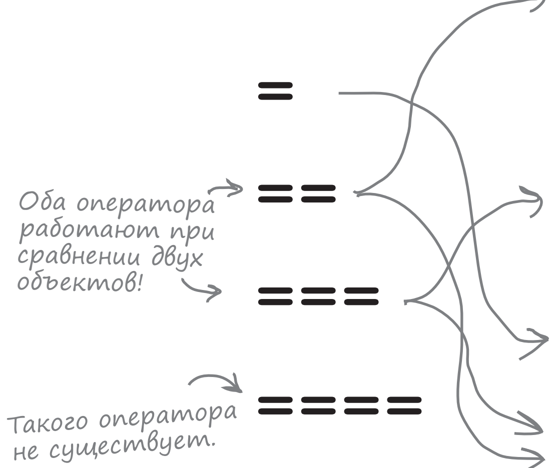
Для каждого сравнения запишите его результат (true или false) при использовании операторов == и ===:

	==	===	
`"42" == 42`	true	false	`"42" === 42`
`"0" == 0`	true	false	`"0" === 0`
`"0" == false`	true	false	`"0" === false`
`"true" == true`	false	false	`"true" === true`
`true == (1 == "1")`	true	false	`true === (1 === "1")`

Трудный вопрос!

Если заменить оба оператора == на ===, получится false.

Мы подобрали описания для всех операторов, но они как-то перепутались. Помогите нам разобраться, кто что делает. Будьте внимательны: мы не уверены в том, сколько возможных описаний может иметь каждый претендент – нуль, одно или более. Ниже приведено наше решение:



Сравнивает значения и проверяет их на равенство. Он настолько внимателен, что пытается выполнить преобразование типов, чтобы узнать, действительно ли значения равны.

Сравнивает значения и проверяет их на равенство. При этом значения, относящиеся к разным типам, попросту не рассматриваются.

Присваивает значение переменной.

Сравнивает ссылки на объекты и возвращает true, если они равны, и false в противном случае.

Возьми в руку карандаш
Решение

Пришло время проверить, насколько вы понимаете преобразования. Запишите результат каждого выражения справа от него. Одно задание мы уже выполнили за вас. Ниже приведено наше решение.

Выражение	Результат
`Infinity - "1"`	Infinity
`"42" + 42`	"4242"
`2 + "1 1"`	"21 1"
`99 + 101`	200
`"1" - "1"`	0
`console.log("Result: " + 10/2)`	"Result: 5"
`3 + " bananas " + 2 + " apples"`	"3 bananas 2 apples"

"1" преобразуется в 1, а Infinity — 1 дает Infinity.

Обе строки преобразуются в 1, а 1-1 дает 0.

Сначала выполняется деление 10/2, затем результат присоединяется к строке "Result: "

Каждый оператор + выполняет конкатенацию, потому что в обоих случаях операндом является строка.

Возьми в руку карандаш
Решение

А вы умеете управляться с детектором лжи? Определите, сколько ложных ответов дал подозреваемый, а также виновен ли он по предъявленным обвинениям (три и более ложных ответа). Прежде чем двигаться дальше, сверьтесь с ответами в конце главы. Ниже приведено наше решение. А вы проверили его в браузере?

```
function lieDetectorTest() {
    var lies = 0;

    var stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    var car = {
        keysInPocket: null
    };
    if (car.keysInPocket) {
        console.log("Uh oh, guess you stole the car!");
        lies++;
    }
    if (car.emptyGasTank) {
        console.log("You drove the car after you stole it!");
        lies++;
    }
    var foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    var yourName = " ";
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
var numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

Любой объект — даже пустой — рассматривается как псевдоистина.

Подозреваемый не угонял машину, потому что значение свойства keysInPocket равно null — псевдоложь.

И он не вел машину, потому что свойство emptyGasTank содержит undefined — псевдоложное значение.

Но [ ] (пустой массив) является псевдоистинным значением, так что подозреваемый был пойман на месте преступления.

Элемент с индексом 0 в массиве отсутствует, соответственно, он содержит undefined — псевдоложное значение. Похоже, подозреваемый уже спрятал добро.

Строка из одного пробела.

Любая непустая строка псевдоистинна, даже если она состоит только из одного пробела!

Консоль JavaScript

```
You stole the diamond
A sure sign of guilt
Guess you lied about your name
You told 3 lies!
Guilty as charged
```

Три ложных утверждения... Пожалуй, подозреваемый виновен.

Возьми в руку карандаш Решение

Эта маленькая программа предназначена для поиска машин на стоянке. Проанализируйте код и запишите значения переменных loc1-loc4 внизу.

```
function findCarInLot(car) {
    for (var i = 0; i < lot.length; i++) {
        if (car === lot[i]) {
            return i;
        }
    }
    return -1;
}
var chevy = {
    make: "Chevy",
    model: "Bel Air"
};
var taxi = {
    make: "Webville Motors",
    model: "Taxi"
};
var fiat1 = {
    make: "Fiat",
    model: "500"
};
var fiat2 = {
    make: "Fiat",
    model: "500"
};

var lot = [chevy, taxi, fiat1, fiat2];

var loc1 = findCarInLot(fiat2);    3
var loc2 = findCarInLot(taxi);     1
var loc3 = findCarInLot(chevy);    0
var loc4 = findCarInLot(fiat1);    2
```

Наши ответы.

Эрл, владелец стоянки.

Снова в лаборатории. Решение

Ученые из лаборатории продолжают исследовать JavaScript с помощью оператора **typeof**, и им удалось обнаружить нечто интересное. В процессе исследований они открыли новый оператор **instanceof**, с которым наконец-то состоится научный прорыв. Наденьте лабораторный халат и защитные очки и посмотрите, удастся ли вам расшифровать этот код JavaScript и результаты. *Предупреждение: это определенно самый странный код из всего, что вы видели ранее.*

Это программа. Прочитайте ее, запустите, внесите изменения, посмотрите, что она делает...

```
function Duck(sound) {
    this.sound = sound;
    this.quack = function() {console.log(this.sound);}
}

var toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Как странно. Какая-то смесь функции и объекта?

Хмм, "new". Мы еще не видели такую конструкцию. Можно предположить, что она создает новый объект Duck и присваивает его переменной toy.

Если это выглядит как объект... Давайте проверим.

Порядок, и мы видим экземпляр...

И что же все значит? Ответ на этот вопрос вы получите через пару глав. А пока, если вы еще не заметили — вы постепенно становитесь крутым программистом JavaScript. Это очень серьезная тема!

8 Все вместе

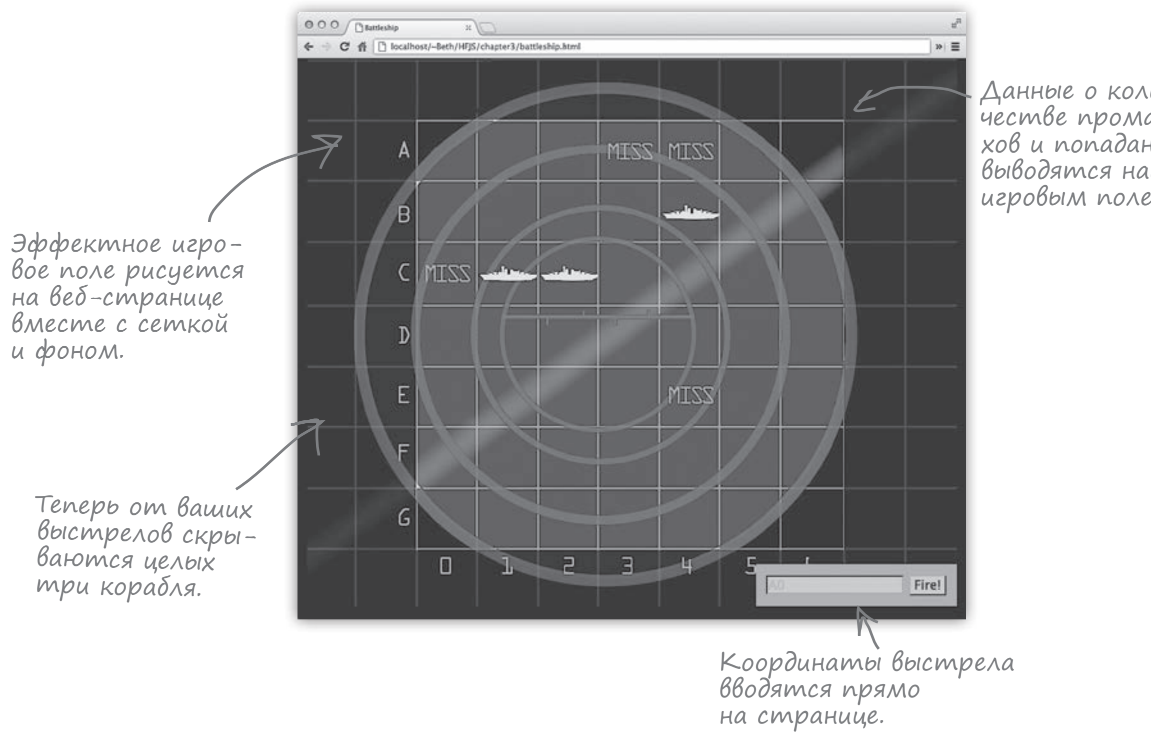
Построение приложения

Подготовьте свой инструментарий к работе. Да, ваш инструментарий — ваши новые навыки программирования, ваше знание DOM и даже некоторое знание HTML и CSS. В этой главе мы объединим все это для создания своего первого полноценного **веб-приложения**. Довольно **примитивных игр** с одним кораблем, который размещается в одной строке. В этой главе мы построим **полную версию**: большое игровое поле, несколько кораблей, ввод данных пользователем прямо на веб-странице. Мы создадим структуру страницы игры в разметке HTML, применим визуальное оформление средствами CSS и напишем код JavaScript, определяющий поведение игры. Приготовьтесь: в этой главе мы займемся полноценным, серьезным программированием и напишем вполне серьезный код.

На этот раз мы построим НАСТОЯЩУЮ игру «Морской бой»

Бесспорно, вы имеете полное право гордиться тем, что в главе 2 мы построили свой миниатюрный «Морской бой» с нуля, но давайте признаем: игра была немного *ненастоящей* — она работала, в нее можно было играть, но вряд ли такая программа может произвести впечатление на друзей или привлечь средства на развитие бизнеса. Чтобы продукт был действительно впечатляющим, нужно эффектное игровое поле, стильные изображения кораблей и возможность делать ходы прямо на игровом поле (вместо обобщенного диалогового окна в браузере). И вообще, хорошо бы улучшить предыдущую версию и ввести в нее поддержку всех трех кораблей.

Другими словами, игра должна выглядеть примерно так:



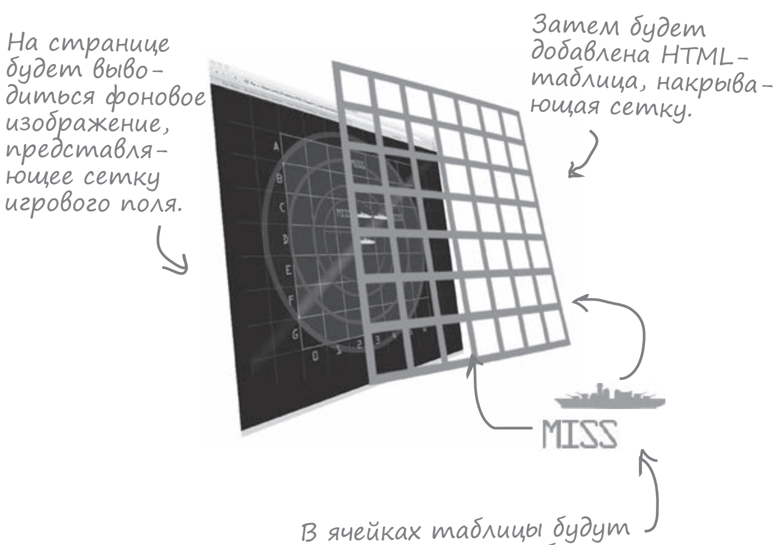
Забудьте на минуту о JavaScript... Взгляните на прототип. Если сосредоточиться на структуре и визуальном представлении страницы, как бы вы создали ее с использованием HTML и CSS?

Возвращаемся к HTML и CSS

Чтобы создать современную интерактивную веб-страницу (или *приложение*), придется работать с тремя технологиями: HTML, CSS и JavaScript. Наверняка вы уже слышали поговорку: «HTML для структуры, CSS для стилевого оформления, JavaScript для поведения». Но мы не ограничимся простым повторением. В этой главе данный принцип будет в полной мере реализован. И начнем мы с HTML и CSS.

Наша первая задача — воспроизвести внешний вид игрового поля с предыдущей страницы. Впрочем, *только* воспроизвести недостаточно, нужно реализовать его так, чтобы его структура позволяла пользователю вводить данные и отображать попадания, промахи и сообщения прямо на странице.

Для этого нам, например, придется наложить фоновое изображение, чтобы имитировать экран радара. Потом на странице будет размещена более функциональная таблица HTML, в которой можно размещать корабли, маркеры попаданий и т. д. Наконец, мы воспользуемся формой HTML для получения введенных пользователем данных.

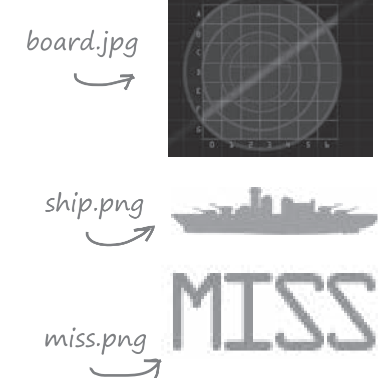


Итак, займемся построением игры. На нескольких ближайших страницах мы будем заниматься основными компонентами HTML и CSS, а когда основа приложения будет построена, можно будет переходить к написанию кода JavaScript.

ИНСТРУМЕНТЫ для создания приложения

Несколько готовых графических изображений помогут вам начать работу над новой версией игры.

INVENTORY includes...



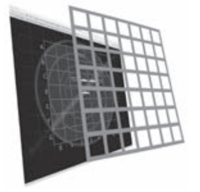
Пакет состоит из трех графических файлов: board.jpg — основной фон игрового поля, включая сетку; ship.png — небольшой кораблик для размещения на игровом поле (обратите внимание: это PNG-файл с прозрачностью, который может накладываться прямо поверх фона), и наконец, miss.png — маркер промаха, который также будет размещаться на поле. Как и положено в игре, при попадании в соответствующей клетке таблицы будет выводиться корабль, а при промахе — графический маркер промаха.

Все необходимое можно скачать по адресу http://wickedlysmart.com/hfjs

Создание страницы HTML: общая картина

Основные пункты плана по созданию HTML-страницы:

1. Начнем с фона игрового поля; для этого будет выбран черный цвет фона, после чего на странице будет размещена радарная сетка.



Размещаем графику на заднем плане, чтобы страница смотрелась более эффектно и напоминала зеленый светящийся экран радара.

2. Затем мы создадим таблицу HTML и разместим ее поверх фона. Каждая ячейка таблицы будет соответствовать одной клетке игрового поля.

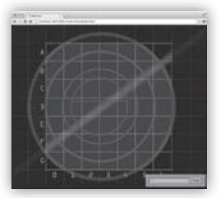


Таблица HTML у верхнего края фона образует игровое поле.

3. Затем мы добавим элемент формы HTML для ввода координат выстрелов, например "A4". Также будет добавлена область для вывода сообщений (например, «Ты потопил мой корабль!»).

Форма HTML для ввода данных игроком.

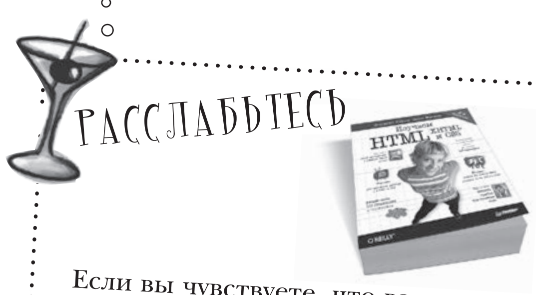
4. Остается разобраться с тем, как использовать таблицу для размещения графики кораблей (попадание) или маркера MISS (промах).

Эти изображения будут размещаться в таблице по ходу игры.

Шаг 1: Базовая разметка HTML

За дело! Прежде всего нам потребуется страница HTML. Мы создадим простую страницу по стандарту HTML5, а также добавим стилевое оформление для фона. В страницу будет включен элемент `<body>` с одним вложенным элементом `<div>` . Элемент `<div>` будет содержать сетку игрового поля.

На следующей странице приведена исходная версия разметки HTML и CSS нашей страницы.



РАССЛАБЬТЕСЬ

Если вы чувствуете, что ваши знания HTML и CSS неплохо было бы освежить, *Изучаем HTML, XHTML и CSS* отлично дополнит материал этой книги.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Battleship</title>
    <style>
      body {
        background-color: black;
      }

      div#board {
        position: relative;
        width: 1024px;
        height: 863px;
        margin: auto;
        background: url("board.jpg") no-repeat;
      }
    </style>
  </head>
  <body>
    <div id="board">

    </div>
    <script src="battleship.js"></script>
  </body>
</html>
```

Обычная страница HTML.

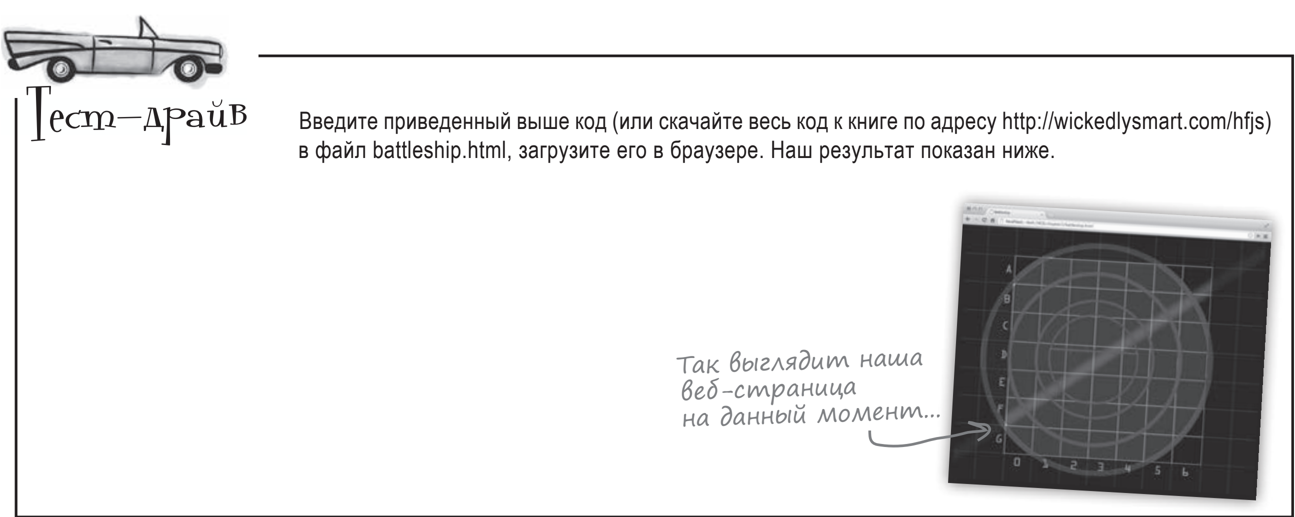
Страница будет иметь черный фон.

Игровое поле должно находиться в середине страницы, поэтому мы назначаем width значение 1024px (ширина игрового поля) с автоматическим выбором величины полей.

Здесь изображение board.jpg добавляется на страницу как фон элемента <div> "board". Для элемента <div> используется относительное позиционирование, чтобы мы могли разместить таблицу, которая будет добавлена на следующем шаге, относительно этого элемента <div>.

Здесь будет размещена таблица, представляющая игровое поле, и форма для получения пользовательского ввода.

Код будет размещен в файле battleship.js. Создайте для него пустой файл.



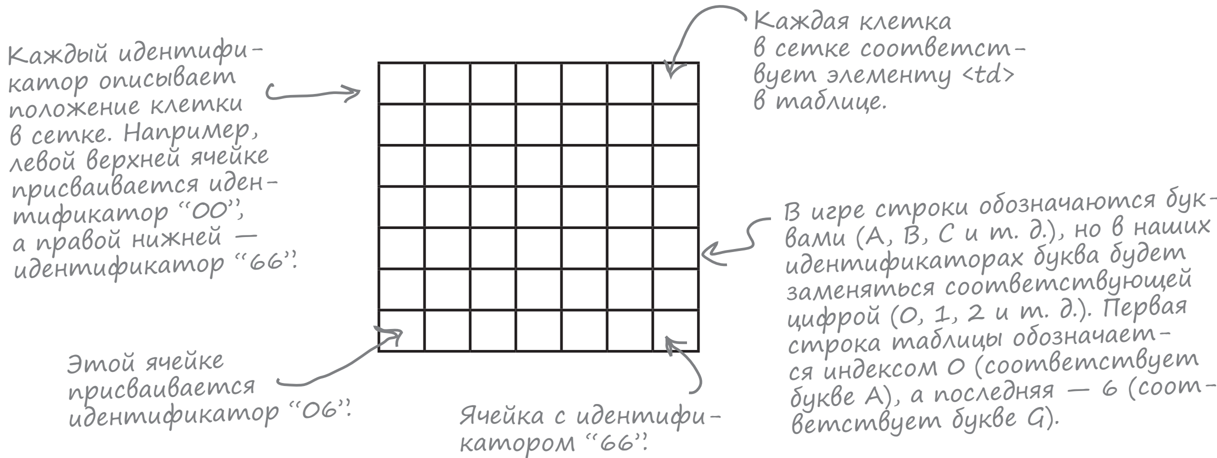
Тест-драйв

Введите приведенный выше код (или скачайте весь код к книге по адресу http://wickedlysmart.com/hfjs) в файл battleship.html, загрузите его в браузере. Наш результат показан ниже.

Так выглядит наша веб-страница на данный момент...

Шаг 2: Создание таблицы

Следующий пункт — таблица. Она размещается поверх сетки фонового изображения и предоставляет место для размещения графики попаданий и промахов в ходе игры. Каждая ячейка (или если вы помните HTML — каждый элемент `<td>`) будет размещаться прямо поверх клетки фонового изображения. А теперь самое важное: мы присваиваем каждой ячейке уникальный идентификатор, чтобы позднее с ней можно было работать из CSS и JavaScript. Давайте посмотрим, как создать идентификаторы и добавить разметку HTML-таблицы:



Разметка HTML-таблицы. Добавьте ее между тегами `<div>`:

```
<div id="board">
  <table>
    <tr>
      <td id="00"></td><td id="01"></td><td id="02"></td><td id="03">
</td><td id="04"></td> <td id="05"></td><td id="06"></td>
    </tr>
    <tr>
      <td id="10"></td><td id="11"></td><td id="12"></td><td id="13"></td>
<td id="14"></td> <td id="15"></td><td id="16"></td>
    </tr>
    ...
    <tr>
      <td id="60"></td><td id="61"></td><td id="62"></td><td id="63"></td>
<td id="64"></td><td id="65"></td><td id="66"></td>
    </tr>
  </table>
</div>
```

Таблица вкладывается в элемент <div> "board".

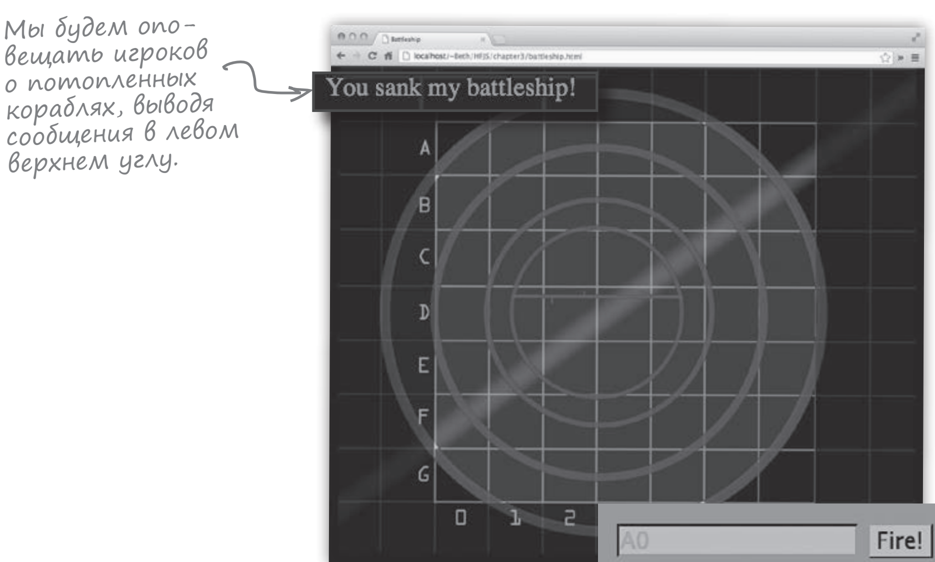
Каждому элементу <td> присваивается идентификатор, определяемый номерами строки и столбца в таблице.

Мы опускаем часть строк для экономии бумаги. Не сомневаемся, что вы сможете заполнить их самостоятельно.

Шаг 3: Взаимодействие с игроком

В приложение нужно включить элемент HTML для ввода координат выстрелов (например, "A0" или "E4") и элемент для вывода сообщений (например, «Ты потопил мой корабль!»). Мы используем элемент `<form>` с полем `<input>` для ввода координат, а сообщения будут выводиться в элементе `<div>`:

Мы будем оповещать игроков о потопленных кораблях, выводя сообщения в левом верхнем углу.



А здесь игроки будут вводить координаты выстрелов.

```
<div id="board">
  <div id="messageArea"></div>
  <table>
    ...
  </table>
  <form>
    <input type="text" id="guessInput" placeholder="A0">
    <input type="button" id="fireButton" value="Fire!">
  </form>
</div>
```

Элемент <div> с идентификатором messageArea будет использоваться для вывода сообщений.

Элемент <form> содержит два элемента input: текстовое поле для ввода выстрелов и кнопка. Обратите внимание на идентификаторы этих элементов. Они понадобятся нам позднее, когда мы будем писать код для ввода координат выстрелов.

Элемент <div> области сообщений, элементы <table> и <form> вкладываются в элемент <div> с идентификатором "board". Это обстоятельство будет использовано в CSS на следующей странице.

Добавление стилевого оформления

Если загрузить страницу сейчас (давайте, попробуйте), большинство элементов будет находиться в неположенных местах с неправильными размерами. Следовательно, мы должны написать CSS для позиционирования элементов и проследить за тем, чтобы размеры всех элементов (например, ячеек таблиц) соответствовали размерам игрового поля.

Чтобы разместить элементы в правильных местах, необходимо использовать позиционирование CSS для формирования макета. Так как элемент `<div>` "board" использует относительное позиционирование, мы теперь можем разместить область сообщений, таблицу и форму в конкретных местах `<div>`, чтобы они отображались именно так, как нам нужно.

Начнем с элемента `<div>` "messageArea". Он вложен в элемент `<div>` "board" и должен размещаться в левом верхнем углу игрового поля:

Область сообщений должна находиться в левом верхнем углу игрового поля.

```
body {
    background-color: black;
}
div#board {
    position: relative;
    width: 1024px;
    height: 863px;
    margin: auto;
    background: url("board.jpg") no-repeat;
}
div#messageArea {
    position: absolute;
    top: 0px;
    left: 0px;
    color: rgb(83, 175, 19);
}
```

Элемент <div> "board" использует относительное позиционирование, а все вложенные элементы будут позиционироваться относительно этого элемента <div>.

Область сообщений размещается в левом верхнем углу игрового поля.

Элемент <div> области сообщений вложен в элемент <div> игрового поля, поэтому его позиция задается относительно последнего. Итак, он будет смещен на 0px по вертикали и на 0px по горизонтали относительно левого верхнего угла элемента <div> игрового поля.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Свойство "position: relative" позиционирует элемент относительно его нормальной позиции в потоке страницы.
- Свойство "position: absolute" позиционирует элемент относительно позиции его ближайшего родителя.
- Свойства top и left задают величину смещения элемента (в пикселах) относительно его позиции по умолчанию.

Мы также можем разместить таблицу и форму в элементе <div> “board”, снова используя абсолютное позиционирование для размещения этих элементов в точности там, где они должны находиться. Остаток кода CSS выглядит так:

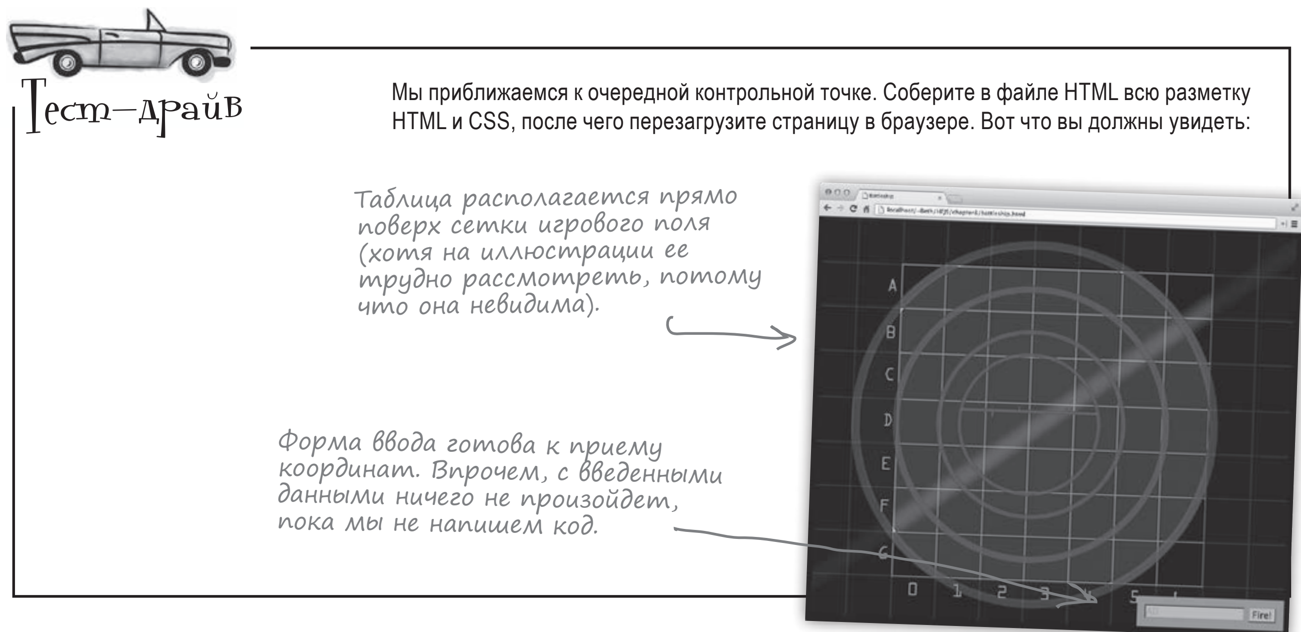
```
body {
    background-color: black;
}
div#board {
    position: relative;
    width: 1024px;
    height: 863px;
    margin: auto;
    background: url("board.jpg") no-repeat;
}
div#messageArea {
    position: absolute;
    top: 0px;
    left: 0px;
    color: rgb(83, 175, 19);
}
table {
    position: absolute;
    left: 173px;
    top: 98px;
    border-spacing: 0px;
}
td {
    width: 94px;
    height: 94px;
}
form {
    position: absolute;
    bottom: 0px;
    right: 0px;
    padding: 15px;
    background-color: rgb(83, 175, 19);
}
form input {
    background-color: rgb(152, 207, 113);
    border-color: rgb(83, 175, 19);
    font-size: 1em;
}
```

Элемент <table> смещается на 173 пиксела по горизонтали от левого края и на 98 пикселов по вертикали от верхнего края игрового поля, поэтому таблица выравнивается по сетке фонового изображения.

Каждый элемент <td> имеет четко определенную ширину и высоту, так что ячейки таблицы выравниваются по клеткам сетки.

Элемент <form> будет отображаться в правом нижнем углу игрового поля. Он немного скрывает числа внизу справа, но это не страшно (вы все равно их знаете). Также элементу <form> назначается приятный зеленый цвет, соответствующий цвету фонового изображения.

Наконец, к двум элементам <input> применяются стили, чтобы они соответствовали оформлению игрового поля, — и работа закончена!

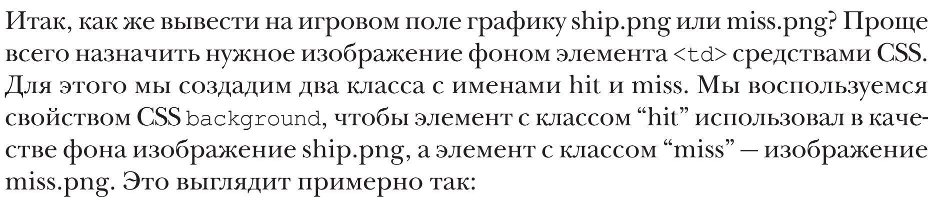


Мы приближаемся к очередной контрольной точке. Соберите в файле HTML всю разметку HTML и CSS, после чего перезагрузите страницу в браузере. Вот что вы должны увидеть:

Шаг 4: Размещение меток промахов и попаданий

Игровое поле выглядит превосходно, вы не находите? Однако нам еще нужно придумать, как размещать на нем визуальные признаки промахов и попаданий — то есть как добавить изображение ship.png или miss.png в соответствующем месте. Сейчас мы только выясним, как написать правильную разметку или стиль для решения этой задачи, а позднее мы используем тот же прием в коде.

Итак, как же вывести на игровом поле графику ship.png или miss.png? Проще всего назначить нужное изображение фоном элемента `<td>` средствами CSS. Для этого мы создадим два класса с именами hit и miss. Мы воспользуемся свойством CSS `background`, чтобы элемент с классом "hit" использовал в качестве фона изображение ship.png, а элемент с классом "miss" — изображение miss.png. Это выглядит примерно так:

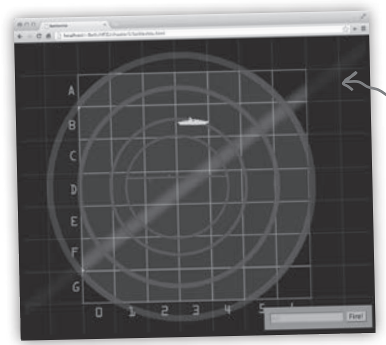


Если элемент относится к классу hit, ему назначается фоновое изображение ship.png. Если же элемент относится к классу miss, то ему назначается фоновое изображение miss.png.

```
.hit {
    background: url("ship.png") no-repeat center center;
}
.miss {
    background: url("miss.png") no-repeat center center;
}
```

Каждое правило CSS размещает в выбранном элементе одно изображение, выровненное по центру.

Использование классов hit и miss



Добавьте определения классов hit и miss в CSS. Как мы собираемся использовать эти классы, спросите вы? Проведем небольшой эксперимент: допустим, корабль занимает клетки “B3”, “B4” и “B5”, пользователь стреляет в клетку B3 — попадание! Следовательно, нужно разместить в B3 изображение ship.png. Для этого сначала “B” преобразуется в число (A соответствует 0, B соответствует 1, и так далее), после чего в таблице находится элемент `<td>` с идентификатором “13”. В найденный элемент `<td>` добавляется класс “hit”:

В элемент <td> добавляется класс “hit”.

```
<tr>
<td id="10"></td> <td id="11"></td> <td id="12"></td> <td id="13" class="hit"></td>
<td id="14"></td> <td id="15"></td> <td id="16"></td>
</tr>
```

Не забудьте добавить классы hit и miss с предыдущей страницы в CSS.

Теперь при перезагрузке страницы в клетке игрового поля “B3” выводится корабль.

То, что мы увидим при добавлении класса “hit” к элементу с идентификатором “13”.

Учебные стрельбы

Прежде чем писать код размещения попаданий и промахов на игровом поле, давайте немного потренируемся в работе с CSS. Добавьте в свою разметку классы “hit” и “miss”, соответствующие приведенным ниже действиям игрока. Обязательно проверьте свой ответ!

```
Корабль 1: A6, B6, C6
Корабль 2: C4, D4, E4
Корабль 3: B0, B1, B2
```

Напоминаем, что буквы нужно преобразовать в цифры: A = 0, ... G = 6.

А это выстрелы игрока:

```
A0, D4, F5, B2, C5, C6
```

Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

Когда упражнение будет выполнено, удалите все классы, добавленные к элементам <td>. Когда мы перейдем к программированию, игровое поле должно быть пустым.

В: Разве можно использовать строку из одних цифр как значение атрибута id в таблицах?

О: Да. В HTML5 идентификаторы элементов могут состоять из одних цифр. Если значение id не содержит ни одного пробела, это разрешено. В приложении «Морской бой» числовые идентификаторы идеально подходят для отслеживания координат и позволяют легко и быстро обратиться к элементу в любой позиции.

В: Уточним для ясности: значит, каждый элемент td используется как ячейка игрового поля, а попадание или промах отмечаются атрибутом class?

О: Верно, решение состоит из нескольких частей: фоновое изображение с сеткой (только для того, чтобы игровое поле хорошо смотрелось), наложенная на него прозрачная таблица HTML и классы hit и miss для размещения графических маркеров в ячейках по ходу игры. Вся последняя часть будет реализована исключительно на программном уровне, а классы будут динамически добавляться к элементам.

В: Похоже, нам потребуется преобразовывать буквы (например, «A6») в цифры («06»). Сможет ли JavaScript автоматически выполнить такое преобразование?

О: Нет, это нам придется проделать самостоятельно, но существует простой способ — мы воспользуемся тем, что вы знаете о массивах, для выполнения простого и быстрого преобразования.

В: Я малость подзабыл, как работает позиционирование CSS.

О: Механизм позиционирования позволяет задать точную позицию элемента. Если элемент позиционируется в режиме relative, то его размещение определяется относительно его нормального размещения в потоке макетирования страницы. Если элемент позиционируется в режиме absolute, то он размещается в конкретной позиции относительно ближайшего родителя. Иногда это вся страница — в этом случае, например, можно разместить в левом верхнем углу окна браузера. В нашем случае элементы таблицы и области сообщений позиционируются абсолютно, но относительно игрового поля (потому что игровое поле является ближайшим родителем таблицы и области сообщений).

Если вам понадобится более подробная информация о позиционировании CSS, обратитесь к главе 11 книги *Изучаем HTML, XHTML и CSS*.

В: Когда я изучал элемент form HTML, меня учили, что существует атрибут action, который выполняет отправку формы. Почему у нас его нет?

О: Атрибут action в элементе <form> не нужен, потому что мы не отправляем форму приложению на стороне сервера. В нашей игре все происходящее обрабатывается в браузере на программном уровне. Вместо отправки формы мы реализуем обработчик события, который оповещается о нажатии кнопки, и когда это происходит, мы обрабатываем в коде все, включая получение данных от пользователя. Обратите внимание: кнопка формы определяется с типом button, а не submit, к которому вы, возможно, привыкли при отправке данных программе PHP или другой программе, работающей на сервере Это хороший вопрос, мы еще вернемся к этой теме позднее в этой главе.

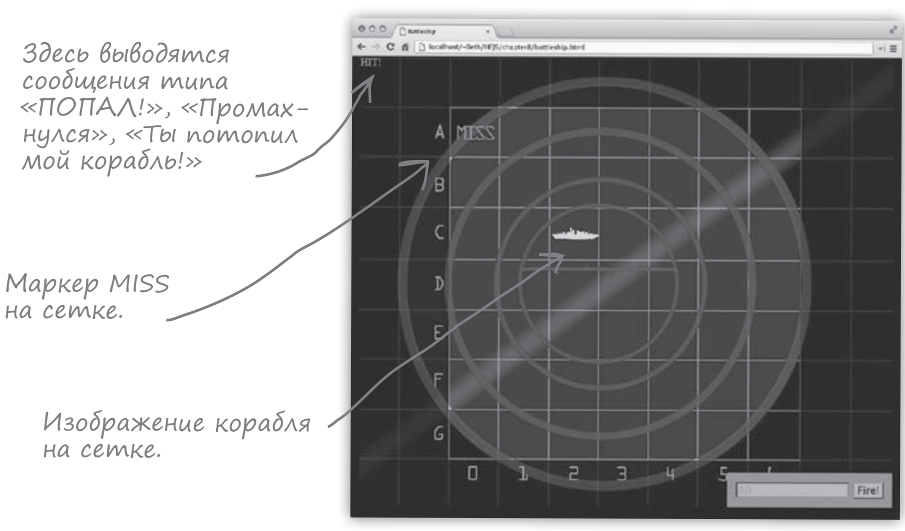
Как спроектировать игру

Разобравшись с HTML и CSS, вернемся к проектированию игры. В главе 2 мы не рассматривали функции, объекты и инкапсуляцию и еще не имели дела с объектно-ориентированным проектированием, поэтому при построении первой версии игры «Морской бой» использовался процедурный подход — игра проектировалась как последовательность выполняемых действий, с логикой принятия решений и циклов. Тогда вы ничего не знали о DOM, поэтому игра была недостаточно интерактивной. В новой реализации игра будет представлять собой набор объектов, обладающих определенными обязанностями, а взаимодействие с пользователем будет осуществляться через DOM. Вы увидите, насколько этот подход упрощает задачу.

Начнем с объектов, которые нам предстоит спроектировать и реализовать. Таких объектов будет три: *модель* для хранения состояния игры (например, позиций кораблей и попаданий); *представление*, ответственное за обновление изображения; и *контроллер*, связывающий все воедино (обработка пользовательского ввода, обеспечение отработки игровой логики, проверка завершения игры и т. д.).

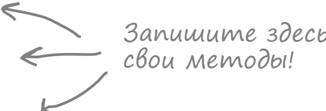
Упражнение

Пора заняться объектным проектированием. Мы начнем с объекта представления. Вспомните, что объект представления отвечает за обновление представления. Взгляните на иллюстрацию — удастся ли вам выбрать методы, относящиеся к объекту представления? Напишите объявления этих методов (только объявления, код тела методов будет приведен позднее) вместе с небольшим описанием действий каждого метода. Мы заполнили одно описание за вас. ***Сверьтесь с ответами, прежде чем двигаться дальше.***




```
var view = {

    // метод получает строковое сообщение и выводит его
    // в области сообщений
    displayMessage: function(msg) {
            // код будет приведен позднее!
    }
```

Мы определяем объект и присваиваем его переменной view.




```
};
```

Реализация представления

А если нет — вам должно быть стыдно! Сделайте это сейчас!

Если вы проверили ответ к предыдущему упражнению, то вы заметили, что код представления разбит на три метода: `displayMessage`, `displayHit` и `displayMiss`. Учтите, что единственно правильного ответа не существует. Например, можно ограничиться всего двумя методами `displayMessage` и `displayPlayerGuess` и передать `displayPlayerGuess` аргумент, указывающий, попал или промахнулся игрок. Такая архитектура тоже разумна. Но мы пока будем придерживаться нашей архитектуры... поэтому давайте подумаем над реализацией первого метода — `displayMessage`:

Наш объект представления.

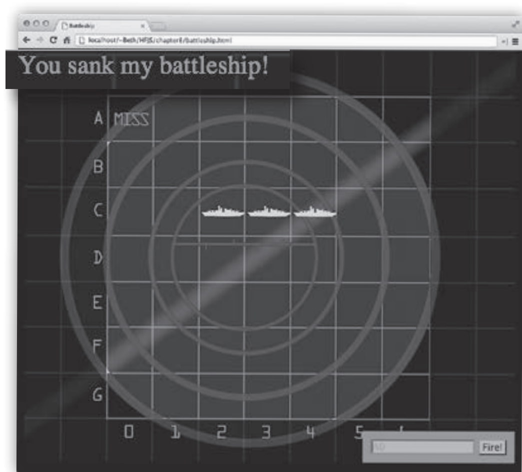
```
var view = {
   displayMessage: function(msg) {

   },
   displayHit: function(location) {
   },
   displayMiss: function(location) {
   }
};
```

Начнем отсюда.

Как работает метод displayMessage

Чтобы реализовать метод `displayMessage`, необходимо проанализировать разметку HTML и убедиться в том, что в ней имеется элемент `<div>` с идентификатором “messageArea”, готовый к выводу сообщений:

```
<div id="board">
  <div id="messageArea"></div>
   ...
</div>
```



Мы используем DOM для получения доступа к этому элементу `<div>`, после чего задаем его текст при помощи свойства `innerHTML`. И помните, что при каждом изменении DOM изменения немедленно отражаются в браузере. Вот что мы сделаем...

Это одно из преимуществ объектов. Мы можем работать над тем, чтобы объекты выполняли свои обязанности, не отвлекаясь на другие подробности программы. В нашем случае представлению достаточно знать, как обновлять область сообщений и как расставлять на сетке маркеры попаданий и промахов. Когда это поведение будет реализовано, работа с объектом представления завершится, и мы можем переходить к другим компонентам кода.

У такого подхода есть и другое преимущество: он позволяет провести изолированное тестирование представления и убедиться в том, что оно работает. При одновременном тестировании многих аспектов повышается вероятность того, что в работе программы проявятся проблемы, а также усложняется диагностика (потому что вам приходится проверять больше кода).

Чтобы протестировать изолированный объект (пока остальной код еще не готов), нужно написать небольшую тестовую программу, которая будет выброшена позднее, но в этом нет ничего страшного.

Итак, давайте завершим представление, протестируем его, а затем двинемся дальше!

Реализация displayMessage

Возвращаемся к написанию кода `displayMessage`. Напомним, что этот код должен:

- Использовать DOM для получения элемента с идентификатором “messageArea”.
- Задавать свойству `innerHTML` элемента сообщение, переданное методу `displayMessage`.

Итак, откройте пустой файл battleship.js и добавьте объект view:

```
var view = {
    displayMessage: function(msg) {
        var messageArea = document.getElementById("messageArea");
        messageArea.innerHTML = msg;
    },
    displayHit: function(location) {
    },
    displayMiss: function(location) {
    }
};
```

Метод displayMessage получает один аргумент — текст сообщения.

Мы получаем элемент messageArea из страницы...

...и обновляем текст элемента messageArea, задавая его свойству innerHTML содержимое msg.

Прежде чем переходить к тестированию, напишите два других метода. В них нет ничего сложного, а мы зато сможем проверить сразу весь объект.

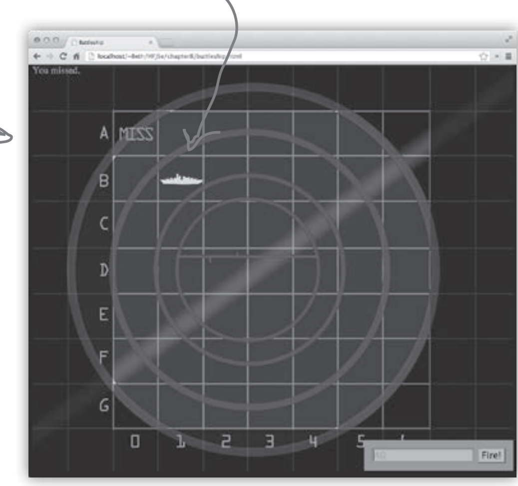
Как работают методы displayHit и displayMiss

Вспомните, о чем говорилось совсем недавно: чтобы изображение появилось на игровом поле, нужно найти элемент <td> и добавить к нему класс “hit” или “miss”. В первом случае в ячейке выводится содержимое ship.png, а во втором — содержимое miss.png.

Мы можем изменить изображение, добавляя к элементам <td> класс “hit” или “miss”. Остается понять, как сделать это в программном коде.

```
<tr>
<td id="10"></td> <td class="hit" id="11"></td> <td id="12"></td> ...
</tr>
```

В коде мы воспользуемся DOM для получения элемента `<td>`, после чего зададим его атрибуту class значение “hit” или “miss” при помощи метода `setAttribute`. Как только атрибут изменится, соответствующее изображение появится в браузере. Итак, вот что мы собираемся сделать:



- Получить строковый идентификатор из двух цифр, определяющих координаты клетки для вывода маркера промаха/попадания.
- Использовать DOM для получения элемента с полученным идентификатором.
- Задать атрибуту class этого элемента значение “hit” (для метода `displayHit`) или “miss” (для метода `displayMiss`).

Реализация displayHit и displayMiss

Оба метода, `displayHit` и `displayMiss`, получают аргумент location, определяющий ячейку для вывода маркера (попадания или промаха). Аргумент должен содержать идентификатор ячейки (элемента `<td>`) в таблице, представляющей игровое поле в HTML. Итак, прежде всего необходимо получить ссылку на этот элемент методом `getElementById`. Попробуем сделать это в методе `displayHit`:

```
displayHit: function(location) {
    var cell = document.getElementById(location);
},
```

Напоминаем: значение location образуется из строки и столбца и совпадает с идентификатором элемента <td>.

Следующим шагом станет добавление класса "hit" к элементу ячейки. Для этого мы можем воспользоваться методом `setAttribute`:

```
displayHit: function(location) {
    var cell = document.getElementById(location);
    cell.setAttribute("class", "hit");
},
```

Элементу назначается класс "hit". При этом в ячейке таблицы немедленно появляется изображение корабля.

Добавим этот код в объект представления, а также напишем аналогичный метод `displayMiss`:

```
var view = {
    displayMessage: function(msg) {
        var messageArea = document.getElementById("messageArea");
        messageArea.innerHTML = msg;
    },

    displayHit: function(location) {
        var cell = document.getElementById(location);
        cell.setAttribute("class", "hit");
    },

    displayMiss: function(location) {
        var cell = document.getElementById(location);
        cell.setAttribute("class", "miss");
    }
};
```

Идентификатор, созданный по введенным пользователем координатам, используется для получения обновляемого элемента.

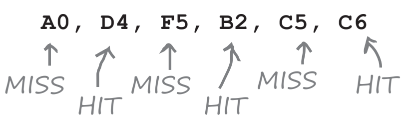
Этому элементу назначается класс "hit".

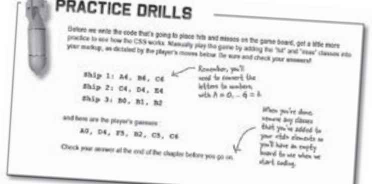
То же самое делается в методе displayMiss, только элементу назначается класс "miss", отвечающий за отображение промаха на игровом поле.

Не забудьте добавить код методов `displayHit` и `displayMiss` в файл battleship.js.

Следующий тест-драйв...

Давайте проверим, как работает этот код, прежде чем двигаться дальше... Мы возьмем выстрелы из упражнения «Учебные стрельбы» и реализуем их в программном коде. Вот как выглядит последовательность, которую мы хотим реализовать:





Чтобы реализовать эту последовательность в коде, добавьте следующий фрагмент в конец файла battleship.js:

```
view.displayMiss("00");
view.displayHit("34");
view.displayMiss("55");
view.displayHit("12");
view.displayMiss("25");
view.displayHit("26");
```

"A0", "D4", "F5", "B2", "C5", "C6"

Напомним, что методы displayHit и displayMiss получают координаты выстрела, уже преобразованные из комбинации "буква+цифра" в строку из двух цифр, соответствующую идентификатору одной из ячеек таблицы.

И не забудьте протестировать `displayMessage`:

```
view.displayMessage("Tap tap, is this thing on?");
```

Для простого тестирования годится любое сообщение...

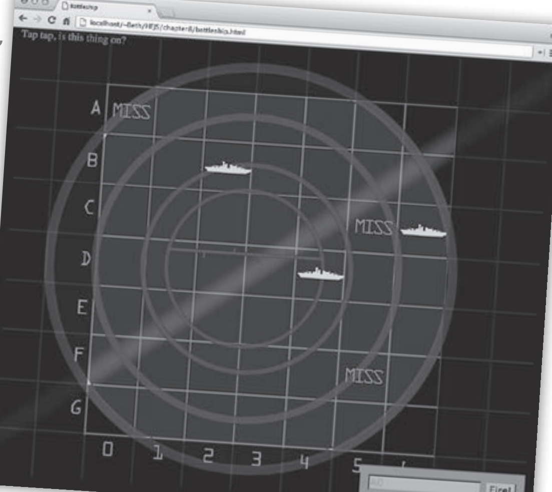
Когда это будет сделано, перезагрузите страницу в браузере и посмотрите, как изменился ее внешний вид.

Одно из преимуществ разбиения кода на объекты и закрепления одной задачи за каждым объектом — это возможность протестировать каждый объект и убедиться, что он правильно выполняет свои обязанности.

Сообщение выводится в левом верхнем углу игрового поля.

Попадания и промахи, для отображения которых используется объект представления, появляются на игровом поле.

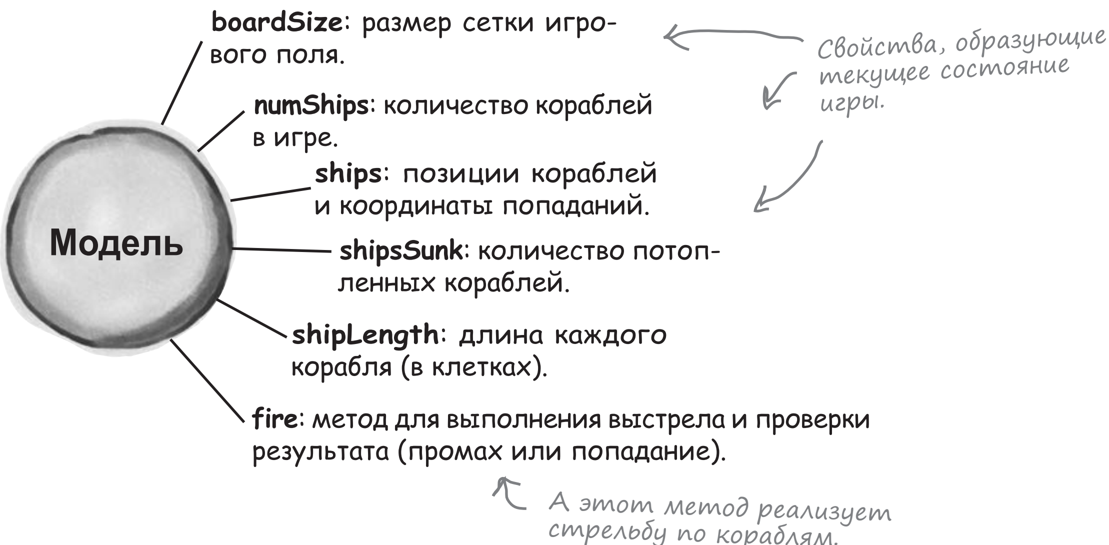
Проверьте каждый маркер и убедитесь в том, что он выводится в нужном месте.



Модель

Разобравшись с объектом представления, перейдем к модели. В объекте модели хранится текущее состояние игры. Модель также часто содержит логику, связанную с изменениями состояния. В нашем случае состояние включает позиции кораблей, координаты попаданий и счетчик потопленных кораблей. Пока вся необходимая логика ограничится проверкой того, попал ли выстрел игрока в корабль, и пометкой попадания.

Объект модели должен выглядеть примерно так:

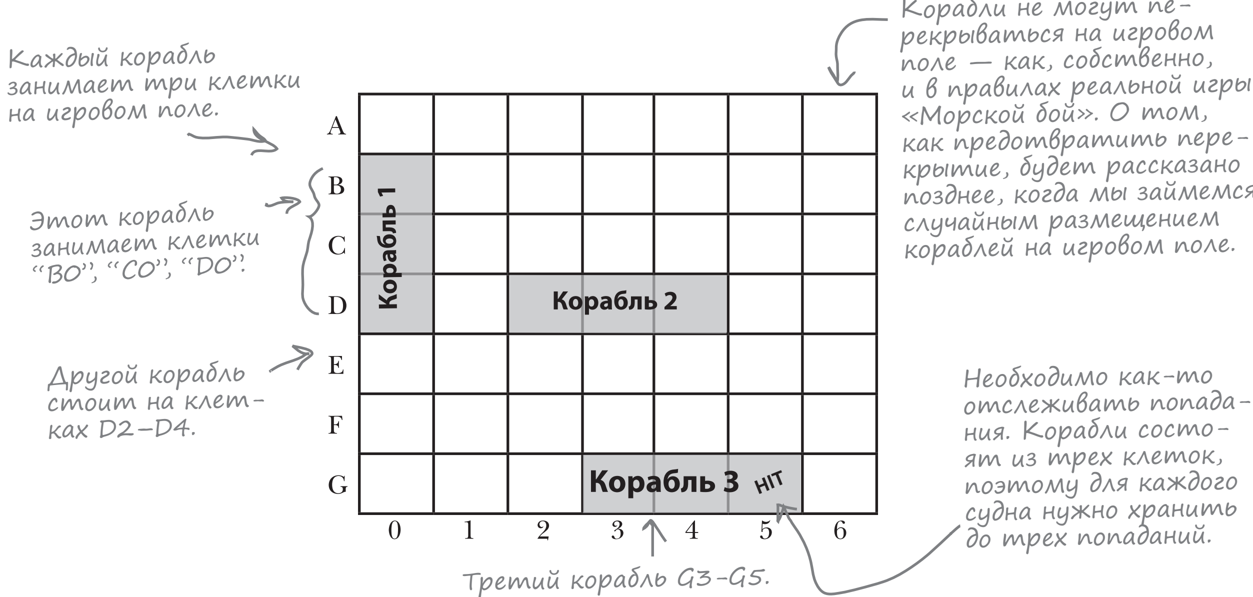


Как модель взаимодействует с представлением

При изменении состояния игры (то есть после выстрела с попаданием или промахом) представление должно обновить изображение в браузере. Для этого модель взаимодействует с представлением; к счастью, у нас есть методы, которые помогут организовать такие взаимодействия. Начнем с определения логики в модели, а потом добавим код для обновления представления.

Нам нужны большие корабли... и большое игровое поле

Прежде чем писать код модели, следует подумать над представлением состояния кораблей в модели. В упрощенной версии «Морского боя» из главы 2 был всего один корабль, который располагался на игровом поле 1×7. Теперь ситуация усложнилась: в игре *три* корабля на поле 7×7. Вот как это выглядит:



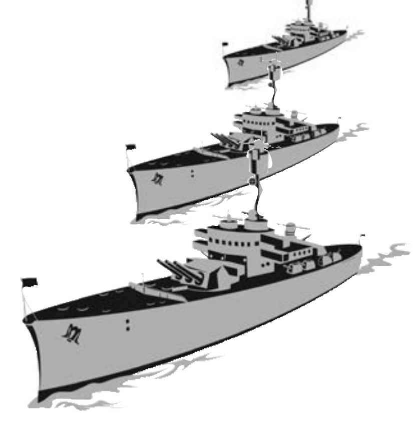
Возьми в руку карандаш

Как бы вы представили корабли в модели для приведенного выше игрового поля (пока только позиции самих кораблей, попаданиями займемся позже)? Выберите лучшее решение из представленных ниже.

- ❑ Использовать девять переменных (по аналогии с тем, как мы представляли позиции кораблей в главе 2).
- ❑ Использовать массив с элементами, представляющими клетки игрового поля (49 элементов). В элементе каждой клетки, содержащей часть корабля, хранится номер корабля.
- ❑ Использовать массив для хранения всех девяти позиций. В элементах 0–2 хранятся данные первого корабля, в элементах 3–5 — данные второго, и т. д.
- ❑ Использовать три разных массива, по одному для каждого корабля. Каждый массив содержит данные трех клеток.
- ❑ Использовать объект с именем ship и тремя свойствами (для каждой из трех клеток). Объекты ship хранятся в массиве с именем ships.
- ❑ ______________________________

Или запишите собственный вариант.

Как мы будем представлять данные кораблей



Существует много способов организации данных кораблей, более того, вы наверняка можете предложить несколько своих вариантов. Какие бы данные вы ни использовали, их всегда можно организовать по-разному, причем у каждого способа есть «плюсы» и «минусы» — одни способы будут эффективны по затратам памяти, другие оптимизируют время выполнения, третьи более понятны программисту и т. д.

Мы выбрали относительно простой способ — каждый корабль представляется объектом, хранящим координаты занятых клеток и количество попаданий. Рассмотрим один из таких объектов:

```
var ship1 = {
          locations: ["10", "20", "30"],
          hits: ["", "", ""]
        };
```

Каждый корабль представляется объектом.

Объект содержит два свойства, locations и hits.

В свойстве locations хранится массив всех клеток, занимаемых кораблем.

Свойство hits содержит массив с информацией о попаданиях выстрелов в клетки. Изначально элементы массива инициализируются пустой строкой и заменяются строкой "hit" при попадании в соответствующую клетку.

Обратите внимание: координаты представляются двумя цифрами (0 = A, 1 = B и т. д.).

Так должны выглядеть все три корабля:

Каждый объект содержит массив location и массив hits. Каждый массив содержит три элемента.

```
var ship1 = { locations: ["10", "20", "30"], hits: ["", "", ""] };
var ship2 = { locations: ["32", "33", "34"], hits: ["", "", ""] };
var ship3 = { locations: ["63", "64", "65"], hits: ["", "", "hit"] };
```

Вместо того чтобы создавать три разные переменные для хранения информации о кораблях, мы создадим один массив для хранения всех данных:

В имени используется множественное число: ships.

Переменной ships присваивается массив, в котором хранятся данные всех трех кораблей.

```
var ships = [{ locations: ["10", "20", "30"], hits: ["", "", ""] },
             { locations: ["32", "33", "34"], hits: ["", "", ""] },
             { locations: ["63", "64", "65"], hits: ["", "", "hit"] }];
```

Первый корабль...

...второй...

...и третий.

Обратите внимание: у этого корабля было попадание в клетке "65".

Развлечения с магнитами

Расставьте на игровом поле магниты с маркерами попаданий и промахов для следующей последовательности ходов и структуры данных кораблей. Удалось ли игроку потопить все корабли? Мы сделали первый ход за вас.

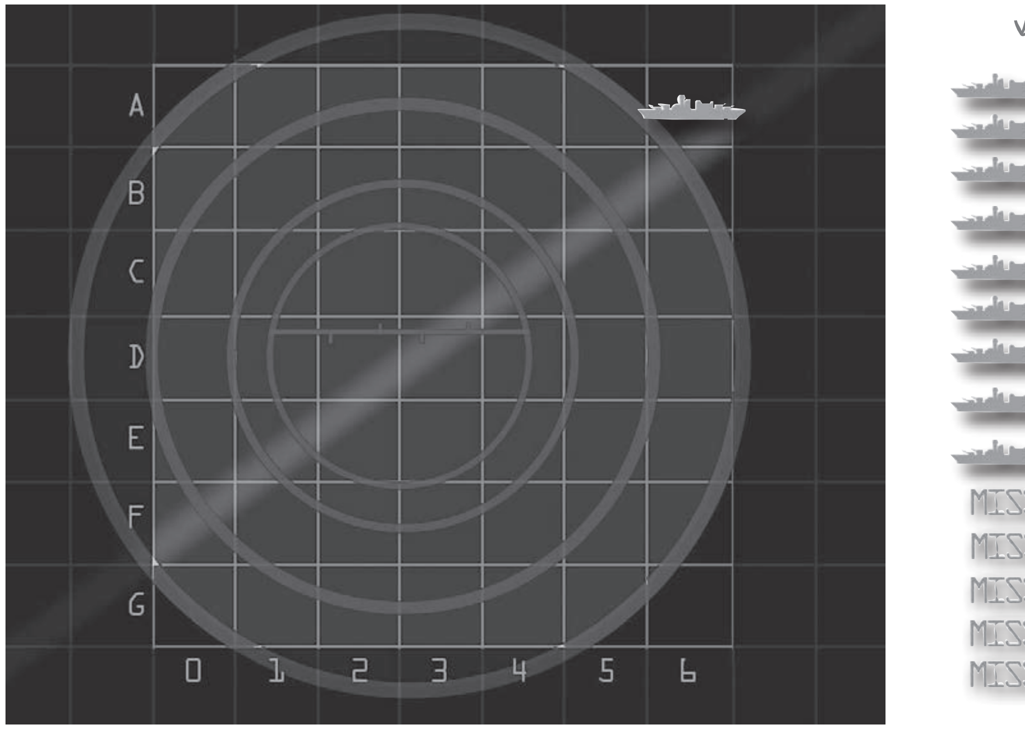
Координаты выстрелов:

A6, B3, C4, D1, B0, D4, F0, A1, C6, B1, B2, E4, B6 ← Реализуйте эти ходы на игровом поле.

```
var ships = [{ locations: ["06", "16", "26"], hits: ["hit", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] }];
```

Структура данных. После серии выстрелов могут появиться новые маркеры попаданий.

Игровое поле и магниты.



У вас могут остаться лишние магниты.

Возьми в руку карандаш

Давайте потренируемся в использовании структуры данных ships и смоделируем серию операций. По приведенному ниже определению ships найдите ответы на следующие вопросы и заполните пропуски. Обязательно проверьте свои ответы, прежде чем следовать дальше, потому что это очень важная часть игрового механизма:

```
var ships = [{ locations: ["31", "41", "51"], hits: ["", "", ""] },
             { locations: ["14", "24", "34"], hits: ["", "hit", ""] },
             { locations: ["00", "01", "02"], hits: ["hit", "", ""] }];
```

Какие корабли уже были «подстрелены»?_____________ В каких позициях? _______________

Игрок стреляет по клетке "D4", попадет ли он в корабль?_______ Если да, то в какой? ___________

Игрок стреляет по клетке "B3", попадет ли он в корабль?_______ Если да, то в какой? ___________

Допишите следующий код, чтобы он определял позицию средней клетки корабля и выводил ее методом console.log:

```
var ship2 = ships[____];
var locations = ship2.locations;
console.log("Location is " + locations[____]);
```

Допишите следующий код, чтобы он определял, было ли попадание в первой клетке третьего корабля:

```
var ship3 = ships[____];
var hits = ship3._____;
if (_____ === "hit") {
   console.log("Ouch, hit on third ship at location one");
}
```

Допишите следующий код, чтобы он записывал попадание в третью клетку первого корабля:

```
var ______ = ships[0];
var hits = ship1._______;
hits[____] = ________;
```

Реализация объекта модели

Итак, мы разобрались, как в программе будут представляться корабли и попадания, теперь можно переходить к коду. Сначала мы создадим объект модели, а затем возьмем структуру данных `ships` и добавим ее как свойство. Раз уж мы этим занялись, нам понадобятся и другие свойства, например `numShips` для информации о количестве кораблей в игре. Возможно, вы удивитесь: «Ведь мы знаем, что в игре три корабля, — зачем нужно свойство `numShips`?» Представьте, что вам захочется создать новую, более сложную игру с четырьмя или пятью кораблями? Отказываясь от фиксированного значения и используя свойство (и указывая в коде свойство вместо конкретного значения), мы избавляемся от будущих проблем, потому что изменения будет достаточно внести в одном месте.

Кстати, раз уж мы заговорили о «фиксированных значениях» — исходные позиции кораблей будут фиксированными... пока. Точная информация о положении кораблей упростит тестирование и позволит сосредоточиться на логике. Код случайного размещения кораблей на игровом поле будет рассмотрен чуть позже.

Модель

boardSize: размер сетки игрового поля.

numShips: количество кораблей в игре.

ships: позиции кораблей и координаты попаданий.

shipsSunk: количество потопленных кораблей.

shipLength: длина каждого корабля (в клетках).

fire: метод для выполнения выстрела и проверки результата (промах или попадание).

Итак, создадим объект модели:

Объект модели.

```
var model = {
   boardSize: 7,
   numShips: 3,
   shipLength: 3,
   shipsSunk: 0,

   ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },
           { locations: ["24", "34", "44"], hits: ["", "", ""] },
           { locations: ["10", "11", "12"], hits: ["", "", ""] }]

};
```

Эти три свойства помогают обойтись без жестко фиксированных значений: boardSize (размер игрового поля), numShips (количество кораблей в игре) и shipLength (длина корабля в клетках, 3).

Свойство shipsSunk (инициализируется значением 0 в начале игры) содержит текущее количество кораблей, потопленных игроком.

В игре накопилось довольно много данных состояния!

Позднее позиции кораблей будут генерироваться случайным образом, но пока мы зафиксируем их в программе, чтобы упростить тестирование.

Свойство ships содержит массив объектов ship, содержащих массивы locations и hits для одного из трех кораблей. (Обратите внимание, что использовавшаяся ранее переменная ships заменена свойством объекта модели).

Обратите внимание: размеры массивов locations и hits тоже фиксируются. О том, как создавать массивы динамически, будет рассказано позднее.

Что должен делать метод fire

Метод `fire` преобразует выстрел игрока в промах или попадание. Мы знаем, что за маркеры промахов и попаданий отвечает объект представления `view`, но `fire` должен предоставить игровую логику определения результата выстрела (промах или попадание).

Определить, попал ли выстрел в корабль, несложно:

- Проверить каждый корабль и узнать, занимает ли он указанную клетку.
- Если клетка присутствует в списке позиций, значит, выстрел попал в цель. Программа помечает соответствующий элемент массива `hits` (и сообщает представлению о попадании). Метод возвращает `true` — признак попадания.
- Если указанная клетка не занята кораблем, значит, игрок промахнулся. Мы сообщаем об этом представлению и возвращаем `false`.

Метод `fire` должен определять, не был ли корабль потоплен, но этим мы займемся чуть позже.

Модель

boardSize: размер сетки игрового поля.

numShips: количество кораблей в игре.

ships: позиции кораблей и координаты попаданий.

shipsSunk: количество потопленных кораблей.

shipLength: длина каждого корабля (в клетках).

fire: метод для выполнения выстрела и проверки результата (промах или попадание).

Подготовка метода fire

Для начала набросаем заготовку метода `fire`. Метод получает аргумент с координатами выстрела, перебирает все корабли и проверяет, пришлось ли попадание в очередной корабль. Код проверки попаданий будет написан чуть позже, а пока подготовим все остальное:

```
var model = {
  boardSize: 7,
  numShips: 3,
  shipsSunk: 0,
  shipLength: 3,
  ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },
          { locations: ["24", "34", "44"], hits: ["", "", ""] },
          { locations: ["10", "11", "12"], hits: ["", "", ""] }],

  fire: function(guess) {

     for (var i = 0; i < this.numShips; i++) {
         var ship = this.ships[i];
     }
   }
};
```

Не забудьте добавить запятую!

Метод получает координаты выстрела.

Затем мы перебираем массив ships, последовательно проверяя каждый корабль.

Здесь мы получаем объект корабля. Необходимо проверить, совпадают ли координаты выстрела с координатами одной из занимаемых им клеток.

Проверка попаданий

Итак, при каждой итерации цикла необходимо проверить, присутствует ли указанная клетка в массиве locations объекта ship:

```
for (var i = 0; i < this.numShips; i++) {
    var ship = this.ships[i];
    locations = ship.locations;
    
}
```

Последовательно перебираем корабли.

Получаем массив клеток, занимаемых кораблем. Стоит напомнить, что это свойство корабля, в котором хранится массив.

Здесь должен находиться код, который проверяет, присутствует ли указанная клетка в массиве locations корабля.

Ситуация такова: имеется строка `guess` с координатами клетки, которая ищется в массиве `locations`. Если значение `guess` встречается в массиве, значит, выстрел попал в корабль:

```
guess = "16";
locations = ["06", "16", "26"];
```

Мы должны определить, совпадает ли значение в guess с одним из значений из массива locations объекта корабля.

В принципе, можно было бы написать еще один цикл, который перебирает все элементы массива `locations` и сравнивает каждый элемент с `guess`; если значения совпадают, значит, выстрел попал в цель.

Но существует и другое, более простое решение:

```
var index = locations.indexOf(guess);
```

Метод indexOf ищет в массиве указанное значение и возвращает его индекс (или -1, если значение отсутствует в массиве).

С методом `indexOf` код проверки попаданий может выглядеть так:

```
for (var i = 0; i < this.numShips; i++) {
    var ship = this.ships[i];
    locations = ship.locations;
    var index = locations.indexOf(guess);
    if (index >= 0) {
        // Есть попадание!
    }
}
```

Обратите внимание на сходство метода indexOf массива с методом indexOf строки. Оба метода получают значение и возвращают индекс (или -1, если значение не найдено).

Если полученный индекс больше либо равен нулю, значит, указанная клетка присутствует в массиве location и выстрел попал в цель.

Решение с `indexOf` не эффективнее простого цикла, но оно чуть более понятно и явно более компактно. С первого взгляда становится очевидно, какое именно значение ищется в массиве с использованием `indexOf`. В любом случае в вашем инструментарии появился новый полезный инструмент.

А теперь все вместе...

Напоследок нужно определиться еще с одним моментом: что должно происходить, когда выстрел попадает в корабль? В текущей версии мы просто отмечаем попадание в модели, что означает добавление строки "hit" в массив `hits`.

Давайте объединим все, о чем говорилось ранее:

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],

     fire: function(guess) {
         for (var i = 0; i < this.numShips; i++) {
             var ship = this.ships[i];
             var locations = ship.locations;
             var index = locations.indexOf(guess);
             if (index >= 0) {
                 ship.hits[index] = "hit";
                 return true;
             }
         }
         return false;
     }
};
```

Для каждого корабля...

Если координаты клетки присутствуют в массиве locations, значит, выстрел попал в цель.

Ставим отметку в массиве hits по тому же индексу.

А еще нужно вернуть true, потому что выстрел оказался удачным.

Если же после перебора всех кораблей попадание так и не обнаружено, игрок промахнулся, а метод возвращает false.

Это отличное начало для работы над объектом модели. Осталось сделать еще пару вещей: определить, был ли корабль потоплен, и сообщить представлению об изменении модели, чтобы оно могло обновить изображение. Посмотрим, как это сделать...

Я же просил выражаться покороче!

Нам снова приходится поднимать эту тему. Дело в том, что наши обращения к объектам и массивам получаются громоздкими. Присмотритесь к коду:

```
for (var i = 0; i < this.numShips; i++) {
  var ship = this.ships[i];               ← Получаем объект ship...
  var locations = ship.locations;         ← Потом массив locations из объекта ship...
  var index = locations.indexOf(guess);   ← Затем индекс клетки в locations.
  ...
}
```

Пожалуй, этот код получился слишком длинным. Почему? Некоторые ссылки можно сократить посредством сцепления — ссылки на объекты объединяются в цепочки, позволяющие избежать создания временных переменных (таких, как переменная `locations` в приведенном коде).

Почему переменная `locations` названа временной? Потому что она используется только для промежуточного хранения массива `ship.locations`, чтобы мы могли вызвать метод `indexOf` для получения индекса `guess`. Ни для чего больше переменная `locations` в этом методе не используется. Сцепление позволяет нам избавиться от временной переменной `locations`:

```
var index = ship.locations.indexOf(guess);   ← Две выделенные строки объединены в одну строку.
```

Как работает сцепление...

Сцепление в действительности представляет собой сокращенную форму записи для обращения к свойствам и методам объектов (и массивов). Давайте поближе присмотримся к тому, что было сделано для сцепления двух команд.

```
var ship = { locations: ["06", "16", "26"], hits: ["", "", ""] };   ← Объект ship.
var locations = ship.locations;   ← Извлекаем массив locations из объекта ship.
var index = locations.indexOf(guess);   ← И используем его для вызова метода indexOf.
```

Две последние команды можно объединить в цепочку (избавляясь от временной переменной `locations`):

```
ship.locations.indexOf(guess)
```

1. Получаем объект ship.
2. Объект обладает свойством locations, которое является массивом.
3. И поддерживает метод indexOf.

Тем временем на корабле...

Теперь мы напишем код, который будет проверять, потоплен ли корабль. Правила вам известны: корабль потоплен, когда выстрелы попали во все его клетки. Мы добавим вспомогательный метод для проверки этого условия:

Метод с именем isSunk получает объект корабля и возвращает true, если корабль потоплен, или false, если он еще держится на плаву.

Метод получает объект корабля и проверяет, помечены ли все его клетки маркером попадания.

```
isSunk: function(ship) {
    for (var i = 0; i < this.shipLength; i++)  {
        if (ship.hits[i] !== "hit") {
            return false;
        }
    }
    return true;
}
```

Если есть хотя бы одна клетка, в которую еще не попал игрок, то корабль еще жив и метод возвращает false.

А если нет — корабль потоплен! Метод возвращает true.

Добавьте этот метод в объект модели, сразу же за методом fire.

Теперь мы можем использовать этот метод в методе `fire`, чтобы проверить, был ли корабль потоплен:

```
fire: function(guess) {
    for (var i = 0; i < this.numShips; i++) {
        var ship = this.ships[i];
        var index = ship.locations.indexOf(guess);
        if (index >= 0) {
            ship.hits[index] = "hit";
            if (this.isSunk(ship)) {
                this.shipsSunk++;
            }
            return true;
        }
    }
    return false;
},
isSunk: function(ship) { ... }
```

Мы добавим проверку здесь, после того как будем точно знать, что выстрел попал в корабль. Если корабль потоплен, то мы увеличиваем счетчик потопленных кораблей в свойстве shipsSunk модели.

Новый метод isSunk добавляется сразу же после fire. И не забудьте, что все свойства и методы модели должны разделяться запятыми!

Взаимодействие с представлением

Вот, собственно, и все, что нужно сказать об объекте модели. Модель содержит состояние игры и логику проверки попаданий и промахов. Не хватает только кода, который бы оповещал представление о попаданиях или промахах в модели. Давайте напишем его:

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipsSunk: 0,
    shipLength: 3,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],
    fire: function(guess) {
        for (var i = 0; i < this.numShips; i++) {
            var ship = this.ships[i];
            var index = ship.locations.indexOf(guess);
            if (index >= 0) {
                ship.hits[index] = "hit";
                view.displayHit(guess);
                view.displayMessage("HIT!");
                if (this.isSunk(ship)) {
                    view.displayMessage("You sank my battleship!");
                    this.shipsSunk++;
                }
                return true;
            }
        }
        view.displayMiss(guess);
        view.displayMessage("You missed.");
        return false;
    },
    isSunk: function(ship) {
        for (var i = 0; i < this.shipLength; i++)  {
            if (ship.hits[i] !== "hit") {
                return false;
            }
        }
        return true;
    }
};
```

Мы приводим полный код объекта модели, чтобы вы могли окинуть взглядом всю картину.

Оповещаем представление о том, что в клетке guess следует вывести маркер попадания.

И приказываем представлению вывести сообщение "HIT!".

Сообщаем игроку, что он потопил корабль!

Сообщаем представлению, что в клетке guess следует вывести маркер промаха.

И приказываем представлению вывести сообщение о промахе.

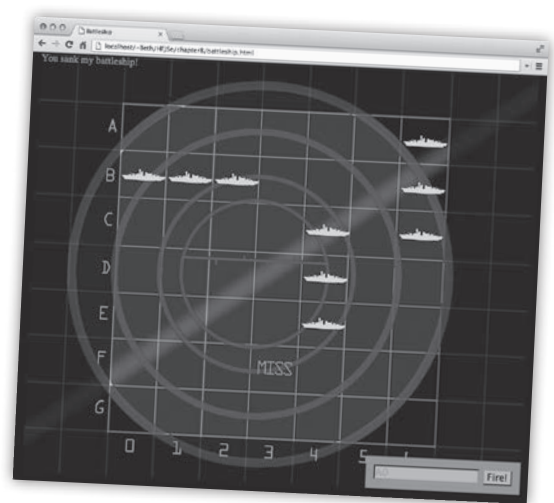
Методы объекта представления добавляют класс "hit" или "miss" к элементу с идентификатором, содержащимся в guess. Таким образом, представление преобразует «попадания» из массива hits в разметку HTML. Не забывайте, что HTML-«попадания» нужны для отображения информации, а «попадания» в модели представляют фактическое состояние.

Тест-драйв

Добавьте весь код модели в файл battleship.js. Протестируйте его методом fire модели, передавая строку и столбец из guess. Позиции кораблей до сих пор жестко фиксированы, поэтому вам будет легко их подбить. Добавьте собственные выстрелы (несколько дополнительных промахов). (Чтобы увидеть нашу версию кода, загрузите файл battleship_tester.js.)

Чтобы получить такие же результаты, вам придется удалить или закомментировать предыдущий код тестирования. В файле battleship_tester.js показано, как это делается.

```
model.fire("53");

model.fire("06");
model.fire("16");
model.fire("26");

model.fire("34");
model.fire("24");
model.fire("44");

model.fire("12");
model.fire("11");
model.fire("10");
```



Перезагрузите страницу battleship.html. Маркеры промахов и попаданий должны появиться на игровом поле.

Часто задаваемые вопросы

В: Объединение команд посредством сцепления лучше серии раздельных команд?

О: Не обязательно. Выигрыш по эффективности от сцепления невелик (экономится всего одна переменная), но код становится короче. Мы считаем, что короткие цепочки (не длиннее двух или трех уровней) читаются лучше нескольких строк кода, но это наше личное мнение. Если вы предпочитаете разделять команды по строкам, это нормально. А если вы применяете сцепление, следите за тем, чтобы цепочки не были слишком длинными; их будет труднее читать и понять.

В: Мы используем массивы (locations) в объектах (ship), хранящихся в массиве (ships). На какую глубину могут вкладываться объекты и массивы?

О: В общем-то на любую. Конечно, на практике слишком большая глубина вложения маловероятна (если в вашей программе будет использоваться больше трех–четырех уровней вложенности, то структуры данных станут слишком сложными, и их стоит переработать).

В: Я заметил, что в модель добавлено свойство boardSize, но в коде модели оно не используется. Для чего оно нужно?

О: Свойство boardSize и другие свойства модели будут использоваться в будущем коде. Модель отвечает за управление состоянием игры, а boardSize определенно является частью состояния. Контроллер обращается к нужному состоянию через свойства модели; позднее мы добавим и другие методы модели, в которых также будут использоваться эти свойства.

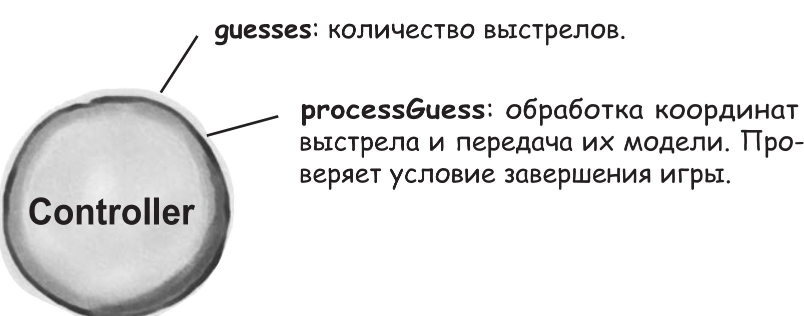
Реализация контроллера

После завершения работы над представлением и моделью можно переходить к реализации контроллера. На верхнем уровне контроллер связывает все остальные компоненты, получая координаты выстрела guess, обрабатывая их и передавая модели. Кроме того, он отслеживает некоторые административные подробности — скажем, текущее количество выстрелов и текущие достижения игрока. В этом контроллер зависит от модели, поддерживающей состояние игры, и от представления, обеспечивающего отображение информации.

Если говорить конкретнее, контроллеру поручаются следующие обязанности:

- Получение и обработка координат выстрела (например, "A0" или "B1").
- Отслеживание количества выстрелов.
- Запрос к модели на обновление в соответствии с последним выстрелом.
- Проверка завершения игры (когда все корабли будут потоплены).

Наша работа над контроллером начинается с определения свойства `guesses` в объекте controller. Затем мы реализуем метод `processGuess`, который получает алфавитно-цифровые координаты выстрела, обрабатывает их и передает модели.



Заготовка кода контроллера; мы наполним ее кодом на нескольких ближайших страницах:

```
var controller = {
   guesses: 0,

   processGuess: function(guess) {
     // Код метода
   }
};
```

Здесь определяется объект контроллера со свойством guesses, которое инициализируется нулем.

Начало метода processGuess, получающего координаты в формате "A0".

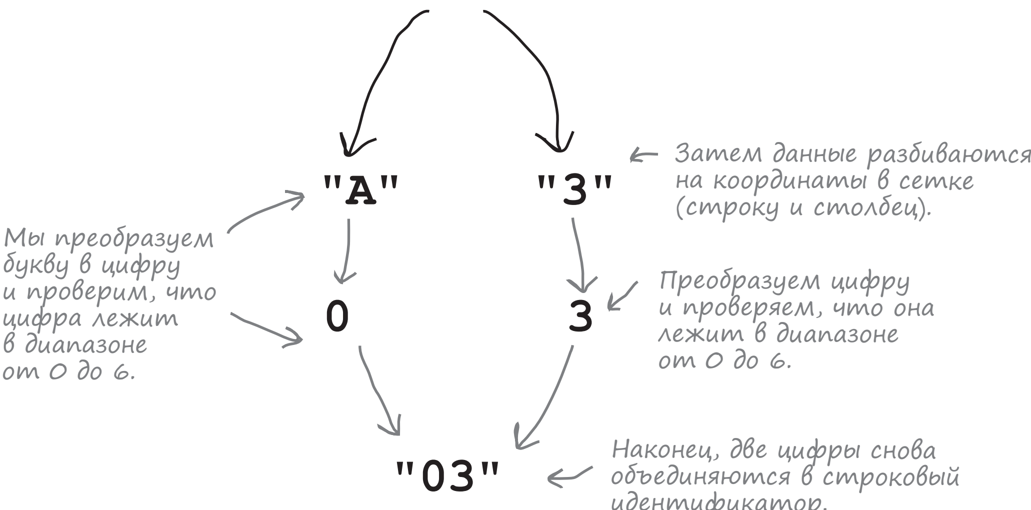
Обработка выстрела

Контроллер должен получить координаты выстрела, введенные пользователем, проверить их и передать объекту модели. Но откуда он получит данные? Не беспокойтесь, скоро мы ответим и на этот вопрос. А пока будем предполагать, что в коде в какой-то момент вызывается метод `processGuess` контроллера, которому передается строка в следующем формате:

Это очень полезный прием. Сосредоточьтесь на требованиях к конкретному коду. Попытки охватить всю задачу сразу часто оказываются менее эффективными.

Теперь после получения координат в этом формате (алфавитно-цифровая комбинация вида «A3») данные необходимо преобразовать в формат, понятный модели (комбинация из двух цифр, например «03»). Высокоуровневая схема преобразования входных данных в цифровой формат выглядит так:

Конечно, сознательный игрок никогда не введет некорректные данные, верно? Если бы! Лучше явно проверить полученные данные на правильность.



Впрочем, обо всем по порядку. Нам также необходимо проверить полученные данные на правильность. Давайте запланируем процедуру проверки, прежде чем писать код.

Планирование кода...

Вместо того чтобы размещать весь код обработки данных в методе `processGuess`, мы напишем небольшую вспомогательную функцию (которую, если потребуется, можно будет использовать повторно). Эта функция будет называться `parseGuess`.

Прежде чем переходить к написанию кода, давайте последовательно разберем, как должна работать функция:

1. Игрок вводит координаты в классическом стиле «Морского боя»: буква, за которой следует цифра.

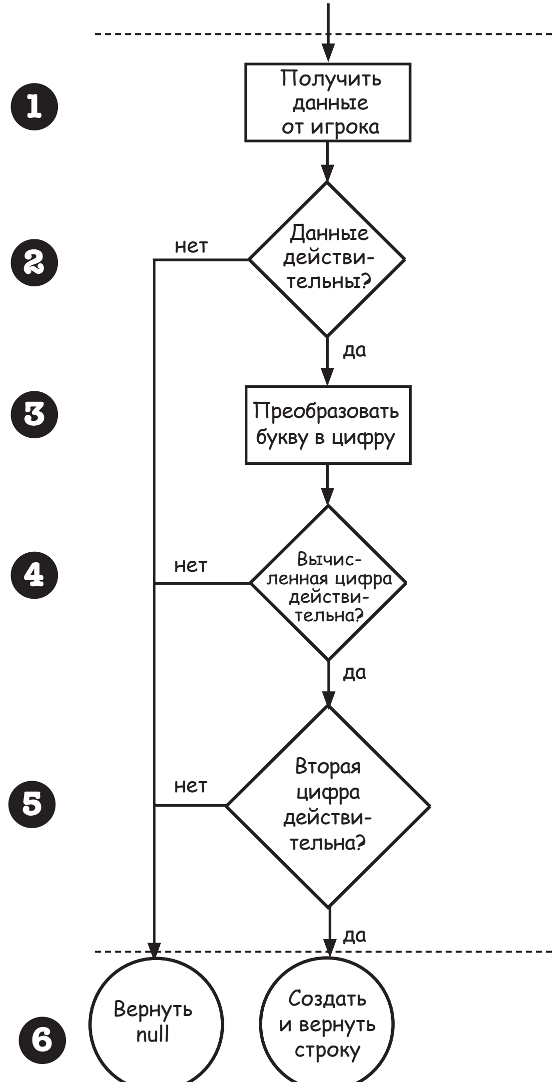
2. Данные проверяются на действительность (не null, не слишком короткие и не слишком длинные).

3. Получаем букву и преобразуем ее в цифру: A в 0, B в 1, и так далее.

4. Проверка цифры с шага 3 (в диапазоне от 0 до 6).

5. Проверка второй цифры (также в диапазоне от 0 до 6).

6. Если хотя бы одна проверка завершилась неудачей, вернуть null. В противном случае метод объединяет две цифры в строку и возвращает ее.



Реализация метода parseGuess

У нас очень серьезные планы, так что приступим:

Начнем с шагов 1 и 2. Все что нужно — получить координаты выстрела от игрока и проверить их на действительность. На данный момент это означает, что мы получаем строку, отличную от null и состоящую ровно из двух символов.

Данные передаются в параметре guess.

Проверяем данные на null и убеждаемся, что в строке два символа.

```
function parseGuess(guess) {
   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   }
}
```

Если условие не выполняется, сообщаем игроку.

3 Затем берем букву и преобразуем ее в цифру при помощи вспомогательного массива с буквами A-F. Для этого мы воспользуемся методом indexOf для получения индекса буквы в массиве:

Массив заполняется всеми буквами, которые могут присутствовать в действительных координатах.

```
function parseGuess(guess) {
   var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      firstChar = guess.charAt(0);
      var row = alphabet.indexOf(firstChar);
   }
}
```

Извлекаем первый символ строки.

При помощи метода indexOf получаем цифру в диапазоне от 0 до 6, соответствующую букве. Попробуйте выполнить пару примеров, чтобы понять, как работает преобразование.

А теперь проверим оба символа и узнаем, являются ли они цифрами в диапазоне от 0 до 6 (другими словами, определяют ли они действительную позицию на игровом поле).

```
function parseGuess(guess) {
   var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

   if (guess === null || guess.length !== 2) {
      alert("Oops, please enter a letter and a number on the board.");
   } else {
      firstChar = guess.charAt(0);
      var row = alphabet.indexOf(firstChar);
      var column = guess.charAt(1);

      if (isNaN(row) || isNaN(column)) {
         alert("Oops, that isn't on the board.");
      } else if (row < 0 || row >= model.boardSize ||
                     column < 0 || column >= model.boardSize) {
         alert("Oops, that's off the board!");
      }
   }
}
```

Здесь добавляется код для получения второго символа, представляющего столбец игрового поля.

А здесь функция isNaN выявляет строки и столбцы, которые не являются цифрами.

Мы также проверяем, что цифры лежат в диапазоне от 0 до 6.

На самом деле программа действует на более общем уровне. Вместо того чтобы жестко фиксировать цифру 6, мы запрашиваем размер доски у модели и используем его для сравнения.

Здесь применяются преобразования типов. Переменная column содержит строковое значение, и проверяя, что значение находится в диапазоне от 0 до 6, мы полагаемся на преобразование строки в число для выполнения сравнения.

Вместо того чтобы жестко фиксировать 6 как максимальное значение строки или столбца игрового поля, мы используем свойство boardSize модели. Как вы думаете, какие преимущества это решение принесет в долгосрочной перспективе?

6 Осталось дописать последний фрагмент функции parseGuess... Если какая-либо из проверок входных данных завершается неудачей, возвращается null. В противном случае возвращается идентификатор из номеров строки и столбца.

```
function parseGuess(guess) {
    var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

    if (guess === null || guess.length !== 2) {
        alert("Oops, please enter a letter and a number on the board.");
    } else {
        firstChar = guess.charAt(0);
        var row = alphabet.indexOf(firstChar);
        var column = guess.charAt(1);

        if (isNaN(row) || isNaN(column)) {
            alert("Oops, that isn't on the board.");
        } else if (row < 0 || row >= model.boardSize ||
                   column < 0 || column >= model.boardSize) {
            alert("Oops, that's off the board!");
        } else {
            return row + column;
        }
    }
    return null;
}
```

Строка и столбец объединяются, а результат возвращается методом. Здесь снова задействовано преобразование типа: row — число, а column — строка, поэтому результатом также является строка.

В этой точке все проверки пройдены, поэтому метод может вернуть результат.

Если управление передано в эту точку, значит, какая-то проверка не прошла, и метод возвращает null.

Тест-драйв

Включите приведенный код в файл battleship.js, затем добавьте следующие вызовы функций:

```
console.log(parseGuess("A0"));
console.log(parseGuess("B6"));
console.log(parseGuess("G3"));
console.log(parseGuess("H0"));
console.log(parseGuess("A7"));
```

Перезагрузите battleship.html и проследите за тем, чтобы окно консоли было открыто. На консоли должны появиться результаты вызовов parseGuess и, возможно, пара оповещений.

Консоль JavaScript

```
00
16
63
null
null
```

Тем временем в контроллере...

Разобравшись со вспомогательной функцией `parseGuess`, мы перейдем к реализации контроллера. Начнем с интеграции функции `parseGuess` с существующим кодом контроллера:

```
var controller = {
    guesses: 0,

    processGuess: function(guess) {
        var location = parseGuess(guess);
        if (location) {

        }
    }
};
```

Метод parseGuess будет использоваться для проверки введенных данных.

Если метод не возвращает null, значит, был получен действительный объект location.

Помните: null является псевдоистинным значением.

Здесь размещается остальной код контроллера.

С первой обязанностью контроллера мы разобрались. Посмотрим, что еще осталось:

- ~~Получение и обработка координат выстрела (например, “A0” или “B1”).~~
- Отслеживание количества выстрелов.
- Запрос к модели на обновление в соответствии с последним выстрелом.
- Проверка завершения игры (когда все корабли будут потоплены).

Следующий шаг.

Подсчет и обработка выстрелов

Следующий пункт списка вполне тривиален: чтобы подсчитать количество выстрелов, необходимо лишь увеличивать свойство `guesses` каждый раз, когда игрок делает очередной выстрел. Как вы вскоре увидите, мы решили не наказывать игрока за ввод недействительных данных.

Затем мы приказываем модели обновить себя по данным текущего выстрела посредством вызова метода `fire` модели. В конце концов, игрок для того и стреляет, чтобы попытаться попасть в корабль. Вспомните, что метод `fire` получает комбинацию строки и столбца игрового поля, и если не случится ничего непредвиденного, мы получим эту комбинацию вызовом `parseGuess`. Весьма удобно.

Соберем все вместе и реализуем следующий шаг...

```
var controller = {
    guesses: 0,

    processGuess: function(guess) {
        var location = parseGuess(guess);
        if (location) {
            this.guesses++;
            var hit = model.fire(location);
        }
    }
};
```

Если пользователь ввел правильные координаты, счетчик выстрелов увеличивается на 1.

Команда this.guesses++ просто увеличивает значение свойства guesses на 1. Она работает точно так же, как i++ в циклах for.

Затем комбинация строки и столбца передается методу fire. Напомним, что метод fire возвращает true при попадании в корабль.

Также заметьте, что при вводе недействительных координат мы не наказываем игрока и не включаем эту попытку в подсчет.

Игра окончена?

Осталось лишь определить, когда игра закончится. Как это сделать? Мы знаем, что для этого должны быть потоплены все три корабля. Каждый раз, когда выстрел игрока попадает в цель, мы будем проверять, потоплены ли три корабля, по свойству `model.shipsSunk`. Мы немного обобщим эту проверку, и вместо того чтобы сравнивать с числом 3, будем использовать для сравнения свойство `numShips` модели. Возможно, позднее вы захотите перейти на другое количество кораблей (скажем, 2 или 4), и тогда вам не придется переделывать код, чтобы он правильно работал.

```
var controller = {
    guesses: 0,

    processGuess: function(guess) {
        var location = parseGuess(guess);
        if (location) {
            this.guesses++;
            var hit = model.fire(location);
            if (hit && model.shipsSunk === model.numShips) {
                    view.displayMessage("You sank all my battleships, in " +
                                                    this.guesses + " guesses");
            }
        }
    }
};
```

Если выстрел попал в цель, а количество потопленных кораблей равно количеству кораблей в игре, выводится сообщение о том, что все корабли потоплены.

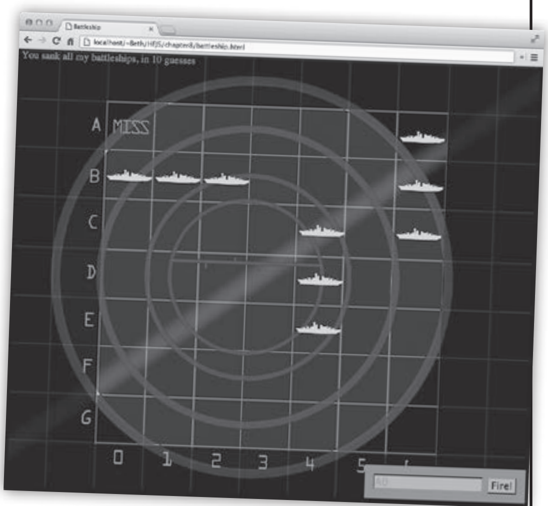
Выводим общее количество выстрелов, которое потребовалось игроку для того, чтобы потопить корабли. Свойство guesses является свойством объекта this, то есть контроллера.

Тест-драйв

Включите весь код контроллера в файл battleship.js и добавьте приведенные вызовы функций для тестирования контроллера. Перезагрузите battleship.html и обратите внимание на маркеры промахов и попаданий. Они в правильных местах? (Загрузите страницу battleship_tester.js, чтобы увидеть нашу версию.)

Вам придется удалить или закомментировать предыдущий тестовый код, чтобы получить показанные результаты. В файле battleship_tester.js показано, как это делается.

```
controller.processGuess("A0");

controller.processGuess("A6");
controller.processGuess("B6");
controller.processGuess("C6");

controller.processGuess("C4");
controller.processGuess("D4");
controller.processGuess("E4");

controller.processGuess("B0");
controller.processGuess("B1");
controller.processGuess("B2");
```



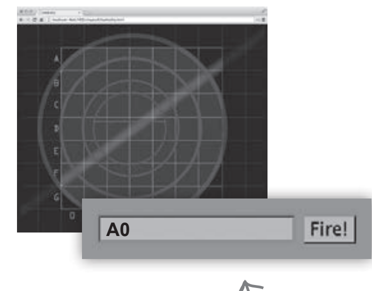
Мы вызываем метод processGuess контроллера и передаем ему координаты выстрелов в формате «Морского боя».

Мозговой штурм

Мы сообщаем игроку о завершении игры в области сообщений, когда он потопит все три корабля. Однако после этого игрок все равно может вводить координаты. Как сделать так, чтобы игрок не мог продолжать стрельбу после того, как все корабли будут потоплены?

Получение данных от игрока

Теперь, когда мы реализовали базовую логику игры и вывода данных, можно перейти к механизму ввода и получения координат выстрелов. Вероятно, вы помните, что в разметке HTML уже присутствует элемент `<form>`, готовый к вводу данных, но как связать его с игрой?



Элемент формы HTML, готовый к вводу данных.

Для этого нам понадобится обработчик событий, о которых уже говорилось ранее в этой книге. Сейчас мы проведем с ними ровно столько времени, сколько понадобится для того, чтобы игра заработала, а все технические подробности будут представлены в следующей главе. Наша цель — получить примерное представление о том, как обработчики событий работают с элементами форм, но не углубляясь в подробные описания.

Итак, начнем с общей картины:

1. Игрок вводит координаты и нажимает кнопку Fire!.
2. При нажатии кнопки Fire! вызывается заранее назначенный обработчик события.
3. Обработчик кнопки Fire! получает данные с формы и передает их контроллеру.

```
function handleFireButton() {
    // Получение координат от формы
    // и передача их контроллеру.
}
```

Как связать обработчик событий с кнопкой Fire!

Чтобы вся эта схема заработала, необходимо, прежде всего, связать обработчик событий с кнопкой Fire! Для этого мы сначала получим ссылку на кнопку по идентификатору. Просмотрите разметку HTML — вы увидите, что кнопке Fire! присвоен идентификатор "fireButton". С этой информацией достаточно вызвать метод `document.getElementById` для получения ссылки на кнопку. Наконец, при известной ссылке можно назначить функцию-обработчик свойству `onclick` кнопки:

Этот код необходимо где-то разместить. Мы создадим для него функцию init.

Сначала мы получаем ссылку на кнопку Fire! по идентификатору кнопки:

```
function init() {
    var fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
}
```

Кнопке можно назначить обработчик события нажатия — функцию handleFireButton.

И не забудьте создать заготовку функции handleFireButton:

Это функция handleFireButton. Она будет вызываться при каждом нажатии кнопки Fire!.

```
function handleFireButton() {
    // Код получения данных от формы
}
```

Вскоре мы напишем код этой функции.

```
window.onload = init;
```

Как было показано в главе 6, браузер должен выполнять init при полной загрузке страницы.

Получение координат от формы

Кнопка Fire! запускает обработку выстрела, но сами введенные данные содержатся в элементе формы "guessInput". Для получения значения от формы можно обратиться к свойству `value` элемента input. Вот как это делается:

Сначала мы получаем ссылку на элемент формы по идентификатору элемента, "guessInput".

```
function handleFireButton() {
    var guessInput = document.getElementById("guessInput");
    var guess = guessInput.value;
}
```

Затем извлекаем данные, введенные пользователем. Координаты хранятся в свойстве value элемента input.

Значение получили, теперь с ним нужно что-то сделать. К счастью, для этого у нас уже имеется немало готового кода. Добавим его на следующем шаге.

Передача данных контроллеру

Пора собрать все воедино. Контроллер ожидает — просто-таки не может дождаться — получения выстрела от игрока. От нас потребуется лишь передать ему введенные данные. Давайте сделаем это:

```
function handleFireButton() {
    var guessInput = document.getElementById("guessInput");
    var guess = guessInput.value;
    controller.processGuess(guess);

    guessInput.value = "";
}
```

Координаты выстрела передаются контроллеру, а дальше все должно заработать как по волшебству!

Короткая команда просто удаляет содержимое элемента input формы, заменяя его пустой строкой. Это делается для того, чтобы вам не приходилось многократно выделять текст и удалять его перед вводом следующего выстрела.

Тест-драйв

Это не обычный тест-драйв — наконец-то все готово для полноценной игры! Убедитесь в том, что весь код добавлен в файл battleship.js, и перезагрузите battleship.html в браузере. Так как позиции кораблей жестко фиксированы, победить будет нетрудно. Ниже приведены координаты выигрышных выстрелов, но вам следует провести более полное тестирование кода. Дополните серию промахами и недействительными комбинациями.

```
A6
B6
C6

C4
D4
E4

B0
B1
B2
```

Это выигрышная серия выстрелов, упорядоченная по кораблям. Но вам не обязательно вводить их в указанном порядке. Попробуйте изменить порядок, введите пару недействительных комбинаций координат. Также добавьте несколько промахов. Все это является частью качественного тестирования.

A6 Fire!

Для любознательных

Не нравится щелкать на кнопке Fire? Да, этот способ работает, но он медленный и неудобный. Гораздо проще нажимать клавишу Enter (Ввод), не правда ли? Простой фрагмент кода для обработки нажатия Enter:

```
function init() {
    var fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
    var guessInput = document.getElementById("guessInput");
    guessInput.onkeypress = handleKeyPress;
}
```

Добавляем новый обработчик — для обработки событий нажатия клавиш в поле ввода HTML.

Обработчик нажатий клавиш; вызывается при каждом нажатии клавиши в поле input страницы.

Браузер передает объект события обработчику. Объект содержит информацию о том, какая клавиша была нажата.

```
function handleKeyPress(e) {
    var fireButton = document.getElementById("fireButton");
    if (e.keyCode === 13) {
        fireButton.click();
        return false;
    }
}
```

Если нажата клавиша Enter, то свойство keyCode события равно 13. В таком случае кнопка Fire! должна сработать так, словно игрок щелкнул на ней. Для этого можно вызвать метод click кнопки fireButton (фактически этот вызов имитирует нажатие кнопки).

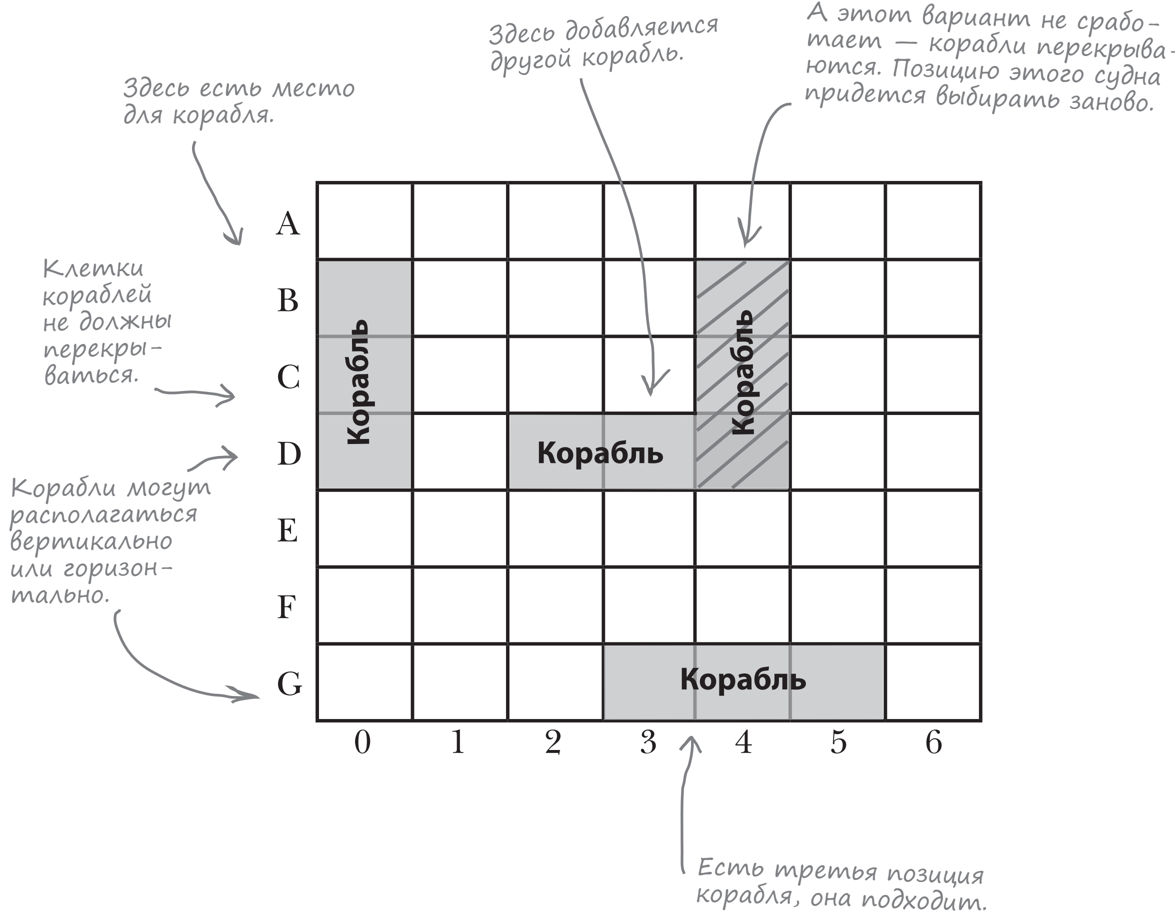
И мы возвращаем false, чтобы форма не делала ничего лишнего (например, не пыталась передать данные).

Обновите функцию init и добавьте функцию handleKeyPress в любой точке кода. Перезагрузите страницу — и открывайте огонь!

Что еще осталось? Ах да, фиксированные позиции кораблей!

К настоящему моменту мы создали довольно впечатляющую браузерную игру из небольшой разметки HTML, нескольких изображений и приблизительно 100 строк кода. Но у этой игры есть одна неприятная особенность: корабли всегда будут на постоянных позициях. Мы должны написать код, генерирующий случайные позиции кораблей при каждом запуске игры (иначе играть в нее будет попросту скучно).

Прежде чем начинать, хотим сообщить, что код будет рассматриваться в более высоком темпе — вы уже знаете достаточно, чтобы лучше читать и понимать код, а нового в нем не так уж много. Итак, за дело! Вот что нужно реализовать:



Развлечения с магнитами

Блоки алгоритма, генерирующего позиции кораблей, перепутались. Сможете ли вы разместить магниты в правильной последовательности, чтобы создать работоспособный алгоритм? Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

Алгоритм — красивое слово, которым обозначается последовательность действий для решения задачи.

Сгенерировать случайную позицию для нового корабля.

Выполнить цикл по количеству создаваемых кораблей.

Сгенерировать случайное направление (вертикальное или горизонтальное) для нового корабля.

Добавить позиции нового корабля в массив ships.

Проверить, не перекрываются ли клетки нового корабля с клетками каких-либо существующих кораблей.

Как размещать корабли

При размещении кораблей на игровом поле необходимо учитывать два обстоятельства. Во-первых, корабли могут располагаться как вертикально, так и горизонтально. Во-вторых, корабли не должны перекрываться на игровом поле. Основная часть кода, который мы напишем, будет связана с обработкой этих ограничений. Как уже говорилось ранее, мы не будем сейчас разбирать этот код во всех подробностях; вы знаете все, что для этого необходимо, и, потратив достаточно времени, сможете понять каждую часть во всех подробностях. Здесь нет ничего такого, чего бы вы не видели ранее (за одним исключением, о котором будет сказано особо). Итак, за дело...

Код будет оформлен в виде трех методов объекта модели:

- **generateShipLocations**: основной метод. Создает в модели массив `ships` с количеством кораблей, определяемым свойством `numShips` модели.
- **generateShip**: метод создает один корабль, находящийся в произвольном месте игрового поля. При этом не исключено перекрытие с другими кораблями.
- **collision**: метод получает один корабль и проверяет, что тот не перекрывается с кораблями, уже находящимися на игровом поле.

Функция generateShipLocations

Начнем с метода `generateShipLocations`. Этот метод в цикле создает корабли, пока массив ships модели не будет заполнен достаточным количеством кораблей. Каждый раз, когда метод генерирует новый корабль (что делается методом `generateShip`), он использует метод `collision` для проверки возможных перекрытий. Если корабль перекрывается с другими кораблями, метод отказывается от него и делает следующую попытку.

В этом коде следует обратить внимание на новую разновидность циклов — do while. Цикл do while работает почти так же, как while, за одним исключением: он *сначала выполняет* команды в теле цикла, а *потом* проверяет условие. Некоторые логические условия (хотя это бывает не так часто) лучше подходят для циклов do while, чем для циклов while.

Этот метод добавляется в объект модели.

```
generateShipLocations: function() {
    var locations;
    for (var i = 0; i < this.numShips; i++) {
        do {
            locations = this.generateShip();
        } while (this.collision(locations));
        this.ships[i].locations = locations;
    }
},
```

Для каждого корабля генерируется набор позиций, то есть занимаемых клеток.

Генерируем новый набор позиций...

...и проверяем, перекрываются ли эти позиции с существующими кораблями на доске. Если есть перекрытия, нужна еще одна попытка. Новые позиции генерируются, пока не будут найдены варианты без перекрытий.

Здесь используется цикл do while!

Полученные позиции без перекрытий сохраняются в свойстве locations объекта корабля в массиве model.ships.

Метод generateShip

Генерация случайного числа 0 или 1 — аналог подбрасывания монетки.

Метод `generateShip` создает массив со случайными позициями корабля, не беспокоясь о возможных перекрытиях. Выполнение метода состоит из двух шагов. На первом шаге случайным образом выбирается направление: будет корабль располагаться горизонтально или вертикально. Если число равно 1, то корабль располагается горизонтально; если число равно 0, то корабль располагается вертикально. Для генерирования случайных чисел, как и прежде, будут использоваться знакомые нам методы `Math.random` и `Math.floor`:

Этот метод также добавляется в объект модели.

```
generateShip: function() {
    var direction = Math.floor(Math.random() * 2);
    var row, col;

    if (direction === 1) {
        // Сгенерировать начальную позицию для горизонтального корабля
    } else {
        // Сгенерировать начальную позицию для вертикального корабля
    }

    var newShipLocations = [];
    for (var i = 0; i < this.shipLength; i++) {
        if (direction === 1) {
            // добавить в массив для горизонтального корабля
        } else {
            // добавить в массив для вертикального корабля
        }
    }
    return newShipLocations;
},
```

При помощи Math.random мы генерируем число от 0 до 1 и умножаем результат на 2, чтобы получить число в диапазоне от 0 до 2 (не включая 2). Затем Math.floor преобразует результат в 0 или 1.

Если значение direction равно 1, создается горизонтальный корабль...

...а для значения 0 создается вертикальный корабль.

Сначала создается начальная позиция нового корабля (например, строка = 0 и столбец = 3). В зависимости от направления начальная позиция должна создаваться по разным правилам (вскоре вы поймете, почему).

Набор позиций нового корабля начинается с пустого массива, в который последовательно добавляются элементы.

В цикле до количества позиций в корабле.

...при каждой итерации новая позиция добавляется в массив newShipLocations. И снова позиция будет генерироваться разным кодом в зависимости от направления корабля.

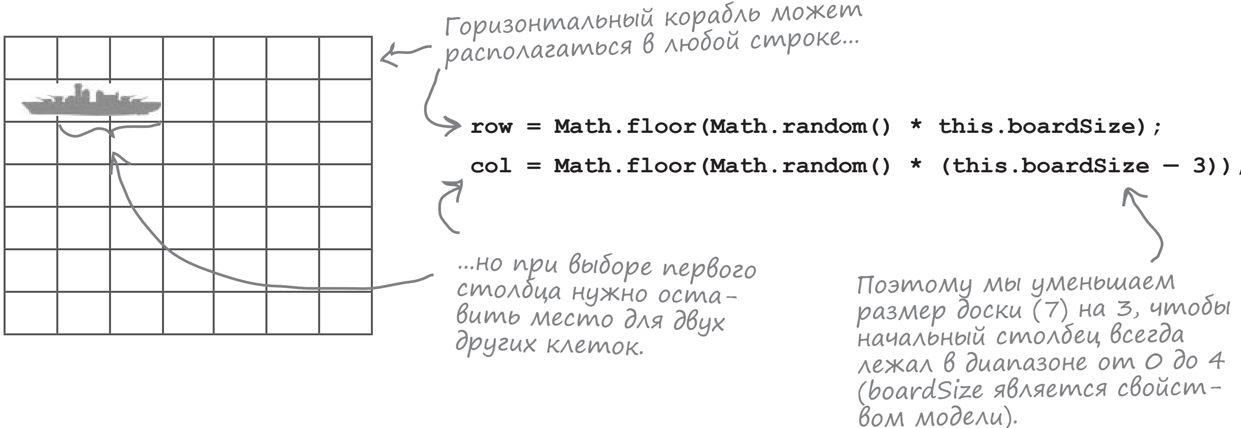
Когда все позиции сгенерированы, метод возвращает массив.

Мы продолжим дописывать код на следующей странице...

Генерирование начальной позиции нового корабля

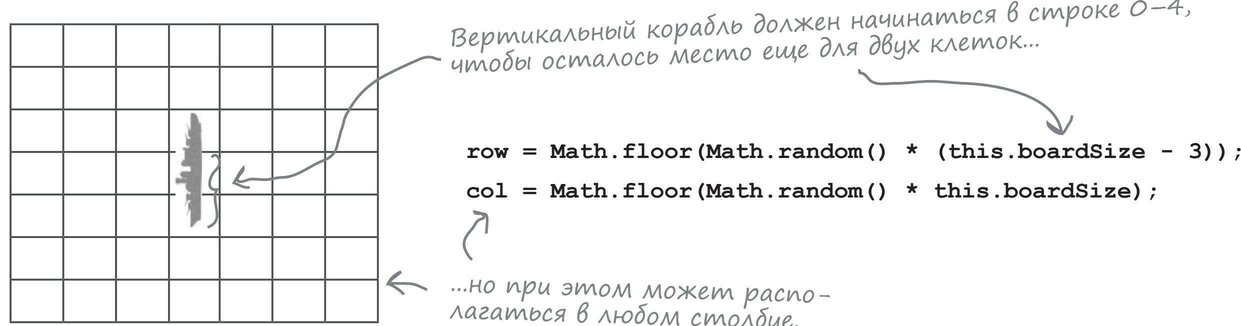
Зная ориентацию корабля на игровом поле, мы можем сгенерировать набор занимаемых им клеток. Сначала генерируется начальная позиция (первая), а остальные позиции будут просто находиться в двух соседних столбцах (при горизонтальном расположении) или строках (при вертикальном расположении).

Для этого необходимо сгенерировать два случайных числа — строку и столбец начальной позиции. Оба числа должны лежать в диапазоне от 0 до 6, чтобы корабль поместился на игровом поле. Но если корабль будет располагаться *горизонтально*, начальный *столбец* должен лежать в диапазоне от 0 до 4, чтобы корабль не вышел за пределы игрового поля:




```
row = Math.floor(Math.random() * this.boardSize);
col = Math.floor(Math.random() * (this.boardSize — 3));
```

По тому же принципу, если корабль будет располагаться вертикально, начальная *строка* должна лежать в диапазоне от 0 до 4, чтобы оставить место для остальных клеток:




```
row = Math.floor(Math.random() * (this.boardSize - 3));
col = Math.floor(Math.random() * this.boardSize);
```

Завершение метода generateShip

После включения этого кода остается лишь добавить начальную позицию вместе с двумя следующими позициями в массив `newShipLocations`.

```
generateShip: function() {
    var direction = Math.floor(Math.random() * 2);
    var row, col;

    if (direction === 1) {
        row = Math.floor(Math.random() * this.boardSize);
        col = Math.floor(Math.random() * (this.boardSize - this.shipLength));
    } else {
        row = Math.floor(Math.random() * (this.boardSize - this.shipLength));
        col = Math.floor(Math.random() * this.boardSize);
    }

    var newShipLocations = [];
    for (var i = 0; i < this.shipLength; i++) {
        if (direction === 1) {
            newShipLocations.push(row + "" + (col + i));
        } else {
            newShipLocations.push((row + i) + "" + col);
        }
    }
    return newShipLocations;
},
```

Этот код генерирует начальную позицию корабля на игровом поле.

Значение 3 (с предыдущей страницы) заменено на this.shipLength для обобщения кода, чтобы его можно было использовать с произвольной длиной корабля.

Код для горизонтального корабля. Разберем его на составляющие...

Круглые скобки гарантируют, что значение i будет прибавлено к col _до_ преобразования результата в строку.

Новая позиция заносится в массив newShipLocations.

Данные состоят из строки (начальной, вычисленной выше)...

...и столбца + i. При первой итерации i равно 0, и сумма обозначает начальный столбец. При второй итерации происходит переход к следующему столбцу, а при третьей — к следующему за ним. Так в массиве генерируются серии элементов “01”, “02”, “03”.

То же самое для вертикального корабля.

Теперь вместо столбца увеличивается строка — при каждой итерации цикла к ней прибавляется i.

Для вертикального корабля в массиве создается серия вида “31”, “41”, “51”.

Не забывайте: при сложении строки с числом знак «+» выполняет конкатенацию, а не сложение, поэтому результат представляет собой строку.

Заполнив массив позициями нового корабля, мы возвращаем его вызывающему методу generateShipLocations.

Как избежать столкновений

Чтобы увидеть, где вызывается метод collision, вернитесь к странице 388.

Метод `collision` получает данные корабля и проверяет, перекрывается ли хотя бы одна клетка с клетками других кораблей, уже находящихся на поле.

Проверка реализована двумя вложенными циклами for. Внешний цикл перебирает все корабли модели (в свойстве `model.ships`). Внутренний цикл перебирает все позиции нового корабля в массиве `locations` и проверяет, не заняты ли какие-либо из этих клеток существующими кораблями на игровом поле.

locations — массив позиций нового корабля, который мы собираемся разместить на игровом поле.

```
collision: function(locations) {
    for (var i = 0; i < this.numShips; i++) {
        var ship = model.ships[i];
        for (var j = 0; j < locations.length; j++) {
            if (ship.locations.indexOf(locations[j]) >= 0) {
                return true;
            }
        }
    }
    return false;
}
```

Для каждого корабля, уже находящегося на поле...

...проверить, встречается ли какая-либо из позиций массива locations нового корабля в массиве locations существующих кораблей.

Метод indexOf проверяет, присутствует ли заданная позиция в массиве. Таким образом, если полученный индекс больше либо равен 0, мы знаем, что клетка уже занята, поэтому метод возвращает true (перекрытие обнаружено).

Возврат из цикла, выполняемого в другом цикле, немедленно прерывает оба цикла, функция немедленно завершается и возвращает true.

Если выполнение дошло до этой точки, значит, ни одна из позиций не была обнаружена в других массивах, поэтому функция возвращает false (перекрытия отсутствуют).

МОЗГОВОЙ ШТУРМ

В этом коде задействованы два цикла: внешний цикл перебирает все корабли в модели, а внутренний цикл перебирает все позиции, проверяемые на перекрытие. Во внешнем цикле используется управляющая переменная i, а во внутреннем — переменная j. Почему мы используем два разных имени переменных?

Два последних изменения

Мы написали весь код, необходимый для генерирования случайных позиций кораблей; остается лишь интегрировать его. Внесите два последних изменения — и новую версию «Морского боя» можно отправлять на тестирование!

```
var model = {
    boardSize: 7,
    numShips: 3,
    shipLength: 3,
    shipsSunk: 0,
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },
             { locations: ["24", "34", "44"], hits: ["", "", ""] },
             { locations: ["10", "11", "12"], hits: ["", "", ""] } ],
    ships: [ { locations: [0, 0, 0], hits: ["", "", ""] },
             { locations: [0, 0, 0], hits: ["", "", ""] },
             { locations: [0, 0, 0], hits: ["", "", ""] } ],
    fire: function(guess) { ... },
    isSunk: function(ship) { ... },
    generateShipLocations: function() { ... },
    generateShip: function() { ... },
    collision: function(locations) { ... }
};

function init() {
    var fireButton = document.getElementById("fireButton");
    fireButton.onclick = handleFireButton;
    var guessInput = document.getElementById("guessInput");
    guessInput.onkeypress = handleKeyPress;

    model.generateShipLocations();
}
```

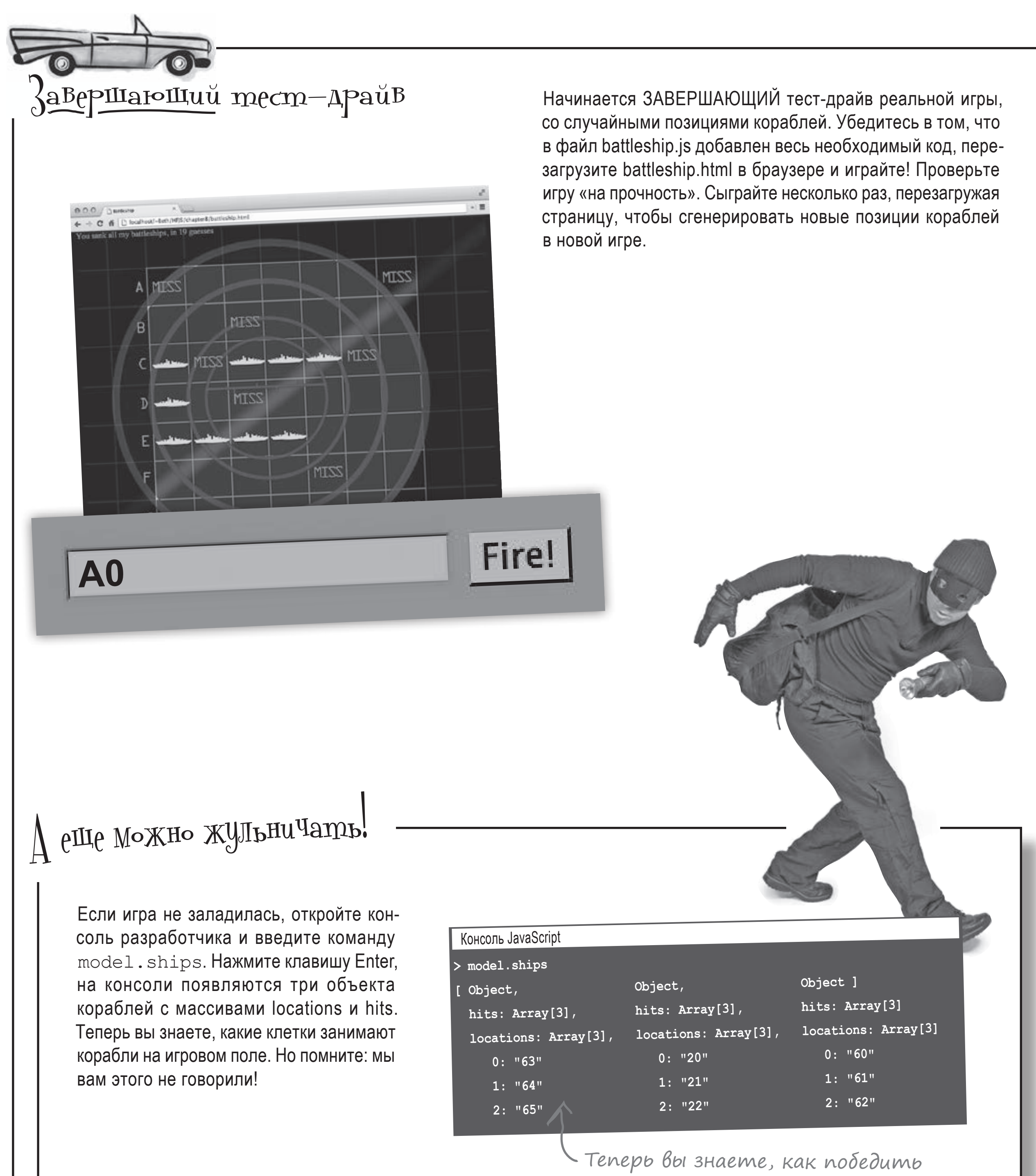
Удалите фиксированные позиции кораблей...

...и замените их массивами, инициализированными нулями.

И конечно, добавьте вызов метода, генерирующего позиции кораблей, который заполнит пустые массивы в объекте модели.

Метод model.generateShipLocations вызывается из функции init, чтобы это происходило во время загрузки игры (до ее начала). При таком вызове позиции всех кораблей будут определены к моменту начала игры.

Напомним, что полный код игры «Морской бой» можно загрузить на сайте http://wickedlysmart.com/hfjs.



Завершающий тест-драйв

Начинается ЗАВЕРШАЮЩИЙ тест-драйв реальной игры, со случайными позициями кораблей. Убедитесь в том, что в файл battleship.js добавлен весь необходимый код, перезагрузите battleship.html в браузере и играйте! Проверьте игру «на прочность». Сыграйте несколько раз, перезагружая страницу, чтобы сгенерировать новые позиции кораблей в новой игре.

А еще можно жульничать!

Если игра не заладилась, откройте консоль разработчика и введите команду `model.ships`. Нажмите клавишу Enter, на консоли появляются три объекта кораблей с массивами locations и hits. Теперь вы знаете, какие клетки занимают корабли на игровом поле. Но помните: мы вам этого не говорили!

Консоль JavaScript

```
> model.ships
[ Object,                 Object,                 Object ]
  hits: Array[3],         hits: Array[3],         hits: Array[3]
  locations: Array[3],    locations: Array[3],    locations: Array[3]
     0: "63"                 0: "20"                 0: "60"
     1: "64"                 1: "21"                 1: "61"
     2: "65"                 2: "22"                 2: "62"
```

Теперь вы знаете, как победить компьютер.

Поздравляем, можно запускать стартап!

Вы только что построили замечательное веб-приложение, всего из 150 (или около того) строк кода, а также HTML и CSS. Как мы уже сказали, код в вашем полном распоряжении. Теперь между вами и предпринимательским капиталом стоит только бизнес-план. Но разве это можно назвать непреодолимым препятствием?

Итак, после всей тяжелой работы можно расслабиться и сыграть пару партий в «Морской бой». Затягивает, не правда ли?

Да, и мы только начали. Освоив дополнительные возможности JavaScript, мы сможем взяться за приложения, не уступающие написанным на традиционном языке программирования.

В этой главе мы рассмотрели достаточно большой объем кода. Перекусите и хорошенько отдохните, чтобы новые знания уложились в голове. Но перед этим ознакомьтесь со списком ключевых моментов этой главы. Не пропускайте их; повторение — мать учения!

Заметки

Кажется, я нашла ошибку. Если после попадания в корабль снова ввести те же координаты, игра снова сообщает о попадании. Попробуйте и убедитесь сами!

Итак, это ошибка? А может, так и должно быть? И если ошибка, то как ее исправить? Запишите здесь, что вы об этом думаете:

КЛЮЧЕВЫЕ МОМЕНТЫ

- HTML используется для создания структуры приложения, CSS — для стилевого оформления и JavaScript — для поведения.
- Идентификатор каждого элемента <td> в таблице используется для обновления игрового поля — вывода маркера попадания или промаха.
- Форма использует элемент ввода с типом "button". К кнопке присоединяется **обработчик события**, чтобы мы могли в программе узнать о том, что игрок ввел координаты выстрела.
- Для получения данных из текстового поля формы используется свойство **value** элемента.
- Механизмы позиционирования CSS могут использоваться для точного размещения элементов на веб-странице.
- Структура кода приложения состоит из трех объектов: **модели**, **представления**, и **контроллера**.
- Каждый объект в игре обладает одной **первичной обязанностью**.
- Модель отвечает за хранение состояния игры и реализацию логики, изменяющей это состояние.
- Представление отвечает за обновление изображения при изменении состояния модели.
- Контроллер отвечает за связывание компонентов, передачу вводимых данных модели для обновления состояния и проверку завершения игры.
- Проектирование с объектами, обладающими **разделенными обязанностями**, позволяет строить и тестировать разные части игры независимо друг от друга.
- Чтобы упростить создание и тестирование модели, мы сначала использовали фиксированные позиции кораблей. Убедившись в том, что модель работает, мы заменили фиксированные позиции случайными, генерируемыми в коде.
- Мы используем свойства модели (такие, как numShips и shipLength), чтобы нам не пришлось фиксировать в методах значения, которые могут измениться позднее.
- Массивы поддерживают метод **indexOf**, сходный с методом indexOf строк. Метод indexOf получает значение и возвращает индекс значения, если оно присутствует в массиве, или –1, если оно отсутствует.
- Механизм **сцепления** позволяет объединять ссылки на объекты в цепочки (оператором «точка»). Таким образом несколько команд объединяются в одну с исключением временных переменных.
- Цикл **do while** аналогичен циклу while за исключением того, что условие проверяется после однократного выполнения команд тела цикла.
- **Контроль качества** является важным аспектом процесса разработки. Контроль качества требует ввода не только правильных, но и ошибочных и некорректных данных.

Учебные стрельбы. Решение

Прежде чем писать код размещения попаданий и промахов на игровом поле, давайте немного потренируемся в работе с CSS. Добавьте в разметку классы “hit” и “miss”, соответствующие приведенным ниже действиям игрока. Мы подготовили два класса CSS для ваших экспериментов. Добавьте эти два правила в CSS, а затем представьте, что на игровом поле расставлены корабли в следующих позициях:

Корабль 1: A6, B6, C6

Корабль 2: C4, D4, E4

Корабль 3: B0, B1, B2

А это выстрелы игрока:

A0, D4, F5, B2, C5, C6

Добавьте один из двух приведенных классов к ячейкам игрового поля (элементам <td> таблицы), чтобы на сетке в правильных местах выводились маркеры промахов и попаданий.

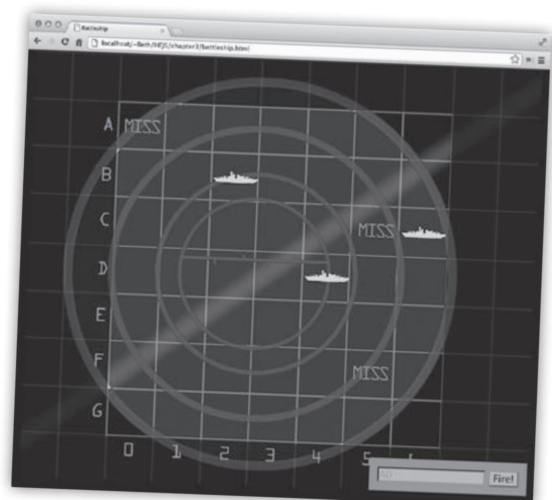
Не забудьте загрузить все необходимое, включая два графических файла, которые потребуются для этого упражнения.

```
.hit {
     background: transparent url("ship.png") no-repeat center center;
}
.miss {
     background: transparent url("miss.png") no-repeat center center;
}
```

А вот наше решение. Класс .hit должен быть назначен элементам <td> с идентификаторами: “00”, “34”, “55”, “12”, “25” и “26”. Для добавления класса к элементу используется атрибут class:

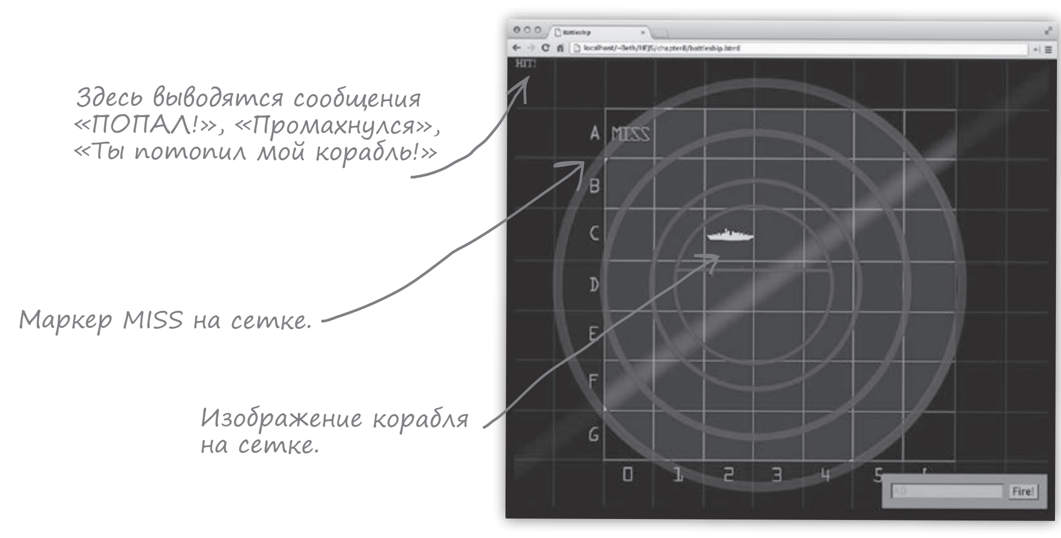
```
<td class="miss" id="55">
```

После добавления классов к элементам игровое поле должно выглядеть так.



Упражнение
Решение

Пора заняться объектным проектированием. Мы начнем с объекта представления. Вспомните, о чем говорилось выше — объект представления отвечает за обновление представления. Взгляните на иллюстрацию и посмотрите, удастся ли вам выбрать методы, которые должны быть реализованы в объекте представления. Напишите объявления этих методов внизу (только объявления; код тела методов будет приведен чуть позднее) вместе с небольшим описанием того, что делает каждый метод. Ниже приведено наше решение:




```
var view = {

    // метод получает строковое сообщение и выводит его
    // в области сообщений
    displayMessage: function(msg) {
        // Код будет приведен позднее
    },

    displayHit: function(location) {
        // Код будет приведен позднее
     },

     displayMiss: function(location) {
        // Код будет приведен позднее
     }
};
```

Мы определяем объект и присваиваем его переменной view.

Запишите здесь свои методы.

Возьми в руку карандаш Решение

Как бы вы представили корабли в модели для приведенного выше игрового поля (пока только позиции самих кораблей, попаданиями займемся позже)? Выберите лучшее решение из представленных ниже.

- ☐ Использовать девять переменных (по аналогии с тем, как мы представляли позиции кораблей в главе 2).
- ☐ Использовать массив с элементами, представляющими клетки игрового поля (49 элементов). В элементе каждой клетки, содержащей часть корабля, хранится номер корабля.
- ☐ Использовать массив для хранения всех девяти позиций. В элементах 0–2 хранятся данные первого корабля, в элементах 3–5 — данные второго, и т. д.
- ☐ Использовать три разных массива, по одному для каждого корабля. Каждый массив содержит данные трех клеток.
- ☑ Использовать объект с именем ship и тремя свойствами (для каждой из трех клеток). Объекты ship хранятся в массиве с именем ships.
- ☐ ______________________________

Или запишите собственный вариант.

Все эти решения работают! (Вообще-то мы опробовали их, когда готовили материал для этой главы.) Этот вариант использовался в книге.

Развлечения с магнитами. Решение

Блоки алгоритма, генерирующего позиций кораблей, перепутались. Сможете ли вы разместить магниты в правильной последовательности, чтобы создать работоспособный алгоритм? Ниже приведено наше решение.

```
Выполнить цикл по количеству создаваемых кораблей.

Сгенерировать случайное направление (вертикальное или горизонтальное) для нового корабля.

Сгенерировать случайную позицию для нового корабля.

Проверить, не перекрываются ли клетки нового корабля с клетками каких-либо существующих кораблей.

Добавить позиции нового корабля в массив ships.
```

Развлечения с магнитами. Решение

Расставьте на игровом поле магниты с маркерами попаданий и промахов для следующей последовательности ходов и структуры данных кораблей. Удалось ли игроку потопить все корабли? Мы сделали первый ход за вас.

Координаты выстрелов:

A6, B3, C4, D1, B0, D4, F0, A1, C6, B1, B2, E4, B6

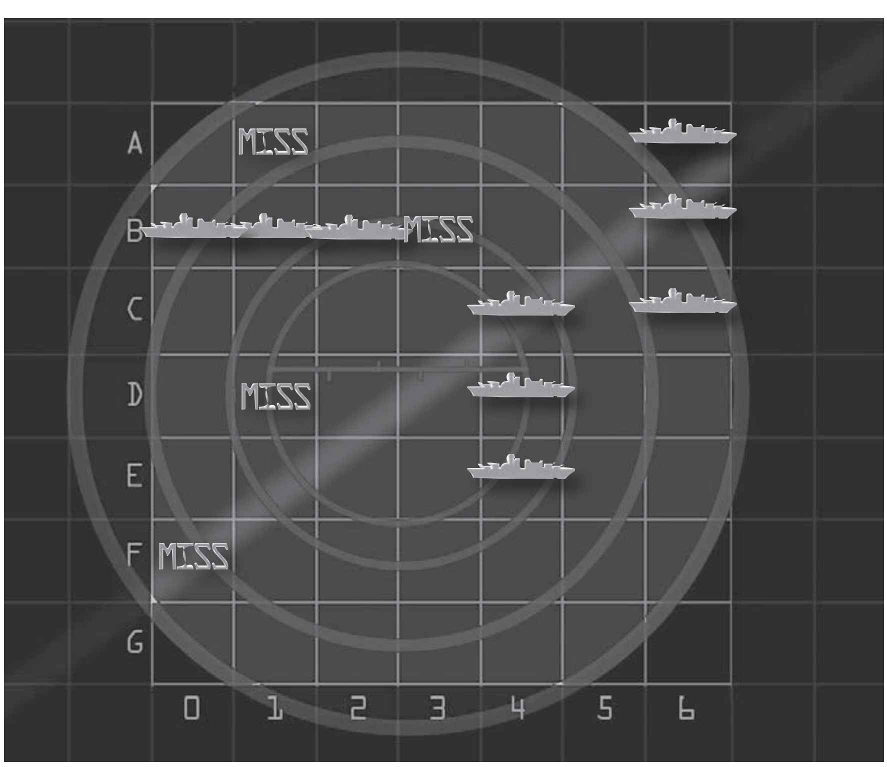
Реализуйте эти ходы на игровом поле.

Наше решение:

```
var ships = [{ locations: ["06", "16", "26"], hits: ["hit", "hit", "hit"] },
             { locations: ["24", "34", "44"], hits: ["hit", "hit", "hit"] },
             { locations: ["10", "11", "12"], hits: ["hit", "hit", "hit"] }];
```

Все три корабля потоплены!

Игровое поле и магниты.



Лишние магниты.

Возьми в руку карандаш
Решение

Давайте потренируемся в использовании структуры данных ships и смоделируем серию операций. По приведенному ниже определению ships найдите ответы на следующие вопросы и заполните пропуски. Обязательно проверьте свои ответы, прежде чем следовать дальше, потому что это очень важная часть игрового механизма:

```
var ships = [{ locations: ["31", "41", "51"], hits: ["", "", ""] },
             { locations: ["14", "24", "34"], hits: ["", "hit", ""] },
             { locations: ["00", "01", "02"], hits: ["hit", "", ""] }];
```

Какие корабли уже были “подстрелены”? Ships 2 и 3 В каких позициях? C4, A0

Игрок стреляет по клетке “D4”, попадет ли он в корабль? Да Если да, то в какой? Ship 2

Игрок стреляет по клетке “B3”, попадет ли он в корабль? Нет Если да, то в какой? ___________

Допишите следующий код, чтобы он определял позицию средней клетки корабля и выводил ее методом console.log:

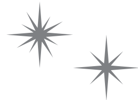
```
var ship2 = ships[ 1 ];
var locations = ship2.locations;
console.log("Location is " + locations[ 1 ]);
```

Допишите следующий код, чтобы он определял, было ли попадание в первой клетке третьего корабля:

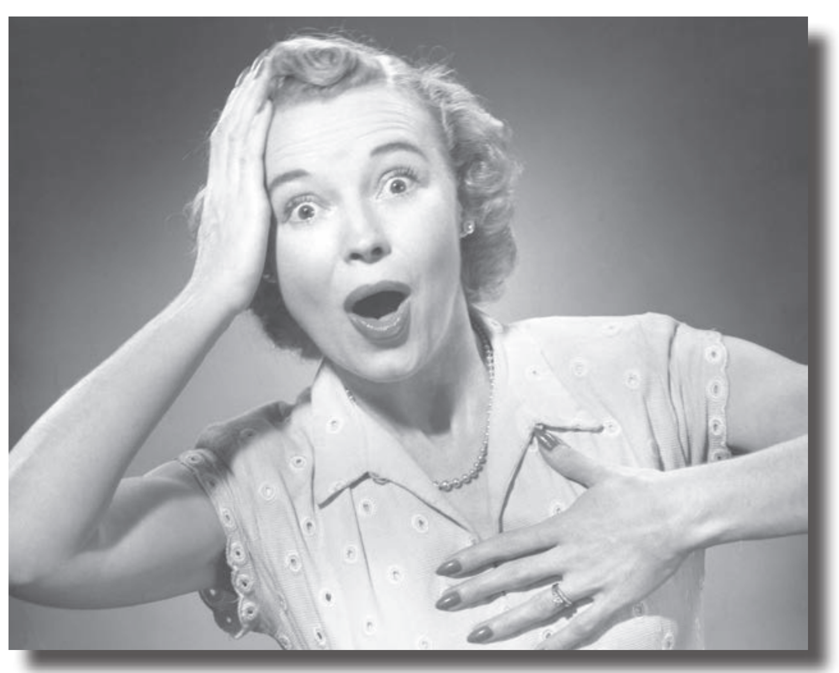
```
var ship3 = ships[ 2 ];
var hits = ship3.hits;
if (hits[0] === "hit") {
   console.log("Ouch, hit on third ship at location one");
}
```

Допишите следующий код, чтобы он записывал попадание в третью клетку первого корабля:

```
var ship1 = ships[0];
var hits = ship1.hits;
hits[ 2 ] = "hit";
```



Обработка событий



В этой главе вам предстоит подняться на принципиально новый уровень. До настоящего момента мы писали код, который обычно выполняется сверху вниз. Конечно, в нем использовались функции, объекты и методы, но выполнение шло по заранее намеченной колее. Жаль, что нам приходится сообщать такие новости во второй половине книги, но **такая структура кода не характерна для JavaScript**. Большая часть кода JavaScript пишется **для обработки событий**. Каких событий? Да любых. Пользователь щелкает на странице, данные поступают из сети, в браузере срабатывает таймер, в DOM происходят изменения... Это далеко не полный список. Более того, в браузере **постоянно** происходят события, которые в основном остаются незамеченными. В этой главе мы пересмотрим свой подход к программированию и узнаем, для чего же нужно писать код, реагирующий на события.

Мозговой штурм

Вы знаете, как работает браузер, правда? Он загружает страницу и все ее содержимое, а затем отображает ее на экране. Но этим роль браузера вовсе не ограничивается. Что еще он делает? Выберите из следующего списка операции, которые, по вашему мнению, браузер может выполнять незаметно для пользователя. Если не уверены — попробуйте угадать.

- ☐ Определяет, когда страница полностью загружена и выведена на экран.
- ☐ Следит за всеми щелчками на элементах страницы — кнопках, ссылках и т. д.
- ☐ Узнает об отправке пользователем данных формы.
- ☐ Узнает о клавишах, нажатых пользователем на клавиатуре.
- ☐ Узнает о получении элементом фокуса ввода.
- ☐ Отслеживает все перемещения мыши.
- ☐ Следит за показаниями часов, управляет таймерами и хронометрированными событиями.
- ☐ Загружает дополнительные данные для страницы.
- ☐ Следит за изменением размеров и прокруткой страницы.
- ☐ Определяет, когда печенье можно вынимать из духовки.

Возьми в руку карандаш

Выберите два события из приведенного списка. Какой интересный или нестандартный код можно было бы написать, если бы браузер мог оповещать ваш код о наступлении этих событий?

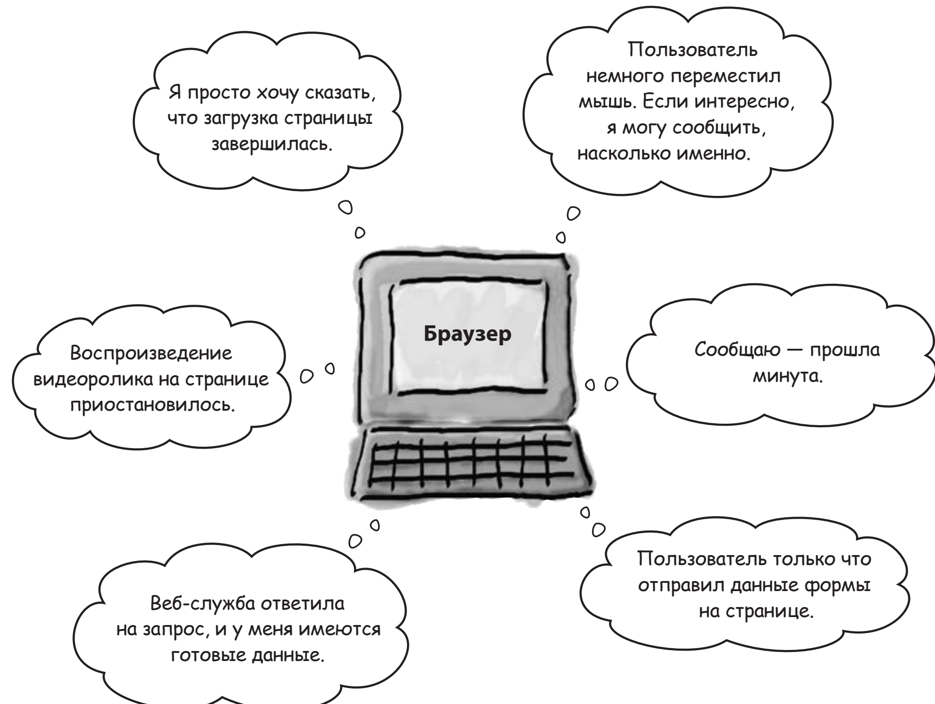
Нет, событие с печеньем вам использовать не удастся!

Что такое «событие»?

Вы уже знаете, что после загрузки и отображения страницы браузер не сидит без дела. За кулисами идет бурная жизнь: пользователи нажимают кнопки, отслеживаются перемещения мыши, поступают данные, меняются размеры окон, срабатывают таймеры и т. д. Все это приводит к срабатыванию событий.

Каждый раз, когда происходит событие, ваш код получает возможность обработать его — иначе говоря, вы можете определить код, который должен вызываться при возникновении заданного события. Обрабатывать какие-либо события в программе не обязательно, но это станет необходимо, если при возникновении событий должно происходить что-то интересное — допустим, при возникновении события нажатия кнопки в список воспроизведения может добавляться новая песня; при поступлении новых данных они могут обрабатываться и отображаться на странице; при срабатывании таймера можно сообщить пользователю, что срок действия его резерва на билет в первом ряду скоро истечет, и т. д.

Средства геопозиционирования, а также другие нетривиальные типы событий рассматриваются в книге «Изучаем программирование на HTML5». А здесь мы ограничимся традиционными типами событий.



Каждый раз, когда происходит <u>событие</u>, вашему коду предоставляется возможность <u>обработать</u> его.

Что такое «обработчик события»?

Обработчики пишутся для обработки событий. Как правило, обработчик представляет собой небольшой фрагмент кода, который должен выполняться при возникновении события. В программном коде обработчик представляет собой обычную функцию. Когда происходит событие, вызывается функция-обработчик.

Разработчики также часто используют термин «функция обратного вызова» и «слушатель».

Чтобы ваш обработчик вызывался при возникновении события, его необходимо сначала *зарегистрировать*. Как вы увидите, существует несколько способов регистрации в зависимости от типов события. До этого мы еще доберемся, а пока начнем с простого примера, который вам уже встречался: события, которое генерируется при полной загрузке страницы.

Браузер, у меня тут есть код, который должен выполняться, когда страница будет полностью загружена. И кстати, я спешу...

Конечно-конечно, как только я завершу загрузку, ваш обработчик будет готов к работе. И не сомневайтесь, я его непременно выполню.

Браузер.

Обработчик — код, который должен быть выполнен позднее, после завершения загрузки страницы.

Как создать первый обработчик событий

Чтобы понять, как работают события, проще всего написать обработчик и подключить его для обработки реального, конкретного события. Ранее мы уже встречали пару примеров обработки событий, включая событие загрузки страницы, но подробно не объясняли, как работает обработка событий. Событие загрузки страницы инициируется тогда, когда браузер полностью загрузил и отобразил все содержимое страницы (и построил модель DOM, представляющую страницу).

Давайте шаг за шагом разберем, как написать обработчик и обеспечить его вызов при возникновении события:

1. Сначала необходимо написать функцию, которая обрабатывает событие загрузки страницы. В нашем случае функция при завершении загрузки просто выводит сообщение «I'm alive!».

Обработчик — это обычная функция.

```
function pageLoadedHandler() {
    alert("I'm alive!");
}
```

А вот и наша функция. Мы назвали ее pageLoadedHandler, но вы можете присвоить своему обработчику любое имя.

Еще раз напомним, что обработчики также часто называются «функциями обратного вызова».

Этот обработчик события ограничивается выводом сообщения.

2. Обработчик написан и готов к работе. Теперь нужно создать необходимые связи, чтобы браузер знал о функции, которая должна вызываться при каждом возникновении события загрузки страницы. В этом нам поможет свойство onload объекта window:

```
window.onload = pageLoadedHandler;
```

В случае события load имя обработчика назначается свойству onload объекта window.

Теперь при возникновении события загрузки страницы будет вызываться функция pageLoadedHandler.

Вскоре вы увидите, что для разных событий используются разные способы назначения обработчиков.

3. Вот и все! Теперь можно устроиться поудобнее и понаблюдать за тем, как браузер будет вызывать функцию, связанную со свойством window.onload, при загрузке страницы.

Тест-драйв

Создайте новый файл event.html и добавьте код тестирования нового обработчика события загрузки. Загрузите страницу в браузере и убедитесь в том, что сообщение появляется на экране.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> I'm alive! </title>
  <script>
    window.onload = pageLoadedHandler;
    function pageLoadedHandler() {
        alert("I'm alive!");
    }
  </script>
</head>
<body>
</body>
</html>
```

Сначала браузер загружает страницу, начинает разбор HTML и построение DOM.

Добравшись до сценария, браузер начинает выполнение кода.

В данный момент браузер просто определяет функцию, которая назначается свойству window. onload. Напомним, что функция будет вызываться при полной загрузке страницы.

Тогда браузер продолжает разбор HTML.

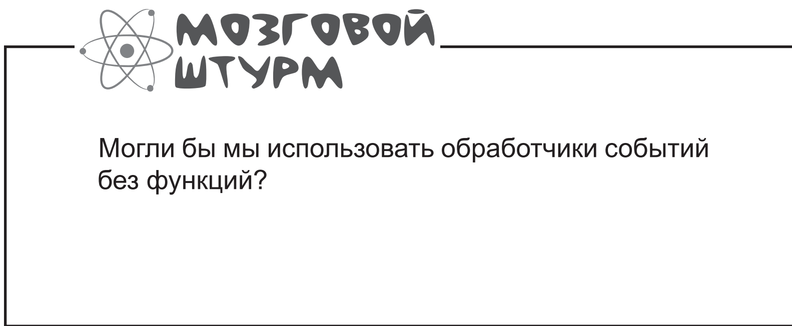
Когда браузер завершает разбор HTML и модель DOM готова к использованию, браузер вызывает обработчик события загрузки страницы.

Что в данном случае приводит к созданию сообщения «I'm alive».

The page at localhost says:

I'm alive!

OK



Могли бы мы использовать обработчики событий без функций?

Как упоминалось ранее, прежде мы применяли относительно линейный подход к написанию кода: брали алгоритм (скажем, вычисление самого эффективного образца раствора, генерирование песни про 99 бутылок и т. д.) и писали код шаг за шагом, от начала к концу.

А теперь вспомните игру «Морской бой». Код этой игры не укладывался в линейную модель. Конечно, мы писали код игры, инициализировали модель и все такое, но далее основная часть игры работала *по-другому*. Каждый раз, когда игрок хотел сделать очередной выстрел, он вводил координаты в элементе формы и нажимал кнопку «Fire». Кнопка запускала последовательность действий, которая приводила к отработке следующего хода игры. Иначе говоря, ваш код *реагировал* на ввод пользователя.

Организация кода на базе реакции на события — совершенно другой подход к программированию. Чтобы написать код подобным образом, необходимо предвидеть события, которые могут произойти, и спланировать реакцию на них в коде. Профессионалы называют такой код *асинхронным*, потому что написанный код должен быть вызван *позднее*, *если произойдет событие – и когда* оно произойдет. Подобная методология программирования также означает переход от пошаговой реализации алгоритма к построению приложения из множества обработчиков, реагирующих на разные события.

Как разобраться в событиях? Написать игру, конечно!

Как обычно, понимание событий приходит с опытом. А для того чтобы этот опыт получить, мы напишем простую игру. Игра работает так: пользователь загружает страницу с изображением — сильно размытым. Игрок должен угадать, что изображено на странице. И чтобы проверить свой ответ, он щелкает на изображении, чтобы восстановить его в исходном виде. Вот так:

Начнем с разметки. В ней будут использоваться два изображения в формате JPG: одно размыто, другое нет. Мы назвали их zeroblur.jpg и zero.jpg соответственно. Разметка выглядит так:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style> body { margin: 20px; } </style>
  <script>  </script>
</head>
<body>
    <img id="zero" src="zeroblur.jpg">
</body>
</html>
```

Базовая разметка HTML с элементом <script> для нашего кода. Вместо того чтобы хранить JavaScript в отдельном файле, мы упростим задачу и добавим сценарий в разметку. Как вы увидите, реализация этой схемы требует совсем небольшого объема кода.

Размытое изображение размещается на странице. Мы присвоили ему идентификатор “zero”. Сейчас вы увидите, как этот идентификатор будет использоваться...

Реализация игры

Загрузите эту разметку в браузере; вы увидите размытое изображение. Чтобы реализовать игру, следует организовать обработку щелчков на изображении (по щелчку должна появляться исходная, неразмытая версия изображения).

К счастью, при каждом щелчке на элементе HTML на странице (или касании на мобильных устройствах) генерируется соответствующее событие. Ваша задача — написать обработчик для этого события и включить в него код отображения исходной версиии изображения. Решение этой задачи будет состоять из двух шагов:

1. **Обратиться к объекту изображения в DOM и назначить обработчик его свойству onclick.**

2. **Написать и включить в обработчик код изменения атрибута src изображения, чтобы размытое изображение заменялось исходным.**

Давайте проанализируем эти два пункта и напишем код.

Шаг 1: обращение к изображению в DOM

Вы уже прекрасно знаете, как это делается; нужно получить ссылку на элемент при помощи нашего старого знакомого, метода `getElementById`.

```
var image = document.getElementById("zero");
```

Получаем ссылку на элемент изображения и присваиваем ее переменной image.

Этот код должен выполняться только *после* создания модели DOM страницы; для соблюдения этого требования используется свойство `onload` объекта window. Мы размещаем свой код в функции `init`, которая назначается свойству `onload`.

Не забудьте, что программа не может получить элемент изображения из DOM, пока загрузка страницы не завершилась.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
}
```

Мы создаем функцию init и назначаем ее обработчику onload, чтобы она гарантированно выполнялась только после полной загрузки страницы.

Функция init получает ссылку на изображение с идентификатором “zero”.

Помните: в JavaScript порядок определения функций не играет роли. Функция init может определяться и после ее назначения свойству onload.

Шаг 2: добавление обработчика и обновление изображения

Чтобы добавить обработчик для событий щелчка на изображении, мы просто назначаем функцию свойству `onclick` элемента изображения. Сейчас мы назовем эту функцию `showAnswer`, а на следующем шаге займемся ее определением.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;
}
```

Обработчик назначается свойству onclick объекта изображения, полученного из DOM.

Затем необходимо написать функцию `showAnswer`, которая восстанавливает исходное (неразмытое) изображение, назначая его свойству src элемента image:

Сначала мы снова получаем элемент image из модели DOM.

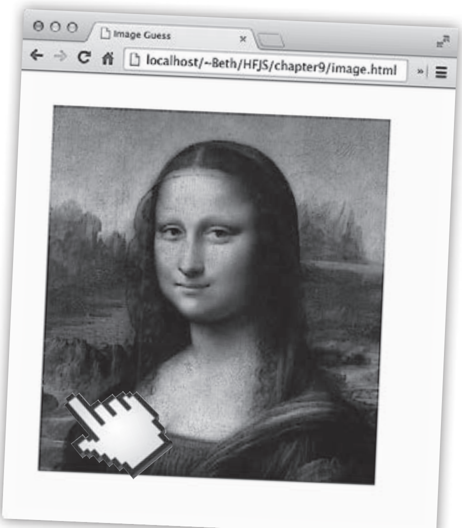
```
function showAnswer() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Размытая версия изображения хранится в файле zeroblur.jpg, а нормальная — в файле zero.jpg.

Получив элемент, мы можем изменить его, назначая свойству src неразмытую версию изображения.

Тест-драйв

Давайте опробуем нашу простую игру. Убедитесь в том, что весь код HTML, CSS и JavaScript сохранен в файле image.html, а загруженные с сайта http://wickedlysmart.com/hfjs изображения хранятся в той же папке. Когда это будет сделано, откройте файл в браузере и проверьте, как он работает!



Щелкните в любой точке изображения, чтобы программа вызвала обработчик showAnswer. Когда это происходит, свойство src элемента изображения изменяется и на экране появляется нормальное изображение.

Напомните, для чего мы использовали getElementById в функции showAnswer? Я не до конца понимаю, что когда происходит.

Да, уследить за последовательностью выполнения в программе с многими обработчиками событий бывает непросто. Напоминаем: функция `init` вызывается function после завершения загрузки страницы. Но функция `showAnswer` будет вызвана позднее, когда пользователь щелкнет на изображении. Итак, эти два обработчика событий вызываются в разное время.

Также следует помнить правила видимости имен. В функции `init` объект, полученный при вызове `getElementById`, присваивается локальной переменной `image`; это означает, что при завершении функции переменная выходит из области действия и уничтожается. Позднее, при вызове функции `showAnswer`, нам приходится заново получать объект изображения из DOM. Конечно, объект можно было сохранить в глобальной переменной, но слишком частое использование глобальных переменных ведет к появлению запутанного кода и повышает вероятность ошибок. Нам бы хотелось этого избежать.

Часто задаваемые вопросы

В: Назначение свойства src изображения — то же, что назначение атрибута src вызовом setAttribute?

О: В данном случае — да. При получении элемента HTML из DOM вызовом getElementById вы получаете объект element с набором методов и свойств. У всех объектов element имеется свойство id, содержащее идентификатор элемента HTML (если он был назначен в HTML). Объект element изображения также содержит свойство src, которому назначается графический файл, указанный в атрибуте src элемента <img>.

Однако не все атрибуты имеют соответствующие свойства объекта, и для их получения приходится использовать методы setAttribute и getAttribute. В случае src и id можно использовать свойства и/или методы getAttribute и setAttribute; результат от этого не изменится.

В: Выходит, обработчик вызывается внутри другого обработчика?

О: Не совсем. Обработчик onload содержит код, вызываемый при завершении загрузки страницы. При его вызове мы назначаем обработчик свойству onclick элемента изображения, но вызван он будет только тогда, когда пользователь щелкнет на изображении — теоретически это может произойти значительно позднее завершения загрузки страницы. Получается, что два обработчика вызываются в разное время.

СТАНЬ браузером

Перед вами код игры. Представьте себя на месте браузера и попробуйте определить, что должно происходить после каждого события. Когда упражнение будет выполнено, загляните в конец главы и проверьте свои ответы. Первую строку мы заполнили за вас.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;
}

function showAnswer() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Код, который нужно выполнить...

Во время загрузки страницы...	Сначала определяются функции init и showAnswer
Когда происходит событие завершения загрузки страницы...	
Когда происходит событие щелчка на изображении...	

Запишите свои ответы.

А если бы у вас была целая страница с размытыми изображениями, которые можно было бы по отдельности восстанавливать щелчком? Как бы вы спланировали код для решения этой задачи? Какой способ можно было бы назвать примитивным? Можно ли реализовать новую постановку задачи с минимальными изменениями в уже написанном коде?

Джуди: Привет, парни. Пока все работает. Но игру было бы неплохо доработать, чтобы на странице выводилось несколько изображений вместо одного.

Джим: Конечно, я как раз над этим думал.

Джо: А у меня уже есть готовая графика, нужно только написать код. Я назначал файлам имена по схеме zero.jpg, zeroblur.jpg, one.jpg, oneblur.jpg и т. д.

Джим: Что же, нам придется писать новый обработчик события щелчка для каждого изображения? Появится много повторяющегося кода. В конце концов, все обработчики будут делать одно и то же: заменять размытое изображение исходной версией, верно?

Джо: Верно. Но я не представляю, как использовать один обработчик для разных изображений. Это вообще возможно?

Джуди: Мы можем назначить один обработчик — то есть одну и ту же функцию — свойству `onclick` каждого изображения в игре.

Джо: Получается, что одна и та же функция будет вызываться для каждого изображения, на котором щелкнет пользователь?

Джуди: Точно. Функция `showAnswer` будет использоваться как обработчик события щелчка для каждого изображения.

Джим: Хмм, но как узнать, какое изображение нужно заменить?

Джо: О чем ты говоришь? А разве обработчик этого не знает?

Джим: Откуда ему знать? Сейчас функция `showAnswer` предполагает, что щелчок сделан на изображении с идентификатором “zero”. Если мы будем вызывать `showAnswer` для каждого щелчка на изображении, код должен работать с любым изображением.

Джо: Да, верно... И как узнать, на каком изображении был сделан щелчок?

Джуди: Я тут кое-что читала о событиях. Думаю, обработчику можно передать информацию об элементе, на котором сделан щелчок. Этим мы займемся потом, а пока добавим изображения на страницу и посмотрим, как назначить им всем один обработчик.

Джо, Джим: Договорились!

Добавим несколько изображений

Начнем с добавления всех новых изображений на страницу. Мы добавим пять изображений, таким образом всего их будет шесть. Мы также изменим правила CSS, чтобы изображения разделялись небольшими полями:

Скачай!

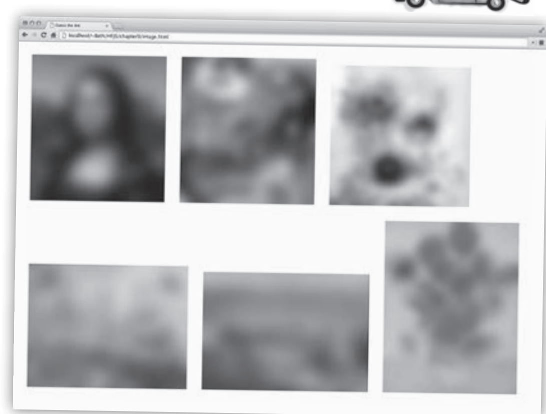
Все графические файлы находятся в папке chapter9 архива, загруженного по адресу http://wickedlysmart.com/hfjs.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style>
    body { margin: 20px; }
    img { margin: 20px; }
  </style>
  <script>
     window.onload = init;
     function init() {
        var image = document.getElementById("zero");
        image.onclick = showAnswer;
    }
    function showAnswer() {
        var image = document.getElementById("zero");
        image.src = "zero.jpg";
    }
  </script>
</head>
<body>
    <img id="zero" src="zeroblur.jpg">
    <img id="one" src="oneblur.jpg">
    <img id="two" src="twoblur.jpg">
    <img id="three" src="threeblur.jpg">
    <img id="four" src="fourblur.jpg">
    <img id="five" src="fiveblur.jpg">
</body>
</html>
```

Это свойство CSS добавляет поля шириной 20px между изображениями.

А это пять новых изображений, которые мы добавляем. Обратите внимание на использование единой схемы формирования значений для id и src (а также имен графических файлов). Вскоре вы увидите, как работает эта схема...

Если вы запустите игру, ваша страница будет выглядеть так:



Теперь нужно назначить один обработчик всем свойствам onclick всех изображений

На странице появилось больше изображений, но и работы тоже прибавилось. В текущей версии вы можете щелкнуть на первом изображении (Мона Лиза) и увидеть исходное изображение, а как с остальными?

Конечно, *можно* написать отдельный обработчик для каждого изображения, но, как говорилось ранее, такое решение получится нудным и неэффективным. Давайте посмотрим:

```
window.onload = init;
function init() {
    var image0 = document.getElementById("zero");
    image0.onclick = showImageZero;
    var image1 = document.getElementById("one");
    image1.onclick = showImageOne;
    ...
}
function showImageZero() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
function showImageOne() {
    var image = document.getElementById("one");
    image.src = "one.jpg";
}
...
```

Поочередно получаем каждый элемент изображения от страницы и назначаем каждому элементу отдельный обработчик. И так шесть раз... Здесь показано только два.

Здесь назначаются еще четыре обработчика.

И нам потребуются шесть разных обработчиков, по одному для каждого изображения.

...Еще четыре функции-обработчика.

Мозговой штурм

Какими недостатками обладает решение с отдельным обработчиком для каждого изображения? Пометьте все подходящие ответы:

- [ ] Присутствие избыточного кода в каждом обработчике.
- [ ] Если потребуется изменить код одного обработчика, изменения придется вносить во всех обработчиках.
- [ ] Решение получается слишком объемистым.
- [ ] Труднее уследить за всеми изображениями и обработчиками.
- [ ] Плохо обобщается для произвольного количества изображений.
- [ ] Другим разработчикам становится труднее работать над кодом.

Как использовать один обработчик для всех изображений

Очевидно, написание отдельного обработчика для каждого изображения — не лучшее решение, поэтому мы воспользуемся существующим обработчиком `showAnswer` для обработки событий щелчка на всех изображениях. Конечно, для этого в `showAnswer` придется внести небольшие изменения. Чтобы функция `showAnswer` обрабатывала щелчки на всех изображениях, необходимо сделать две вещи:

1. **Назначить функцию showAnswer обработчиком события щелчка для каждого изображения на странице.**

2. **Переработать функцию showAnswer, чтобы она выводила все исходные изображения, а не только zero.jpg.**

И все это хотелось бы сделать обобщенным способом, который будет работать даже при добавлении новых изображений на страницу. Другими словами, если код будет правильно написан, добавление изображений на страницу (или их удаление со страницы) не должно приводить к изменениям в коде. Давайте разберемся, как это сделать.

Назначение обработчика щелчка всем изображениям на странице

Первая проблема: в текущей версии кода метод `getElementById` используется для получения ссылки на изображение “zero” и назначения функции `showAnswer` его свойству `onclick`. Вместо того чтобы включать фиксированный вызов `getElementById` для каждого изображения, мы покажем более простой способ: получим все изображения одновременно, переберем их и назначим каждому обработчик щелчка. Для этого будет использован метод DOM, который вы еще не видели: `document.getElementsByTagName`. Этот метод получает имя тега (допустим, `img`, `p` или `div`) и возвращает список подходящих элементов. Применим его на практике:

```
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;

    var images = document.getElementsByTagName("img");
    for (var i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};
```

Избавляемся от старого кода получения элемента с идентификатором “zero” и назначения ему обработчика. (две зачёркнутые строки)

Здесь происходит выборка элементов по имени тега img. Вызов находит все теги изображения на странице и возвращает весь набор. Результат сохраняется в images.

Затем мы перебираем все изображения и поочередно назначаем обработчик события щелчка showAnswer каждому изображению. После завершения цикла свойству onclick каждого изображения назначен обработчик showAnswer.

document.getElementsByTagName под увеличительным стеклом

Метод `document.getElementsByTagName` во многом похож на `document.getElementById`, но вместо работы по идентификатору он получает элементы по имени тега, в нашем случае "img". Конечно, разметка HTML может включать много элементов `<img>`, поэтому метод может вернуть набор элементов, один элемент и даже нуль в зависимости от количества изображений на странице. В нашем примере страница игры содержит шесть элементов `<img>`, поэтому метод вернет список из шести объектов.

Метод возвращает список объектов элементов, соответствующих заданному имени тега.

```
var images = document.getElementsByTagName("img");
```

Возвращается список объектов, напоминающий массив. Формально он не является массивом, но похож на массив по своим характеристикам.

Обратите внимание на букву "s" в имени — метод возвращает не один объект, а набор объектов.

Имя тега заключается в кавычки (и не содержит символы < и >!).

Часто задаваемые вопросы

В: Вы сказали, что метод getElementsByTagName возвращает список. Имеется в виду массив?

О: Метод возвращает объект, с которым можно работать как с массивом, но на самом деле это объект типа NodeList. Объект NodeList представляет коллекцию узлов (технический термин для объектов элементов в дереве DOM). Чтобы выполнить перебор коллекции, следует получить ее длину из свойства length, а затем последовательно обратиться к каждому элементу NodeList по индексу в квадратных скобках (так же, как это делается с массивами). Впрочем, на этом сходство между NodeList и массивом завершается, поэтому в остальном при работе с объектом NodeList необходимо действовать осторожно. Обычно это все, что необходимо знать о NodeList, если только вы не собираетесь добавлять и удалять элементы из DOM.

В: Значит, обработчик события щелчка можно назначить любому элементу?

О: Почти. Получите любой элемент страницы и назначьте функцию его свойству onclick. Готово. Как вы уже видели, обработчик может обслуживать только этот конкретный элемент или же целую группу элементов. Конечно, элементы, не имеющие визуального представления на странице (например, эелменты <script> и <head>), не поддерживают события щелчка.

В: Функции-обработчики получают аргументы?

О: Хороший вопрос... и очень своевременный. Да, получают, и мы как раз собираемся рассмотреть объект события, передаваемый некоторым обработчикам.

В: Элементы поддерживают другие типы событий, кроме щелчков?

О: Да, есть и другие события. Одно из них уже встречалось нам в коде игры «Морской бой» — это событие нажатия клавиши. Тогда функция-обработчик вызывалась при нажатии клавиши Enter. Мы также рассмотрим другие виды событий в этой главе.

Итак, у нас есть общий обработчик showAnswer, который должен обрабатывать щелчки на всех изображениях. Так ты знаешь, как определить, на каком изображении был сделан щелчок при вызове showAnswer?

Джуди: Да, знаю. При каждом вызове обработчика события щелчка передается *объект события*. Мы можем воспользоваться этим объектом для получения подробной информации о событии.

Джо: Например, на каком изображении был сделан щелчок?

Джуди: На более общем уровне — с каким элементом произошло это событие. Этот элемент называется *источником события*.

Джо: Что за источник?

Джуди: Как я уже сказала, это элемент, сгенерировавший событие. При щелчке на конкретном изображении оно становится источником события.

Джо: Значит, если я щелкну на изображении с идентификатором "zero", то источником события будет это изображение?

Джуди: А точнее, объект element, представляющий это изображение.

Джо: Что-что?

Джуди: Объект element — то, что ты получаешь при вызове метода `document.getElementById` с аргументом "zero". Это объект, представляющий изображение в модели DOM.

Джо: Допустим, и как узнать источник события? Похоже, это необходимо для определения того, на каком изображении щелкнул пользователь.

Джуди: Источник определяется по свойству объекта события.

Джо: То, что нужно для `showAnswer`. Сейчас сделаем... Погоди, `showAnswer` получает объект события?

Джуди: Вот именно.

Джо: Тогда как наша функция `showAnswer` работала до этого? Она получает объект события, но в самой функции для этого объекта не определен параметр!

Джуди: Не забудь, что JavaScript позволяет игнорировать параметры.

Джо: Да, точно.

Джуди: И еще нам нужно понять, как заменить свойство `src` правильным именем файла с нормальной версией изображения. В старой версии было известно, что она хранится в файле с именем zero.jpg, но теперь такой способ не проходит.

Джо: Возможно, мы сможем определить имя нормального изображения по id изображения. Идентификатор изображения совпадает с именем файла, содержащего неразмытую версию этого изображения.

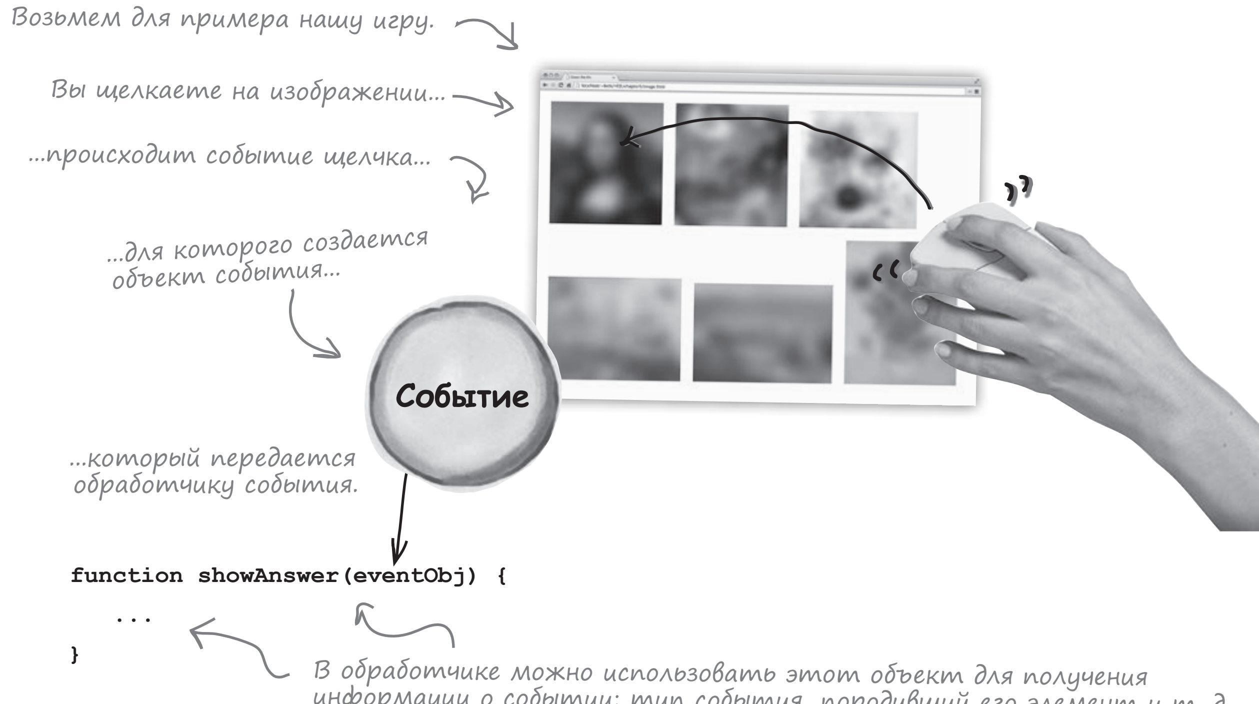
Джуди: Кажется, у нас есть план!

Как работает объект события

При вызове обработчик события щелчка получает *объект события* — более того, этот объект передается для большинства событий, связанных с моделью DOM. Объект события содержит общую информацию о событии: какой элемент породил это событие, в какое время оно произошло и т. д. Кроме того, передается информация, связанная с конкретным событием: например, для щелчка мышью вы получите координаты точки щелчка.

Существуют и другие виды событий (помимо событий DOM). Пример такого события встретится нам в этой главе...

Давайте последовательно разберемся, как работают объекты событий:



Итак, что же собой представляет объект события? Как было сказано ранее, он содержит как общую, так и специализированную информацию о событии. Специализированная информация зависит от типа события, мы еще вернемся к этой теме. Общая информаия включает свойство `target`, в котором хранится ссылка на объект, сгенерировавший событие. Итак, если пользователь щелкнул на элементе страницы (например, на изображении), этот элемент является источником события и к нему можно обратиться следующим образом:

```
function showAnswer(eventObj) {
   var image = eventObj.target;
}
```

Свойство target сообщает, какой элемент сгенерировал данное событие.

Вы уже знаете, что объект события (для событий DOM) содержит свойства, которые предоставляют расширенную информацию о произошедшем событии. Ниже перечислены другие свойства, которые могут присутствовать в объекте события. Соедините каждое свойство объекта события с его кратким описанием.

target	Хотите узнать, на каком расстоянии от верхней стороны окна браузера был сделан щелчок? Используйте меня.
type	Я содержу объект, породивший событие. Объекты могут быть разными, но чаще всего это объект, представляющий элемент страницы.
timeStamp	Используете устройство с сенсорным экраном? Я подскажу вам, сколькими пальцами пользователь прикасается к экрану.
keyCode	Я – строка вида “click” или “load”, которая описывает, что только что произошло.
clientX	Интересует, когда произошло событие? Я сообщу вам об этом.
clientY	Хотите узнать, на каком расстоянии от левой стороны окна браузера был сделан щелчок? Используйте меня.
touches	С моей помощью вы узнаете, какую клавишу нажал пользователь.

Работаем с объектом события

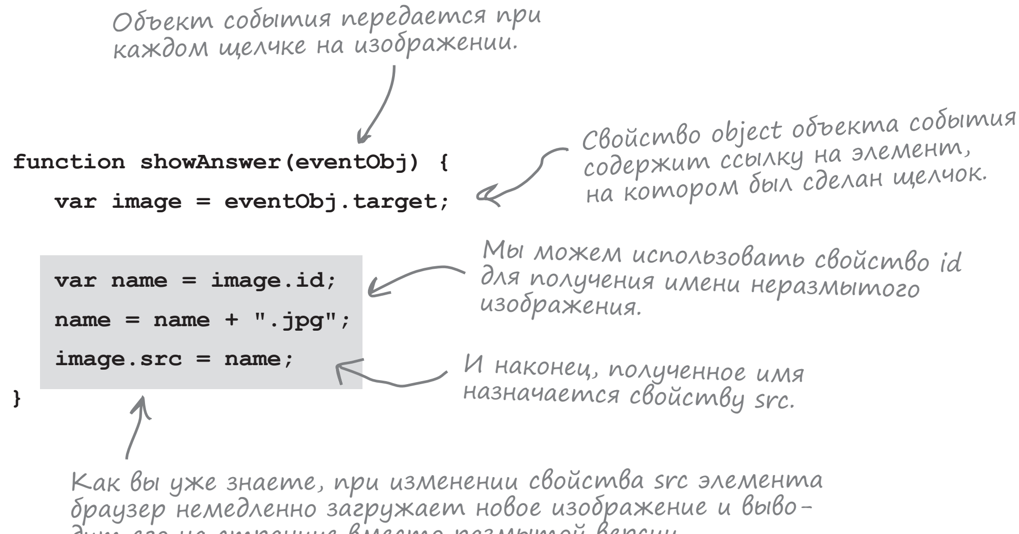
Итак, теперь вы немного больше знаете о событиях, а если говорить конкретнее, то об объекте события, передаваемом обработчику события щелчка. Давайте посмотрим, как использовать информацию, содержащуюся в объекте события, для замены произвольного размытого изображения на странице его нормальной версией. Начнем с возвращения к разметке HTML.

```
<!doctype html>
 ...
<body>
    <img id="zero" src="zeroblur.jpg">
    <img id="one" src="oneblur.jpg">
    <img id="two" src="twoblur.jpg">
    <img id="three" src="threeblur.jpg">
    <img id="four" src="fourblur.jpg">
    <img id="five" src="fiveblur.jpg">
</body>
</html>
```

Знакомая разметка HTML.

Каждому элементу <img> назначается идентификатор, соответствующий имени неразмытого изображения. Так, нормальная версия изображения с идентификатором "zero" хранится в файле zero.jpg. Изображение с идентификатором "one" хранится в файле one.jpg и т. д.

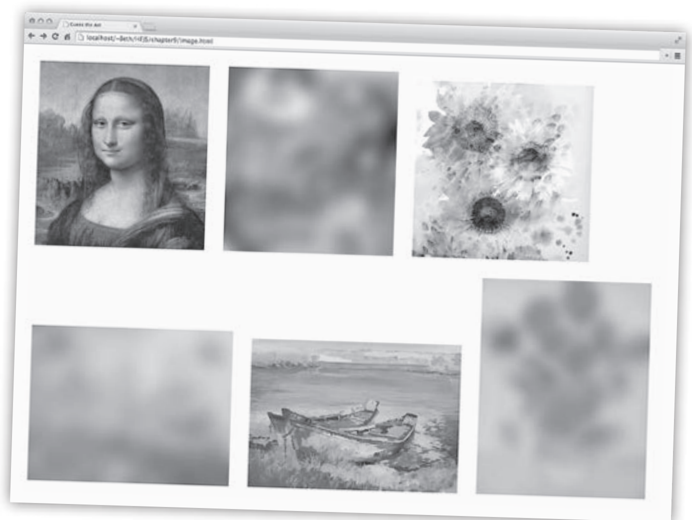
Обратите внимание: идентификатор каждого изображения соответствует имени нормальной версии изображения (если отбросить расширение ".jpg"). Теперь, зная идентификатор, мы можем просто прибавить к нему ".jpg" для создания имени соответствующего неразмытого изображения. Теперь остается лишь изменить свойство `src` элемента. Вот как это делается:



Тест-драйв: объект события и источник

Обновите код в файле image.html и загрузите страницу для тестирования. Попробуйте понять, какая картина скрыта за размытым изображением, и щелкните на нем — открывается исходная версия изображения. Вспомните, что это приложение проектировалось не как программа с последовательным выполнением, а как набор действий, которые генерируются событиями в результате щелчков на изображениях. Обратите внимание на то, что события всех изображений обрабатываются одним фрагментом кода, достаточно умным для определения выбранного изображения. Поэкспериментируйте с приложением. Что произойдет, если щелкнуть на изображении дважды? А вообще что-нибудь произойдет?

Теперь щелчок на любом изображении открывает его исходную версию. Сколько картин узнали вы?



Часто задаваемые вопросы

В: Обработчику события onload тоже передается объект события?

О: Да, передается. В нем содержится информация об источнике (объект window), времени возникновения события и типе события (“load”). Можно с уверенностью сказать, что объект события в обработчиках onload используется очень редко, потому что для этого события хранящаяся в нем информация не столь актуальна. Иногда объект события полезен, иногда нет — все зависит от типа события. Если вы не уверены в том, какую информацию содержит объект конкретного типа события, обращайтесь к справочнику по JavaScript.

Допустим, вы хотите, чтобы изображение снова становилось размытым через несколько секунд после открытия исходной версии. Как это можно сделать?

Откровенно о событиях

Интервью недели:
Беседа с браузером о событиях

Head First: Привет, Браузер. Спасибо, что заглянули. Мы знаем, как у вас много работы.

Браузер: Пожалуйста. И вы правы, едва успеваю управляться со всеми этими событиями.

Head First: И как вы управляетесь с ними? Расскажите, как это волшебство выглядит, так сказать, из-за кулис.

Браузер: Как вам известно, события происходят постоянно. Пользователь перемещает мышь, делает жесты на мобильном устройстве, поступают новые данные, срабатывают таймеры... Суета, как в час пик. В общем, хлопот выше крыши.

Head First: Но ведь вам ничего не нужно делать, если для события не определен обработчик?

Браузер: Даже если обработчика нет, работы все равно хватает. Кто-то должен обнаружить событие, интерпретировать его и проверить, назначен ли для него обработчик. Если обработчик есть, нужно передать ему управление.

Head First: И как же вы следите за всеми событиями? Что если произойдет несколько событий одновременно? Вы же не можете делать несколько дел сразу?

Браузер: Да, за короткий промежуток времени может случиться множество событий. Иногда события происходят слишком быстро, чтобы работать в реальном времени. Поэтому я ставлю их в очередь по мере поступления, а затем прохожу по очереди и запускаю обработчики там, где это необходимо.

Head First: Прямо как в юности, когда я подрабатывал в закусочной!

Браузер: Наверное... Если заказы поступали каждую миллисекунду или около того!

Head First: И вам приходится просматривать содержимое очереди, событие за событием?

Браузер: Конечно, и это очень важная особенность JavaScript: существует всего одна очередь и один «программный поток выполнения». Значит, только один «экземпляр меня» перебирает события.

Head First: Что это означает для наших читателей?

Браузер: Допустим, вы пишете обработчик, требующий большого объема вычислений, — то есть обработчик, на выполнение которого потребуется много времени. Пока ваш обработчик усердно трудится, я сижу и ожидаю, когда он завершится. Только тогда я смогу продолжить обработку очереди.

Head First: Надо же! И часто вам приходится ждать завершения медленного кода?

Браузер: Бывает... Но веб-программист быстро понимает, что страница или приложение не реагирует из-за медленного обработчика. В общем, если веб-программист знает, как работают очереди сообщений, эта проблема встречается нечасто.

Head First: Теперь наши читатели тоже это знают! Но вернемся к событиям. Их много видов?

Браузер: Много. Есть события, относящиеся к передаче данных, события DOM, связанные со страницей, и много других. События DOM, например, генерируют объекты событий, содержащие более подробную информацию о событии. В событии щелчка мышью есть информация о том, в какой точке он был сделан, а в событии нажатия клавиши — о нажатой клавише и т. д.

Head First: Итак, вы тратите значительное время на обработку событий. Стоит ли оно того? В конце концов, вам еще приходится заниматься загрузкой, разбором, отображением страниц и т. п.

Браузер: О, это очень важно. В наши дни пишется много кода, который делает страницы интерактивными и увлекательными, а для этого нужны события.

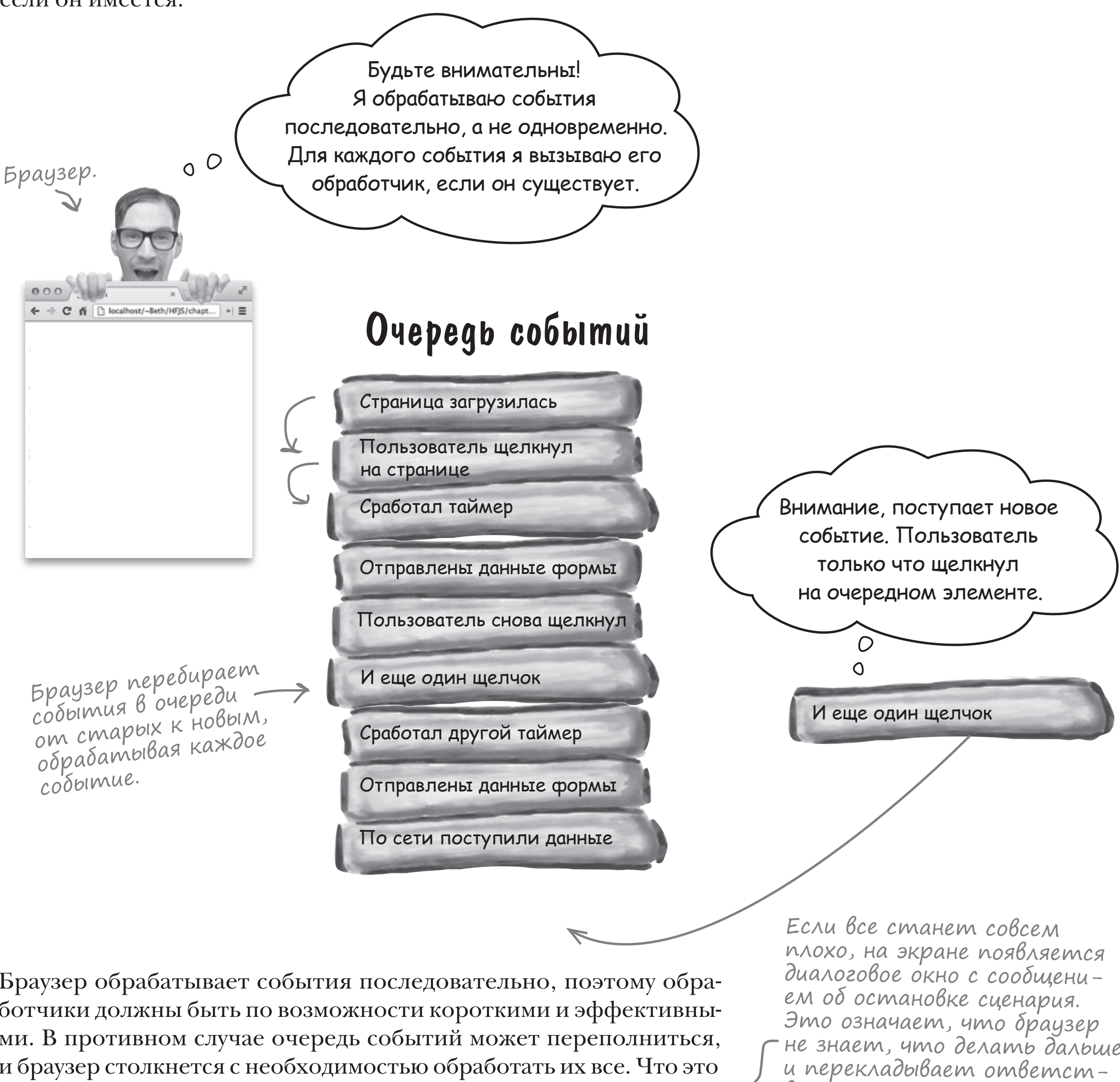
Head First: Конечно, времена простых статических страниц прошли.

Браузер: Вот именно. Ой, у меня сейчас очередь переполнится... Надо бежать!

Head First: Хорошо... До следующего раза!

Очереди и события

Вы уже знаете, что браузер поддерживает очередь событий. И он незаметно для пользователя постоянно извлекает события из очереди и обрабатывает их, передавая подходящему обработчику, если он имеется.



Браузер обрабатывает события последовательно, поэтому обработчики должны быть по возможности короткими и эффективными. В противном случае очередь событий может переполниться, и браузер столкнется с необходимостью обработать их все. Что это означает для вас? Что интерфейс начнет «тормозить» и с большими задержками реагировать на действия пользователя.

Эй, салага! У нас тут есть карта сокровищ, но ты должен помочь нам определить координаты клада. Для этого нужно написать фрагмент кода, который выводит координаты при наведении указателя мыши на точку карты. Начало кода приведено на следующей странице, но тебе придется закончить его.

Карта. Клад помечен крестиком!

Когда код будет написан, наведите указатель мыши на крестик — и вы узнаете координаты клада.

Ваш код должен выводить координаты под картой.

P. S. Мы настоятельно рекомендуем выполнить это упражнение — пираты будут очень недовольны, если не получат координаты... Кстати, для завершения кода вам кое-что понадобится:

Событие перемещения мыши

Событие перемещения мыши сообщает обработчику о том, что указатель мыши переместился над некоторым элементом. Обработчик назначается свойству **onmousemove** элемента. При вызове обработчику передается объект события, который предоставляет следующие свойства:

clientX, clientY: смещение (в пикселах) указателя мыши от левого (и верхнего) края окна браузера.

screenX, screenY: смещение (в пикселах) указателя мыши от левого (и верхнего) края пользовательского экрана.

pageX, pageY: смещение (в пикселах) указателя мыши от левого (и верхнего) края страницы браузера.

Разрази меня гром! Код приведен ниже. Пока он включает страницу с картой и создает элемент абзаца для вывода координат. Вы должны поработать над кодом и сделать так, чтобы он заработал. Желаем удачи! А не то отправитесь на корм рыбам, а у нас еще несколько глав впереди...

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Pirates Booty</title>
  <script>
       window.onload = init;
       function init() {
           var map = document.getElementById("map");
           ________________________________________
       }

       function showCoords(eventObj) {
           var map = document.getElementById("coords");
           ________________________
           ________________________
           map.innerHTML = "Map coordinates: "
                                  + x + ", " + y;
       }
  </script>
</head>
<body>
       <img id="map" src="map.jpg">
       <p id="coords">Move mouse to find coordinates...</p>
</body>
</html>
```

Здесь назначается обработчик события.

Здесь определяются координаты.

Когда ваш код заработает в реальной странице, загрузите его и запишите координаты клада.

____

Еще больше событий

Пока что мы видели три типа событий: событие загрузки (load), происходящее при завершении загрузки страницы браузером; событие щелчка (click) — пользователь щелкает на элементе страницы; и событие перемещения мыши (mousemove) — перемещение мыши над элементом. Скорее всего, вы в своей работе столкнетесь с множеством других событий, например событиями поступления данных по сети, событиями геопозиционирования и хронометражными таймеров (и это лишь несколько примеров).

Для всех событий, которые вы видели, обработчик всегда назначался некоторому свойству — `onload`, `onmouseover`, `onclick` и т. д. Но не все события работают так, например для хронометражных событий программист вместо этого вызывает функцию `setTimeout`, передавая ей свой обработчик.

Пример: допустим, вы хотите, чтобы ваша программа ожидала 5 секунд перед выполнением некоторой операции. Вот как это делается с функцией `setTimeout` и обработчиком:

Сначала мы пишем обработчик события. Этот обработчик будет вызван при возникновении события таймера.

```
function timerHandler() {
    alert("Hey what are you doing just sitting there staring at a blank screen?");
}
```

В этом обработчике мы просто выводим сообщение.

```
setTimeout(timerHandler, 5000);
```

А здесь вызывается функция setTimeout, которая получает два аргумента: обработчик события и интервал времени (в миллисекундах).

Вызов setTimeout можно сравнить с установкой секундомера.

Здесь мы приказываем таймеру ожидать 5000 миллисекунд (5 секунд).

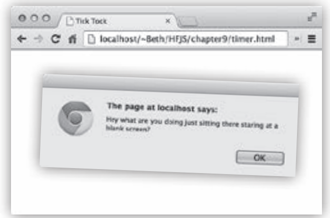
И тогда вызывается обработчик timerHandler.

Как сделать, чтобы сообщение появлялось через каждые 5 секунд снова и снова?

Тестирование таймера

Не сидите сложа руки! Код нужно протестировать! Включите его в простейшую страницу HTML и загрузите страницу в браузере. Сначала вы не увидите ничего, но через 5 секунд появится сообщение.

Будьте терпеливы, пять секунд — и вы увидите то же, что и мы. А если прошла уже пара минут и на экране ничего не появилось, попробуйте дать пинка компьютеру... Шутка. Лучше повнимательнее проверьте свой код.



Как работает setTimeout

Давайте разберемся, что же здесь происходит.

Время пошло. У меня есть таймер, который сработает через 5000 миллисекунд, и обработчик, который вызывается по срабатыванию таймера.

Ваш браузер управляет таймерами.

1. При загрузке страницы мы выполняем две операции: назначаем обработчик с именем timerHandler и вызываем функцию setTimeout для создания события таймера, которое будет сгенерировано через 5000 миллисекунд. Когда сработает событие таймера, обработчик будет выполнен.

Браузер следит за всеми таймерами (да, можно запустить сразу несколько таймеров) и обработчиками, которые должны вызываться по этим таймерам.

2. Пока таймер отсчитывает миллисекунды, браузер продолжает работать как обычно.

5000, 4999, 4998...

..., 6, 5, 4, 3, 2, 1, 0.

Прошло 5000 миллисекунд, таймер сработал — вызываем обработчик.

3. Когда отсчет времени доходит до нуля, браузер вызывает обработчик.

Событие таймера происходит при завершении отсчета времени. Браузер выполняет обработчик события, вызывая переданную вами функцию.

4. Вызванный обработчик создает сообщение и выводит его в браузере.

Обработчик был вызван. Все, этот таймер отработал свое.

```
function timerHandler() {
    alert("Hey what are you doing just sitting there staring at a blank screen?");
}
```

Когда браузер выполняет обработчик события, на экране появляется сообщение!

The page at localhost says:
Hey what are you doing just sitting there staring at a blank screen?
OK

Я что-то не понял этот фокус с setTimeout: мы передали функцию другой функции?

Верно замечено! Помните, мы в самом начале главы говорили, что ваши представления о программировании сильно изменятся? Наступил тот переломный момент, когда черно-белое кино заменяется цветным. Но вернемся к вашему вопросу. Да, мы определили функцию, а потом взяли ее и передали функции `setTimeout` (которая на самом деле является методом).

```
setTimeout(timerHandler, 5000);
```

Ссылка на функцию передается setTimeout (другой функции).

Зачем мы это делаем и что это означает? Взгляните на происходящее так: функция `setTimeout` фактически создает таймер обратного отсчета и связывает с ним обработчик, который вызывается при обнулении таймера. Чтобы сообщить функции `setTimeout`, какой обработчик следует вызывать, нужно передать ей ссылку на функцию-обработчик. `setTimeout` сохраняет ссылку, чтобы использовать ее позднее, когда сработает таймер.

Если вы говорите: «Ага, понятно» — отлично. С другой стороны, возможна и другая реакция: «Простите? Передать функцию функции? Это как?» Вероятно, у вас есть опыт программирования на таких языках, как C и Java, в которых так просто передать функцию не удастся. В JavaScript это возможно; более того, передача функций чрезвычайно полезна, особенно при написании кода, реагирующего на события.

А скорее всего, вы говорите: «Да, я примерно понимаю... Но полной уверенности нет». Если так — не беспокойтесь. Пока просто считайте, что `setTimeout` передается ссылка на обработчик, который должен быть вызван при срабатывании таймера. Мы подробнее расскажем о функциях и о том, что с ними можно делать (например, передавать другим функциям), в следующей главе. Так что пока просто потерпите.

Упражнение

Взгляните на приведенный ниже код и попробуйте определить, что делает вызов `setInterval`. Эта функция похожа на `setTimeout`, но с небольшими отличиями. Сверьтесь с ответом в конце главы.

Это код.

```
var tick = true;
function ticker() {
    if (tick) {
        console.log("Tick");
        tick = false;
    } else {
        console.log("Tock");
        tick = true;
    }
}
setInterval(ticker, 1000);
```

Запишите результаты своего анализа.

Консоль JavaScript
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock

А это результат.

Часто задаваемые вопросы

В: Можно ли остановить отсчет setInterval?

О: Можно. При вызове setInterval вы получаете объект таймера, если его передать функции clearInterval, можно остановить таймер.

В: Вы говорите, что setTimeout — метод, но выглядит как обычная функция. Какому объекту принадлежит этот метод?

О: Хороший вопрос. Формально мы могли бы использовать запись window.setTimeout, но поскольку объект window object считается глобльным, мы можем опустить имя объекта и использовать сокращенную форму setTimeout, которая часто встречается на практике.

В: Можно ли опустить window в синтаксисе window.onload?

О: Можно, но обычно программисты так не делают, потому что onload является достаточно распространенным именем свойства (у других элементов тоже есть свойство onload), и его использование без уточнения может создать путаницу.

В: С onload я назначаю один обработчик для события. Но похоже, setTimeout позволяет назначать сколько угодно обработчиков для любого количества таймеров?

О: Точно. При вызове setTimeout вы создаете таймер и связываете с ним обработчик. Таймеров можно создать сколько угодно. Браузер следит за тем, какой обработчик связан с тем или иным таймером.

В: Где еще используется передача функций функциям?

О: Много где. Более того, передача функций вообще является вполне рядовым приемом в JavaScript. Не только многие встроенные функции (такие, как setTimeout и setInterval) используют передачу функций — вы также будете часто писать код, получающий функции в аргументах. Впрочем, это далеко не полная картина; в следующей главе мы рассмотрим эту тему глубже и узнаем, сколько всего интересного можно проделать с функциями в JavaScript.

Парни, я дорабатываю приложение с картинами. Хочу, чтобы изображения снова размывались через пару секунд после того, как пользователь щелкнет на них.

Джим Фрэнк Джо

Джо: Классная идея. Наверное, ты будешь использовать `setTimeout`?

Фрэнк: Да, собираюсь, хотя пока не знаю, как выбрать изображение для применения размывки.

Джим: О чем ты говоришь?

Фрэнк: Когда пользователь щелкает на изображении и открывает его, я создаю таймер, который срабатывает через две секунды. При срабатывании события таймера вызывается новый обработчик, который я написал, — он называется `reblur`.

Джо: И в `reblur` тебе нужно определить, какое изображение следует размыть?

Фрэнк: Верно. Я не передаю обработчику никакие аргументы, он просто вызывается браузером по истечении времени, поэтому я не могу сообщить обработчику, с каким изображением выполняется операция. В общем, я в тупике.

Джим: А ты изучал программный интерфейс `setTimeout`?

Фрэнк: Нет, я знаю только то, что сказала Джуди: `setTimeout` передается функция и промежуток времени в миллисекундах.

Джим: Ты можешь добавить к вызову `setTimeout` аргумент, который будет передаваться обработчику при срабатывании события таймера.

Фрэнк: О, это замечательно. Значит, я могу просто передать ссылку на изображение, которое нужно снова размыть, и эта ссылка будет передана обработчику при вызове?

Джим: Точно.

Фрэнк: Видишь, Джо, чего можно добиться, просто обсудив код с коллегой?

Джо: Еще бы. Давайте попробуем...

Завершение кода игры

Осталось наложить последние штрихи на игру с размытыми изображениями. Мы хотим, чтобы изображение автоматически восстанавливало размывку через несколько секунд после открытия. И как мы только что узнали, при вызове `setTimeout` можно передать обработчику события аргумент. Давайте посмотрим, как это делается:

```
window.onload = function() {
    var images = document.getElementsByTagName("img");
    for (var i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
    }
};

function showAnswer(eventObj) {
    var image = eventObj.target;
    var name = image.id;
    name = name + ".jpg";
    image.src = name;

    setTimeout(reblur, 2000, image);
}

function reblur(image) {
    var name = image.id;
    name = name + "blur.jpg";
    image.src = name;
}
```

Код, который мы написали ранее. Ничего не изменилось...

Но теперь при выводе нормального изображения мы также вызываем setTimeout для назначения события, которое сработает через две секунды.

В качестве обработчика указывается метод reblur (см. ниже), которому передается значение времени 2000 миллисекунд (две секунды) и аргумент — изображение, которое необходимо снова вывести в размытой версии.

Теперь при вызове обработчик будет получать изображение.

Обработчик получает изображение, определяет его идентификатор и строит имя размытого изображения. Присваивая атрибуту src изображения построенное имя, мы заменяем нормальное изображение размытым.

Будьте осторожны!

setTimeout не поддерживает дополнительные аргументы в IE8 и ранее.

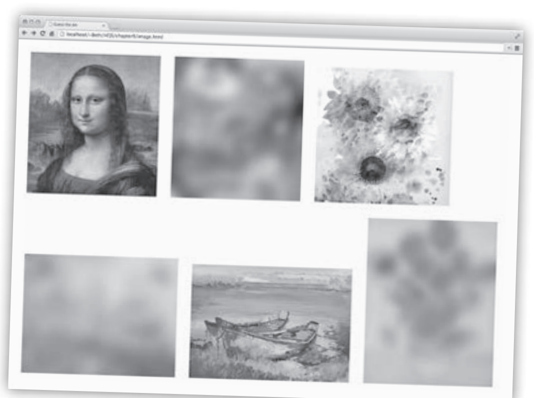
Да, вы правильно поняли. Этот код не будет работать в IE8 и более ранних версиях. Но вам в любом случае не стоило брать IE8 для этой книги! Впрочем, позднее будет дано другое решение, которое справится с проблемой совместимости IE8 (и более ранних версий).

Тест-драйв таймеров

Нового кода не так уж много, но он влияет на работу приложения. Теперь, если щелкнуть на изображении, браузер (через события таймера) следит за тем, когда нужно вызвать обработчик `reblur`. Этот обработчик снова размывает открытое изображение. Обратите внимание на *асинхронность* действий — вы управляете тем, на каких изображениях будут сделаны щелчки, а служебный код активизируется по событиям щелчков и таймеров. Нет никакого супералгоритма, который бы контролировал все происходящее, определяя, что и когда должно вызываться. Программа строится из небольших фрагментов кода, которые реагируют на события.

Теперь изображение, на котором щелкнул пользователь, выводится в нормальном виде, но через две секунды снова размывается.

Протестируйте программу, быстро щелкая на разных изображениях. Всегда ли она работает? Вернитесь к коду и подумайте, как браузер отслеживает все изображения, которые необходимо размыть.



Часто задаваемые вопросы

В: Я могу передать всего один аргумент обработчику setTimeout?

О: Нет, можно передать сколько угодно аргументов: нуль, один и более.

В: Как насчет объекта события? Почему setTimeout не передает его обработчику события?

О: Объекты событий в основном используются с обработчиками событий DOM. Функция setTimeout не передает обработчику объекты событий, потому эти события не происходят ни с каким конкретным элементом.

В: Функция showAnswer — обработчик, но она создает новый обработчик reblur прямо в коде. Это нормально?

О: Вполне. Более того, это часто происходит в коде JavaScript. Обработчики очень часто создают дополнительные обработчики для различных событий. Такой стиль программирования упоминается в начале этой главы: *асинхронное программирование*. Мы не пытаемся реализовать алгоритм, выполняемый в определенной последовательности. Вместо этого мы подключаем обработчики событий, которые управляют процессом выполнения по мере возникновения событий. Проанализируйте щелчки на изображениях и различные вызовы, сопровождающие открытие и размывку изображений.

В: Итак, существуют события DOM, события таймеров... Какие еще?

О: Большинство событий относятся к категории событий DOM (например, щелчок на элементе) или событий таймеров (создаются вызовами setTmeout или setInterval). Существуют события конкретных API, например JavaScript API, относящиеся к Geolocation, LocalStorage, Web Workers и т. д. (обращайтесь к книге *Изучаем программирование на HTML5*). Наконец, существует целая категория событий, относящихся к вводу/выводу, например запрос данных от веб-служб с использованием XmlHttpRequest (также обращайтесь к *Изучаем программирование на HTML5*) или Web Sockets.

Парни, наши пользователи настольных систем хотят, чтобы изображение открывалось без щелчка — простым наведением указателя мыши. Это можно сделать?

Джуди: Для этого необходимо воспользоваться событием наведения указателя мыши (mouseover). Обработчик этого события для любого элемента страницы назначается в свойстве `onmouseover`:

```
myElement.onmouseover = myHandler;
```

Джуди: Кроме того, событие mouseout сообщает о выходе указателя мыши за границы элемента. Обработчик этого события определяется свойством `onmouseout`.

Переработайте свой код, чтобы изображение открывалось и снова размывалось при наведении и выходе указателя мыши за границы элементов. Протестируйте свой код и сверьтесь с ответами в конце главы:

Запишите здесь код JavaScript.

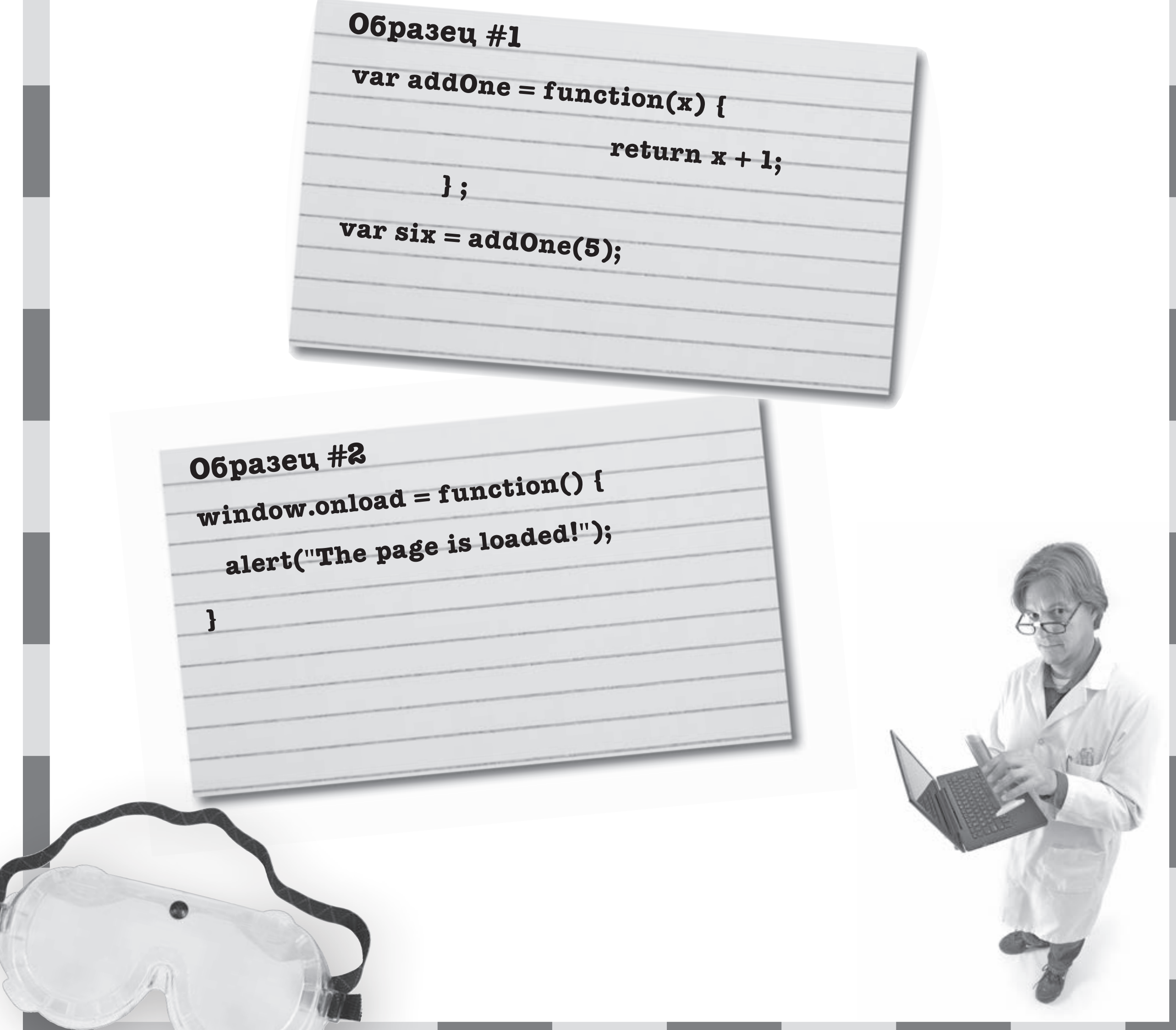
Когда игра с картинками была готова, Джуди написала код для обсуждения на еженедельной встрече группы. Она даже объявила маленький конкурс — первый, кто расскажет, что делает этот код, награждается бесплатным обедом. Кто победит? Джим, Джо, Фрэнк? А может, вы?

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        var element = document.getElementById("display");
        element.innerHTML = element.innerHTML + " that tickles!";
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don't resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

Запишите, что делает этот код. Какие события в нем задействованы? Как создается обработчик? Когда происходит событие? Не ограничивайтесь одними заметками, опробуйте этот код в своем браузере.

Лаборатория кода

Мы нашли крайне подозрительный код, и нам понадобится ваша помощь при тестировании. Мы уже провели предварительный анализ, и хотя на первый взгляд мы имеем дело с абсолютно стандартным JavaScript, что-то кажется странным. Ниже приведены два образца. В каждом случае необходимо указать, что же в коде кажется подозрительным, протестировать код на работоспособность, а потом попытаться проанализировать, что именно делает код. Запишите свои комментарии на странице. Наш анализ приведен на следующей странице.



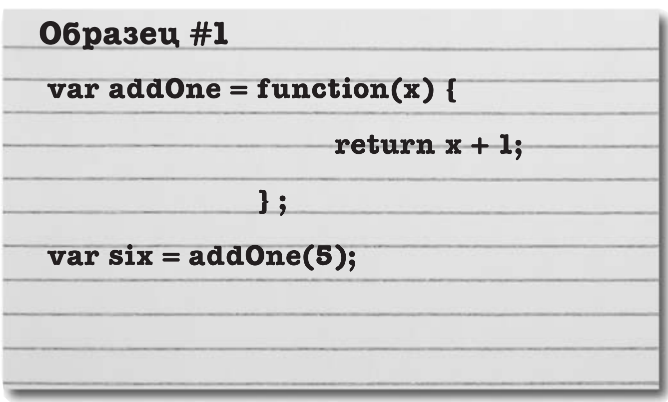
Образец #1

```
var addOne = function(x) {
                    return x + 1;
      };
var six = addOne(5);
```

Образец #2

```
window.onload = function() {
  alert("The page is loaded!");
}
```

Лаборатория кода: анализ



Образец #1

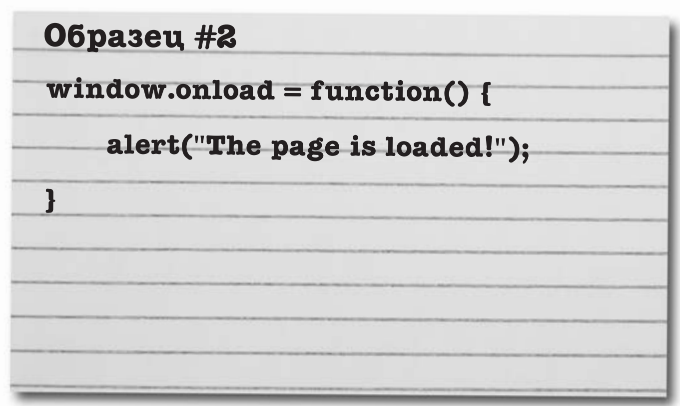
```
var addOne = function(x) {
                        return x + 1;
                  } ;
var six = addOne(5);
```

На первый взгляд этот код просто определяет функцию, которая увеличивает параметр на 1 и возвращает его.

Но если присмотреться поближе, это нельзя назвать обычным определением функции. Мы объявляем переменную и присваиваем ей функцию, которой даже имя не назначено.

Кроме того, функция вызывается по имени переменной, а не по имени функции, входящей в ее определение.

Странно (хотя и немного похоже на то, как определяются объекты).



Образец #2

```
window.onload = function() {
     alert("The page is loaded!");
}
```

Здесь происходит нечто похожее. Вместо того чтобы определять функцию и присваивать ее имя свойству window.onload, мы назначаем функцию непосредственно этому свойству. И снова в определении функции ее имя не указывается.

Мы добавили этот код в страницу HTML и протестировали ее. В целом код работает так, как и ожидалось. С образцом №1 при вызове функции, присвоенной addOne, мы получаем результат, на 1 больший переданного числа, — вроде все правильно. С образцом №2 при загрузке страницы выводится сообщение «The page is loaded!».

По результатам тестирования все выглядит так, словно функции могут определяться без имен и использоваться там, где должно находиться выражение.

Что все это значит? Оставайтесь с нами; в следующей главе мы расскажем об этих странных функциях более подробно...

КЛЮЧЕВЫЕ МОМЕНТЫ

- Большая часть кода JavaScript пишется для обработки **событий**.
- Существует много разных видов событий, на которые может реагировать ваш код.
- Чтобы отреагировать на событие, следует написать функцию — **обработчик события** и зарегистрировать ее. Например, чтобы зарегистрировать обработчик для события щелчка, следует присвоить функцию-обработчик свойству onclick элемента.
- Вы не обязаны обрабатывать какие-либо события в своих программах. Вы сами выбираете события, на которые будете реагировать.
- **Функции** используются для назначения обработчиков, поскольку в них удобно упаковывать код для выполнения в будущем (когда произойдет событие).
- Код, написанный для обработки событий, отличается от кода, который выполняется от начала до конца, а потом завершается. Обработчики событий могут выполняться в любое время и в любом порядке: они выполняются **асинхронно**.
- События, происходящие с элементами DOM (события DOM), передаются в обработчик событий.
- **Объект события** содержит свойства с дополнительной информацией о событии, включая тип (например, “click” или “load”) и источник (объект, с которым произошло событие).
- В старых версиях IE (IE 8 и ранее) используется модель события, отличающаяся от других браузеров.
- Многие события могут происходить с большой частотой. Когда происходит слишком много событий, которые браузер не успевает обрабатывать, события сохраняются в **очереди событий** (в порядке возникновения), чтобы браузер мог последовательно выполнять обработчики всех событий.
- Если обработка события требует значительного объема вычислений, это замедлит обработку событий в очереди, потому что в любой момент времени может выполняться только один обработчик.
- Функции **setTimeout** и **setInterval** используются для генерирования событий таймера по истечении заданного интервала времени.
- Метод **getElementsByTagName** возвращает нуль, один или несколько объектов элементов, объединенных в объект NodeList (аналог массива с возможностью перебора).

Суп с событиями

click
Событие генерируется при щелчке на элементе страницы (или прикосновении).

load
Событие происходит при завершении загрузки страницы браузером.

mousemove
Событие генерируется при перемещении указателя мыши над элементом.

keypress
Событие генерируется при каждом нажатии клавиши.

unload
Событие генерируется при закрытии окна браузера или уходе с веб-страницы.

mouseover
При наведении указателя мыши на элемент генерируется это событие.

mouseout
Событие генерируется при выходе указателя мыши за границы элемента.

resize
Событие генерируется при каждом изменении размера окна браузера.

dragstart
Событие генерируется при перетаскивании элемента на странице.

touchstart
Событие генерируется при прикосновении к элементам на устройствах с сенсорными экранами.

play
Страница содержит элемент <video>? Вы будете получать это событие при нажатии кнопки воспроизведения.

drop
Событие генерируется при отпускании перетаскиваемого элемента.

pause
А это — при нажатии кнопки приостановки.

touchend
Событие генерируется при завершении прикосновения.

Наше знакомство с событиями load, click, mousemove, mouseover, mouseout, resize и timer было крайне поверхностным. Попробуйте этот восхитительный суп с событиями, которые вы встретите в своей работе и которые стоит изучить для веб-программирования.

Возьми в руку карандаш
Решение

Выберите два события из приведенного выше списка. Браузер может оповестить ваш код о возникновении этих событий; какие интересные или нестандартные применения вы бы предложили для их обработки?

Возьмем событие, которое оповещает об отправке данных формы. При получении этого события мы могли бы получить все данные, введенные пользователем на форме, и проверить их на корректность (например, что в поле телефонного номера введена серия цифр, похожая на телефонный номер, или что обязательные поля не остались пустыми). После того как проверка будет успешно выполнена, данные формы можно отправить на сервер.

Как насчет события перемещения мыши? Например, на основе этих событий можно создать графический редактор, работающий прямо в окне браузера.

А с оповещениями о прокрутке страницы можно реализовать такие интересные эффекты, как открытие изображений в процессе прокрутки.

СТАНЬ браузером. Решение

Перед вами код игры. Представьте себя на месте браузера и попробуйте определить, что должно происходить после каждого события. Когда упражнение будет выполнено, загляните в конец главы и проверьте свои ответы. Ниже приведено наше решение.

```
window.onload = init;
function init() {
    var image = document.getElementById("zero");
    image.onclick = showAnswer;
}

function showAnswer() {
    var image = document.getElementById("zero");
    image.src = "zero.jpg";
}
```

Во время загрузки страницы...	*Сначала определяются функции init и showAnswer*
	init назначается обработчиком load
Когда происходит событие завершения загрузки страницы....	*Вызывается обработчик load — init*
	Получаем изображение с идентификатором "zero"
	Обработчиком click изображения назначается showAnswer
Когда происходит событие щелчка на изображении...	*Вызывается метод showAnswer*
	Получаем изображение с идентификатором "zero"
	Атрибуту src задается значение "zero.jpg"

Вы уже знаете, что объект события (для событий DOM) содержит свойства, которые предоставляют расширенную информацию о произошедшем событии. Ниже перечислены другие свойства, которые могут присутствовать в объекте события. Соедините каждое свойство объекта события с его кратким описанием.

target

type

timeStamp

keyCode

clientX

clientY

touches

Хотите узнать, на каком расстоянии от верхней стороны окна браузера был сделан щелчок? Используйте меня.

Я содержу объект, породивший событие. Объекты могут быть разными, но чаще всего это объект, представляющий элемент страницы.

Используете устройство с сенсорным экраном? Я подскажу вам, сколькими пальцами пользователь прикасается к экрану.

Я — строка вида “click” или “load”, которая описывает, что только что произошло.

Интересует, когда произошло событие? Я сообщу вам об этом.

Примечание: timeStamp не поддерживается в старых версиях IE.

Хотите узнать, на каком расстоянии от левой стороны окна браузера был сделан щелчок? Используйте меня.

Я скажу, какую кнопку нажал пользователь.

Эй, салага! У нас тут есть карта сокровищ, но ты должен помочь нам определить координаты клада. Для этого нужно написать фрагмент кода, который выводит координаты при наведении указателя мыши на точку карты.

Разрази меня гром! Код приведен ниже. Пока он включает страницу с картой и создает элемент абзаца для вывода координат. Вы должны поработать над кодом и сделать так, чтобы он заработал. Желаем удачи! А не то отправитесь на корм рыбам, а у нас еще несколько глав впереди... Вот и наше решение.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Pirates Booty</title>
  <script>
       window.onload = init;
       function init() {
           var map = document.getElementById("map");
             map.onmousemove = showCoords;
       }

       function showCoords(eventObj) {
           var map = document.getElementById("coords");
           var x = eventObj.clientX;
           var y = eventObj.clientY;
           map.innerHTML = "Map coordinates: "
                                  + x + ", " + y;
       }
  </script>
</head>
<body>
       <img id="map" src="map.jpg">
       <p id="coords">Move mouse to find coordinates...</p>
</body>
</html>
```

Остается навести указатель мыши на X и получить координаты: 200, 190

Взгляните на приведенный ниже код и попробуйте определить, что делает вызов `setInterval`. Эта функция похожа на `setTimeout`, но с небольшими отличиями. Наш ответ:

Это код.

```
var tick = true;

function ticker() {
    if (tick) {
        console.log("Tick");
        tick = false;
    } else {
        console.log("Tock");
        tick = true;
    }
}

setInterval(ticker, 1000);
```

Консоль JavaScript

```
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
```

А это результат.

Запишите результат своего анализа.

Как и setTimeout, функция setInterval получает два аргумента: функцию-обработчик события и промежуток времени.

Но, в отличие от setTimeout, функция setInterval выполняет обработчик несколько раз... А вернее, бесконечно много раз! (Конечно, вы можете приказать ей остановиться — см. ниже). В этом примере каждые 1000 миллисекунд (1 секунду) функция setInterval вызываем обработчик ticker. Обработчик проверяет значение переменной tick и определяет, нужно ли вывести на консоль сообщение "Tick" или "Tock".

Итак, setInterval генерирует событие при срабатывании таймера, а затем перезапускает таймер.

```
var t = setInterval(ticker, 1000);

clearInterval(t);
```

Чтобы остановить таймер, сохраните результат вызова setInterval в переменной...

...и передайте его clearInterval позднее, когда захотите остановить таймер.

Переработайте свой код, чтобы изображение открывалось и снова размывалось при наведении и выходе указателя мыши за границы элементов. Протестируйте свой код. Наше решение:

```
window.onload = function() {
    var images = document.getElementsByTagName("img");
    for (var i = 0; i < images.length; i++) {
        images[i].onclick = showAnswer;
        images[i].onmouseover = showAnswer;
        images[i].onmouseout = reblur;
    }
};
function showAnswer(eventObj) {
    var image = eventObj.target;
    var name = image.id;
    name = name + ".jpg";
    image.src = name;

    setTimeout(reblur, 2000, image);
}

function reblur(eventObj) {
    var image = eventObj.target;
    var name = image.id;
    name = name + "blur.jpg";
    image.src = name;
}
```

Сначала удаляем назначение обработчика свойству onclick.

Затем назначаем обработчик showAnswer свойству onmouseover изображения...

А теперь функция reblur назначается обработчиком события mouseout (вместо события таймера). Для этого она назначается свойству onmouseout изображения.

Отныне повторная размывка изображения не будет осуществляться по таймеру; вместо этого она будет происходить при отведении указателя мыши от элемента изображения.

Так как теперь reblur используется как обработчик события mouseout, для получения изображения, которое следует размыть, необходимо использовать объект события. Как и в случае с showAnswer, для получения объекта будет использоваться свойство target. Весь остальной код reblur остается неизменным.

Когда игра с картинками была готова, Джуди написала код для обсуждения на еженедельной встрече группы. Она даже объявила маленький конкурс — первый, кто расскажет, что делает этот код, награждается бесплатным обедом. Кто победит? Джим, Джо, Фрэнк? А может, вы?

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        var element = document.getElementById("display");
        element.innerHTML = element.innerHTML + " that tickles!";
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don’t resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

Обработчик события называется resize. При вызове он просто добавляет текст в абзац с идентификатором “display”.

Нас интересует событие изменения размера, поэтому мы создаем функцию-обработчик (с именем resize) и назначаем ее свойству onresize объекта window.

Обработка события определяется в сценарии в конце страницы. Помните, что этот сценарий должен выполняться после полной загрузки страницы, а такое размещение гарантирует, что обработчик не запустится слишком рано.

При изменении размера окна браузера вызывается обработчик resize, обновляющий страницу добавлением нового текста (“ that tickles”) в абзац “display”.

10 первоклассные функции

Функции без ограничений

Изучайте функции и блистайте. В каждом ремесле, искусстве и дисциплине есть ключевой принцип, который отличает игроков «среднего звена» от настоящего профессионала — и когда речь заходит о JavaScript, признаком профессионализма является хорошее понимание **функций**. Функции играют фундаментальную роль в JavaScript, и многие приемы, применяемые при **проектировании и организации** кода, основаны на хорошем знании функций и умении использовать их. Путь изучения функций на этом уровне интересен и непрост, так что приготовьтесь... Эта глава немного напоминает экскурсию по шоколадной фабрике Вилли Вонка — во время изучения функций JavaScript вы увидите немало странного, безумного и замечательного.

Только без пляшущих «умпа-лумпа».

Двойная жизнь ключевого слова function

До сих пор мы объявляли функции следующим образом:

```
function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Quack!");
   }
}

quack(3);
```

Стандартное объявление функции с ключевым словом function, именем, параметром и блоком кода.

Функция вызывается по имени, за которым следует пара круглых скобок со списком необходимых аргументов.

Пока никаких сюрпризов, но давайте определимся с терминологией: формально первая из приведенных выше команд является *объявлением функции*. Объявление создает функцию с именем (в данном случае `quack`), которое может использоваться для *ссылок* и *вызова* функции.

Пока все хорошо, но дальше ситуация усложняется, потому что, как было показано в конце предыдущей главы, ключевое слово function может использоваться и другим способом:

```
var fly = function(num) {
   for (var i = 0; i < num; i++) {
      console.log("Flying!");
   }
};

fly(3);
```

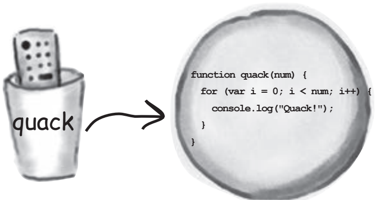
А вот эта конструкция уже не выглядит стандартной: функция не имеет имени и находится в правой части команды присваивания переменной.

Эту функцию тоже можно вызвать — на этот раз через переменную fly.

Такое использование ключевого слова function — внутри команды, как в команде присваивания, — называется *функциональным выражением*. Обратите внимание: в отличие от объявления, эта функция не имеет имени. Кроме того, результатом этого выражения является значение, которое затем присваивается переменной `fly`. Что это за значение? Мы присваиваем его переменной `fly`, а затем вызываем через эту переменную, значит, это должна быть *ссылка на функцию*.

Для любознательных

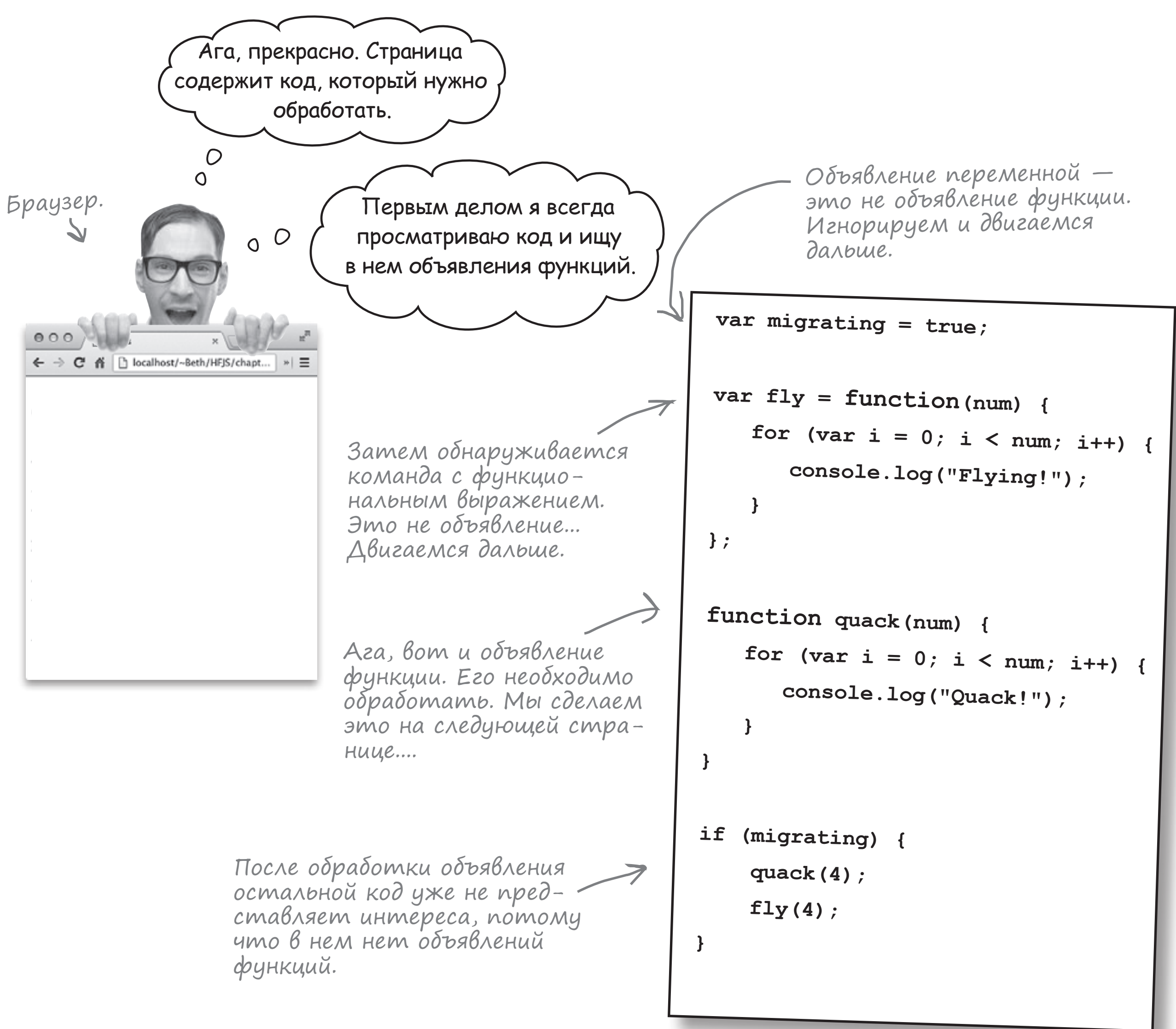
Ссылка на функцию — это... Да в общем именно то, что следует из названия, — ссылка, которая указывает на функцию. Ссылки могут использоваться для вызова функции. Кроме того, их можно присваивать переменным, сохранять в объектах, передавать и возвращать из функций (как и ссылки на объекты).



Объявления функций и функциональные выражения

И с объявлением функции, и с функциональным выражением вы получаете одно и то же: функцию. Тогда чем они отличаются? Может, объявления более удобны? Или в функциональных выражениях есть нечто такое, что делает их более практичными? Или это просто два способа делать одно и то же?

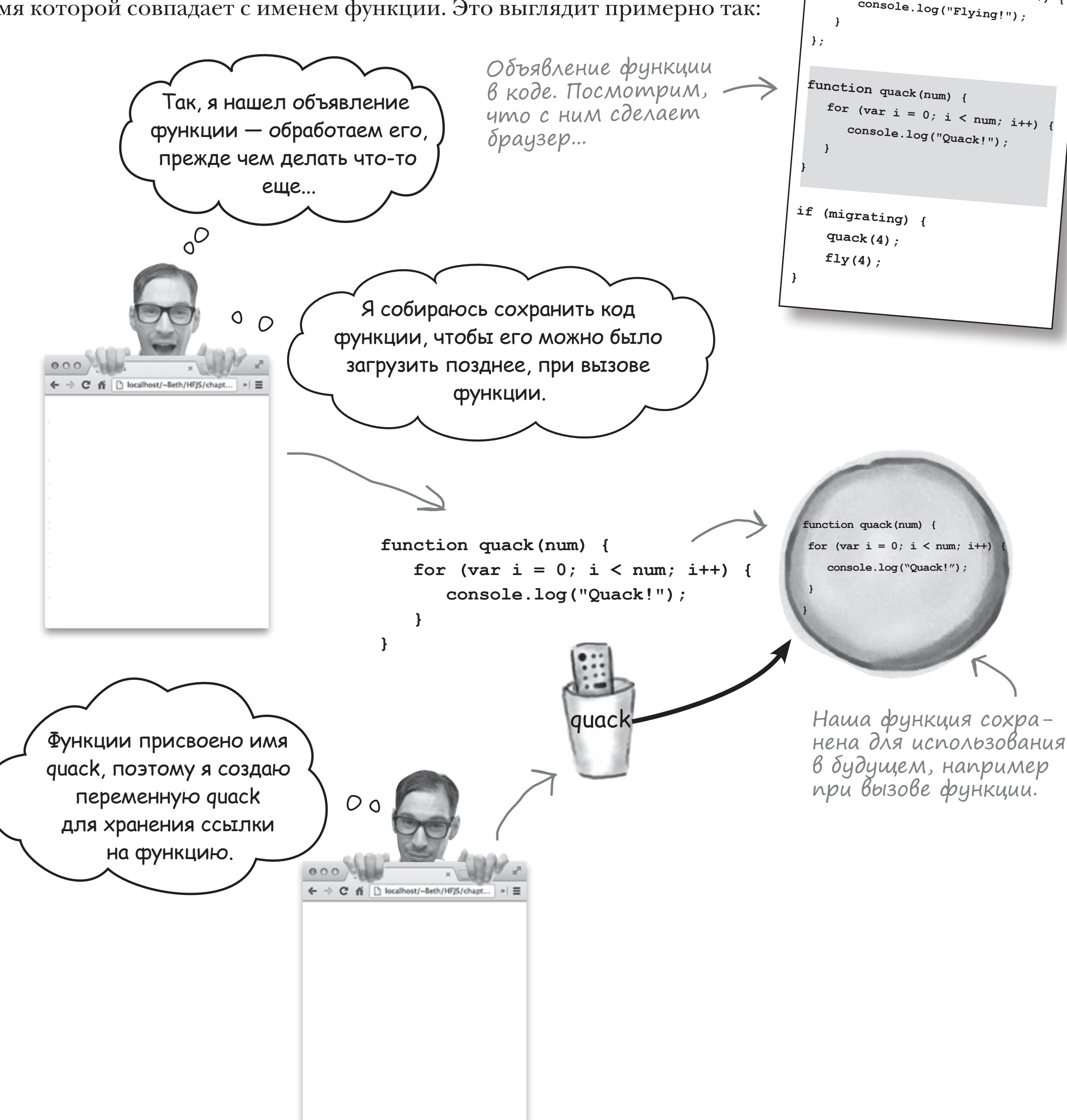
На первый взгляд объявления функций и функциональные выражения в чем-то похожи. Но в действительности между ними существуют весьма принципиальные различия. Чтобы понять суть этих различий, нужно разобраться с тем, как код интерпретируется браузером во время выполнения. Давайте посмотрим, как действует браузер при разборе и обработке кода страницы:




```
var migrating = true;

var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
};

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

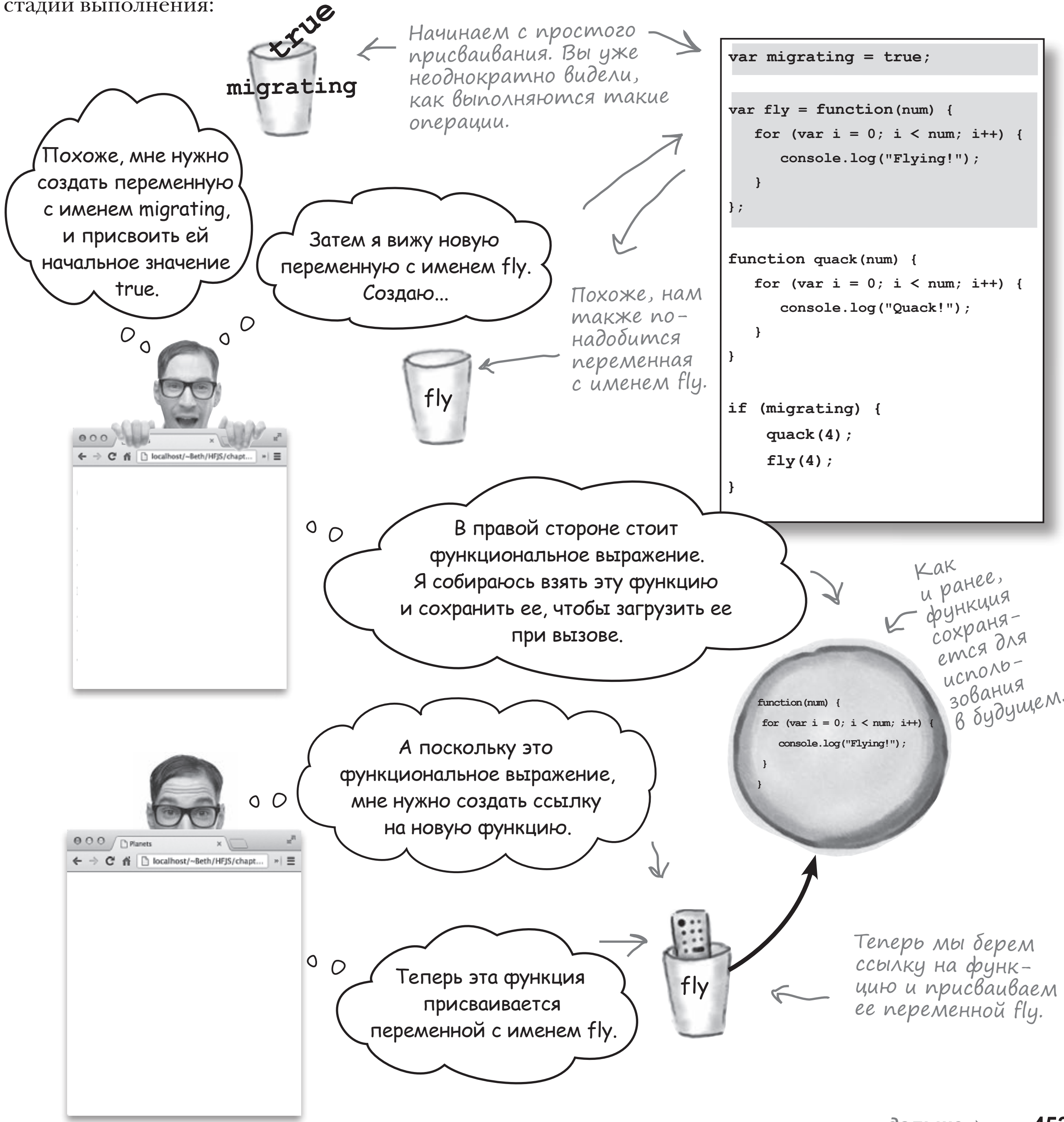
Разбор объявления функции

Когда браузер разбирает вашу страницу (до того, как он займется выполнением кода), он ищет объявления функций. Обнаружив такое объявление, браузер создает функцию и присваивает полученную ссылку переменной, имя которой совпадает с именем функции. Это выглядит примерно так:



Что дальше? Браузер выполняет код

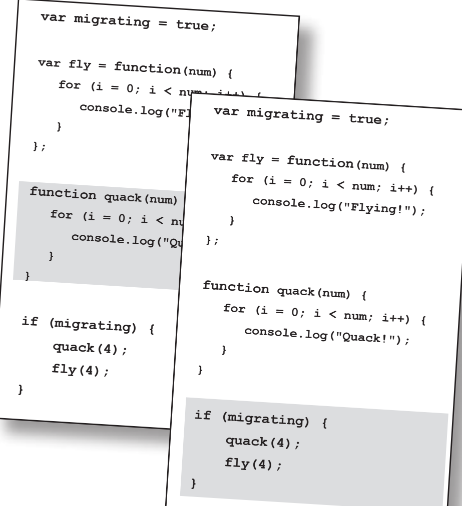
Теперь, когда все объявления функций были обработаны, браузер переходит в начало вашего кода и начинает выполнять его от начала к концу. Давайте посмотрим, что происходит в браузере на этой стадии выполнения:



Двигаемся вперед... Проверка условия

Разобравшись с переменной fly, браузер двигается дальше. Следующая команда содержит объявление функции quack, которое было обработано при первом проходе, поэтому браузер пропускает его и переходит к проверке условия. Смотрим, что будет дальше...

Были здесь, сделали это, двигаемся дальше...



Так, переменная migrating истинна, поэтому я должен выполнить тело команды if. Внутри нее вызывается функция quack. Я знаю, что это вызов функции, потому что в программе указывается имя функции quack, за которым следуют круглые скобки.

```
quack(4);
```

Вспомните, что в переменной quack хранится ссылка на функцию, которая была сохранена ранее...

Функция, созданная по объявлению функции quack.

quack

```
function quack(num) {
 for (var i = 0; i < num; i++) {
   console.log("Quack!");
 }
}
```

В вызове функции присутствует аргумент, его нужно передать функции...

4

```
function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
```

При вызове функции копия значения аргумента передается в параметре...

...после чего выполняется тело функции.

...и выполняем код тела функции, который четыре раза выводит на консоль сообщение “Quack!”.

```
var migrating = true;

var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
};

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

И напоследок...

Остается вызвать функцию fly, созданную функциональным выражением. Давайте посмотрим, как браузер справляется с этой задачей:

Взгляните-ка, еще один вызов функции. Я знаю, что это вызов функции, потому что мы используем имя переменной fly, за которым следуют круглые скобки.

```
fly(4);
```

Вспомните, что переменной fly хранится ссылка на функцию, которая была сохранена ранее...

Функция, на которую ссылается переменная fly.

```
function(num) {
  for (vari = 0; i < num; i++) {
    console.log("Flying!");
  }
}
```

fly

В вызове функции присутствует аргумент, его нужно передать функции...

При вызове функции копия значения аргумента передается в параметре...

4

```
function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}
```

...после чего выполняется тело функции.

Затем я выполняю код тела функции, который четырежды выводит на консоль сообщение “Flying!”.

Возьми в руку карандаш

Какие выводы можно сделать об объявлениях функций и функциональных выражениях на основании того, как браузер обрабатывает код quack и fly? Пометьте истинные утверждения. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

- ❑ Объявления функций обрабатываются раньше остального кода.
- ❑ Функциональные выражения обрабатываются позднее, вместе с остальным кодом.
- ❑ Объявление функции не возвращает ссылку на функцию; вместо этого оно создает переменную с именем функции и присваивает ей новую функцию.
- ❑ Функциональное выражение возвращает ссылку на новую функцию, созданную выражением.
- ❑ Ссылки на функции могут храниться в переменных.
- ❑ Объявления функций являются командами; функциональные выражения используются в командах.
- ❑ Процесс вызова функции, созданной по объявлению, не отличается от процесса вызова функции, созданной по выражению.
- ❑ Объявления — традиционный способ создания функций.
- ❑ Используйте объявления функций там, где это возможно, потому что они обрабатываются на более ранней стадии.

Часто задаваемые вопросы

В: Мы видели выражения вида 3+4 и Math.random() * 6, но как функция может быть выражением?

О: Выражением может быть все, что при вычислении дает некоторое значение. 3+4 дает результат 7, Math.random() * 6 дает случайное число, а функциональное выражение дает ссылку на функцию.

В: Но объявление функции не является выражением?

О: Нет, объявление функции является командой. Считайте, что здесь выполняется скрытое присваивание, которое связывает ссылку на функцию с переменной за вас. Функциональное выражение не присваивает ссылку на функцию ничему; вам приходится делать это самостоятельно.

В: Какая польза от переменной, которая указывает на функцию?

О: Прежде всего, вы можете воспользоваться ею для вызова функции:

```
myFunctionReference();
```

Кроме этого, можно передать ссылку на функцию или вернуть ссылку из функции. Впрочем, не будем забегать вперед — мы вернемся к этому вопросу через несколько страниц.

В: Функциональные выражения могут находиться только в правой части команды присваивания?

О: Вовсе нет. Функциональное выражение может находиться в разных местах, как и другие выражения. Впрочем, это очень правильный вопрос, и мы вскоре вернемся к нему.

В: Хорошо, в переменной может храниться ссылка на функцию. Но на что конкретно эта ссылка указывает? На какой-то код в теле функции?

О: Это хорошая точка зрения на ссылки, но лучше думать о них как об указателях на «замороженный» код, который можно в любой момент извлечь и пустить в дело. Позднее вы увидите, что эти «замороженные» функции содержат нечто большее, чем просто код из своего тела.

Мы только что увидели, что функции, созданные по объявлениям или с использованием функциональных выражений, вызываются абсолютно одинаково. Так чем объявления отличаются от выражений? Мне кажется, мы упустили что-то важное.

Это довольно тонкий момент. Вы правы: и с объявлением функции, и с функциональным выражением вы получаете функцию. Но между ними существует ряд важных отличий. Прежде всего, с объявлением функция создается и подготавливается *до обработки остального кода*. С функциональным выражением функция создается при выполнении кода, *на стадии выполнения*.

Другое отличие связано с именами функций — с объявлением имя функции используется для создания переменной, содержащей ссылку на функцию. Кроме того, при использовании функционального выражения имя функции обычно не указывается, а функция либо присваивается переменной в коде, либо функциональное выражение используется иным образом.

Эти возможности будут описаны позднее в этой главе.

Постарайтесь зафиксировать эти отличия в памяти, потому что мы очень скоро ими воспользуемся. А пока просто запомните, как обрабатываются объявления функций и выражения и что при этом происходит с их именами.

СТАНЬ браузером

Ниже приведен код JavaScript. Представьте себя на месте браузера, который обрабатывает этот код. Справа запишите каждую функцию при ее создании. Не забудьте, что код должен обрабатываться в два прохода: на первом обрабатываются объявления, а на втором — выражения.

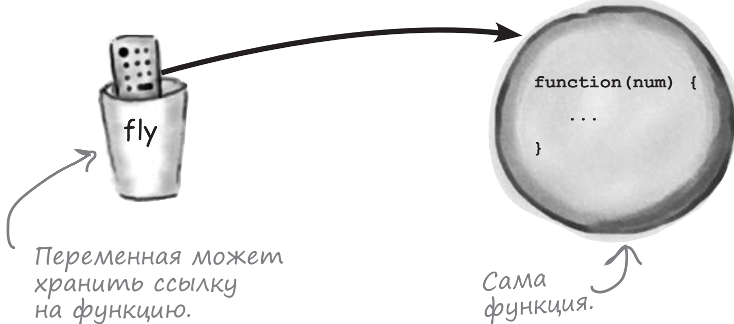
```
var midi = true;
var type = "piano";
var midiInterface;

function play(sequence) {
    // ...
}
var pause = function() {
    stop();
}
function stop() {
    // ...
}

function createMidi() {
   // ...
}

if (midi) {
    midiInterface = function(type) {
        // ...
    };
}
```

Запишите имена функций в последовательности их создания. Если функция создается функциональным выражением, укажите имя переменной, которой оно присваивается. Первую функцию мы уже записали за вас.

play

Функции как значения

Обычно мы представляем себе функции как нечто такое, что можно вызвать, — однако функции также можно рассматривать и как *значения*. Это значение в действительности является ссылкой на функцию, и как вы уже видели, независимо от способа определения функции (через объявление или функциональное выражение) вы получаете ссылку на эту функцию.



Одна из самых тривиальных операций с функциями — присваивание их переменным, как в следующем примере:

```
function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
var fly = function(num) {
   for (var i = 0; i < num; i++) {
      console.log("Flying!");
   }
}


var superFly = fly;
superFly(2);


var superQuack = quack;
superQuack(3);
```

И снова наши знакомые функции. Запомните, что quack определяется объявлением функции, а fly — функциональным выражением. В обоих случаях создаются ссылки на функции, которые сохраняются в переменных quack и fly соответственно.

С объявлением функции JavaScript берет на себя присваивание ссылки переменной с заданным вами именем (в данном случае quack).

С функциональным выражением необходимо присвоить полученную ссылку переменной самостоятельно. В данном случае ссылка сохраняется в переменной fly.

После того как значение из fly будет присвоено superFly, переменная superFly содержит ссылку на функцию. Добавив круглые скобки и аргумент, мы сможем вызвать функцию!

И хотя переменная quack была создана по объявлению функции, хранящееся в quack значение тоже представляет собой ссылку на переменную, поэтому мы можем присвоить его переменной superQuack и вызвать функцию.

Консоль JavaScript

```
Flying!
Flying!
Quack!
Quack!
Quack!
```

Другими словами, ссылка остается ссылкой независимо от того, как она была создана (по объявлению функции или функциональному выражению)!

Возьми в руку карандаш

Чтобы лучше закрепить основные принципы функций в памяти, давайте сыграем в маленькую азартную игру. Знаете, как играют в «наперстки»? Сможете ли вы выиграть? Попробуйте и узнайте.

```
var winner = function() { alert("WINNER!") };
var loser = function() { alert("LOSER!") };
// Простой тест
winner();
// Присваивание ссылок переменным
var a = winner;
var b = loser;
var c = loser;
a();
b();
// Проверяем удачу в игре "наперстки"
c = a;
a = b;
b = c;
c = a;
a = c;
a = b;
b = c;
a();
```

В этих переменных хранятся ссылки на функции winner и loser. Эти ссылки можно присваивать другим переменным, как и любые другие значения.

Помните: функцию можно в любой момент вызвать по ссылке на нее.

Выполните этот код (вручную!) и определите, выиграли вы или проиграли.

Начинайте думать о функциях как о значениях — таких же, как числа, строки, булевские значения или объекты. Функциональные значения отличаются от других прежде всего тем, что эти значения могут использоваться для вызова.

Откровенно о функциях

Интервью недели:
Как работают функции

Head First: Мы так рады, что вы снова в нашей студии. Вас окружают сплошные тайны, и читателям не терпится узнать побольше.

Функция: Это правда, меня так сразу не поймешь.

Head First: Начнем с того, что вас можно создавать при помощи как объявлений, так и выражений. Зачем два способа? Разве одного недостаточно?

Функция: Не забывайте, что эти способы определения функций работают по-разному.

Head First: Но результат ведь один: функция, так?

Функция: Да, но приглядитесь: объявление функции выполняет больше работы за вас. Оно создает функцию, а затем переменную для хранения ссылки на функцию. Функциональное выражение создает функцию, а с создаваемой ссылкой вы должны что-то делать сами.

Head First: Но разве ссылка, полученная от функционального выражения, не всегда сохраняется в переменной?

Функция: Разумеется, нет. Более того, чаще этого не происходит. Помните: функция — это значение. Представьте, сколько всего можно делать с другими видами значений, например ссылками на объекты. Вот и с функциями можно сделать то же самое.

Head First: Но как такое возможно? Я объявляю функцию, я вызываю ее. Практически все, что позволяет любой язык, правильно?

Функция: Неправильно. Думайте о функциях как о значениях, по аналогии с объектами или примитивными типами. Располагая ссылкой на функцию, вы можете делать с ней все, что угодно. Но между функциями и другими видами значений существует различие, в котором проявляется моя особая роль: функцию можно вызвать для выполнения кода, содержащегося в ее теле.

Head First: Звучит впечатляюще, но я пытаюсь представить, что можно еще делать с функциями, кроме как определять и вызывать их?

Функция: В этом-то профессионалы с высокими зарплатами отличаются от «серой массы». Интерпретация функции как любого другого значения делает возможным применение многих интересных программных конструкций.

Head First: Можете привести хотя бы один пример?

Функция: Конечно. Допустим, вы хотите написать функцию, которая может сортировать *что угодно*. Нет проблем, вам понадобится функция, которая получает два аргумента: коллекцию данных, которые нужно отсортировать, и другую функцию, которая умеет сравнивать два элемента в коллекции. На JavaScript такой код написать несложно. Вы пишете одну функцию сортировки для всех разновидностей коллекций, а потом сообщаете ей, как сравнивать данные, передавая функцию, которая умеет выполнять сравнение.

Head First: Эээ...

Функция: Как я и говорю, в этой области проявляется отличие настоящих профессионалов от рядовых программистов. Еще раз: мы *передаем функцию*, которая умеет выполнять сравнение, *другой функции*. Иначе говоря, мы рассматриваем функцию как значение, передавая ее другой функции *как значение*.

Head First: И что нам это дает, кроме путаницы?

Функция: Меньше кода, меньше черной работы, больше надежности, больше гибкости, больше удобства сопровождения... И более высокие зарплаты.

Head First: Все это хорошо, конечно, но я плохо представляю, как разобраться в этой теме.

Функция: Придется немного поработать. Безусловно, это одна из тех областей, для которой вам придется напрячь свои мыслительные способности.

Head First: Я и так напрягаюсь, голова сейчас взорвется. Пожалуй, мне надо прилечь.

Функция: Как скажете. Спасибо за беседу!

Функции как полноправные граждане JavaScript

Программист, переходящий на JavaScript с более традиционного языка, может относиться к функциям... ну как к обычным функциям. Их можно объявлять, можно вызывать, но ни на что другое функции попросту не способны.

Теперь вы знаете, что функции в JavaScript являются значениями — значениями, которые могут присваиваться переменным. И вы знаете, что со значениями других типов (таких, как числа, булевские значения, строки и даже объекты) можно проделывать много полезных операций, например передавать их функциям, возвращать из функций и даже сохранять в объектах или массивах.

У специалистов по информатике для таких значений существует специальный термин: они называются *первоклассными значениями*. С первоклассным значением можно выполнять следующие операции:

- ❑ Присвоить значение переменной (или сохранить его в структуре данных — такой, как массив или объект).
- ❑ Передать значение функции.
- ❑ Вернуть значение из функции.

И знаете что? Все это можно делать и с функциями. Более того, с функциями можно делать все, что можно делать с другими значениями JavaScript. Итак, первоклассные значения функций в JavaScript следует рассматривать наряду с другими значениями уже известных вам типов: чисел, строк, булевских значений и объектов.

Более формальное определение «первоклассности» выглядит так:

Первоклассное значение: значение, с которым в языке программирования могут выполняться те же операции, что и с любыми другими значениями, включая возможность присваивания переменной, передачи в аргументе и возвращения из функции.

FIRST CLASS
FIRST CLASS
FIRST CLASS

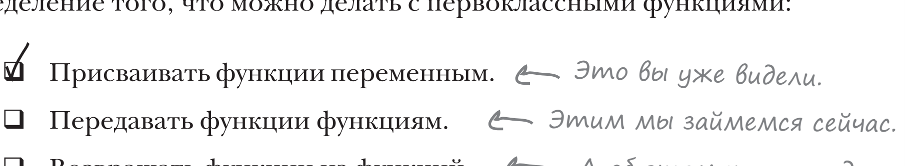
Как видите, функции JavaScript легко удовлетворяют критериям первоклассных значений — давайте выделим немного времени и разберемся, что же означает «первоклассность» функций в каждом из этих случаев. Но сначала небольшой совет: перестаньте думать о функциях как о чем-то особенном и отличающемся от других значений JavaScript. Интерпретация функций как значений открывает исключительные возможности, и в оставшейся части главы мы постараемся вас в этом убедить.

Нам всегда казалось, что «VIP-статус» звучит лучше, но нас никто не слушал, поэтому придется пользоваться термином «первоклассный».

Если в поисках ответа на этот вопрос вы получите престижную работу, не забудьте про нас! Принимаем пожертвования шоколадом, пиццей и биткойнами.

Полеты первым классом

Когда в следующий раз на собеседовании вас спросят: «Что позволяет отнести функции JavaScript к первоклассным значениям?», вы с блеском справитесь с этим вопросом. Но прежде чем праздновать начало новой карьеры, запомните, что до настоящего момента ваши знания первоклассных функций были *чисто теоретическими*. Конечно, вы можете процитировать определение того, что можно делать с первоклассными функциями:



- ☑ Присваивать функции переменным. ← Это вы уже видели.
- ❑ Передавать функции функциям. ← Этим мы займемся сейчас.
- ❑ Возвращать функции из функций. ← А об этом чуть позднее...

Но сможете ли вы использовать эти возможности в своем коде или хотя бы понять, когда это стоит сделать? Не беспокойтесь; сейчас мы как раз займемся этой проблемой и узнаем, как передавать функции функциям. Начнем с простого примера — структуры данных, описывающей пассажиров авиарейса:

Структура данных, представляющая пассажиров:

Все пассажиры хранятся в массиве.

А это данные четырех пассажиров (если хотите, дополните список своими друзьями и знакомыми).

```
var passengers = [  { name: "Jane Doloop", paid: true },
                    { name: "Dr. Evel", paid: true },
                    { name: "Sue Property", paid: false },
                    { name: "John Funcall", paid: true } ];
```

Каждый пассажир представляется объектом со свойствами name и paid.

Свойство name содержит имя в виде простой строки.

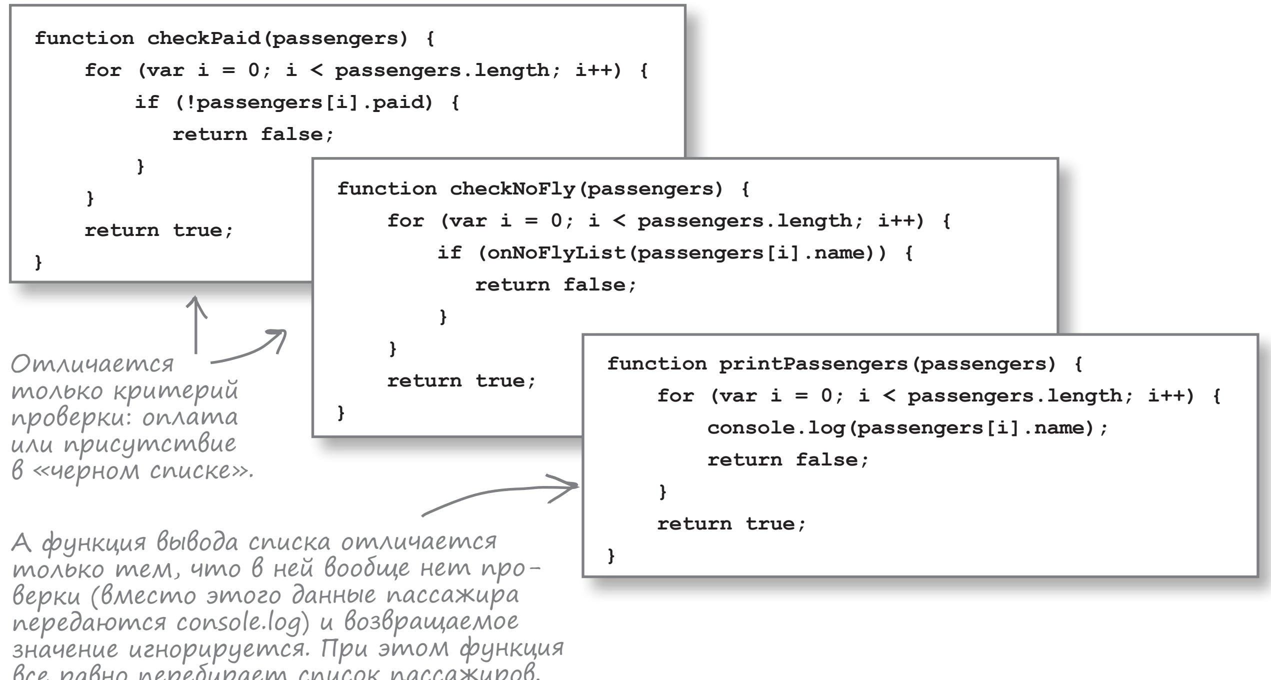
А в свойстве paid хранится булевское значение, которое показывает, заплатил ли пассажир за билет.

Подумайте, как бы вы написали код для решения этих задач (проверка по «черному списку», получение списка пассажиров, оплативших перелет, и получение общего списка пассажиров)?

Наша цель: написать код, который обращается к списку пассажиров и проверяет выполнение некоторых условий, необходимых для вылета, например убеждается в том, что на борту нет пассажиров, занесенных в «черный список». Или что все пассажиры оплатили свой полет... Или, наконец, просто составляет список всех пассажиров рейса.

Написание кода для обработки и проверки пассажиров

Обычно программист пишет функцию для каждого проверяемого условия: одна функция проверяет пассажиров по «черному списку», другая проверяет оплату, третья выводит список всех пассажиров. Но если написать все эти функции, а потом отступить назад и посмотреть на них «на расстоянии», мы увидим, что все они выглядят примерно одинаково:



В этих функциях много повторяющегося кода: все они перебирают список пассажиров и что-то делают с каждым пассажиром. А если в будущем добавятся новые проверки — что у пассажиров имеются проблемы со здоровьем, что они выключили электронику перед взлетом и т. д. Повторяющегося кода станет еще больше.

Или другой, еще худший сценарий: что, если структура данных пассажиров из простого массива объектов превратится во что-то другое? Тогда вам придется открывать каждую функцию и переписывать ее… Нехорошо.

Проблему можно решить при помощи первоклассных функций. Вот как это делается: мы напишем одну функцию, которая умеет перебирать список пассажиров, и передадим ей другую функцию, которая знает, как выполнять нужную проверку (то есть проверить, что имя отсутствует в «черном списке», что пассажир заплатил за билет и т. д.

Упражнение

Давайте проведем небольшую подготовительную работу и напишем функцию, которая получает пассажира в аргументе и проверяет, присутствует ли его имя в «черном списке». Функция возвращает true, если пассажир в «черном списке», или false в обратном случае. Напишите функцию, которая получает объект пассажира и проверяет, не задолжал ли он за билет. Функция возвращает true, если пассажир не заплатил, или false в обратном случае. Начало кода мы написали за вас; вам остается только закончить начатое. Наше решение приведено на следующей странице, но не подглядывайте!

```
function checkNoFlyList(passenger) {




}



function checkNotPaid(passenger) {




}
```

Подсказка: считайте, что «черный список» состоит из одной личности: доктора Зло.

Возьми в руку карандаш

Пора немного потренироваться в передаче функций другим функциям. Просмотрите приведенный ниже код и попробуйте предсказать, что получится при его выполнении. Прежде чем двигаться дальше, проверьте свой ответ.

```
function sayIt(translator) {
    var phrase = translator("Hello");
    alert(phrase);
}

function hawaiianTranslator(word) {
    if (word === "Hello") return "Aloha";
    if (word === "Goodbye") return "Aloha";
}

sayIt(hawaiianTranslator);
```

Перебор пассажиров

Нам нужна функция, которая получает список пассажиров, и другая функция, которая умеет проверять отдельного пассажира по заданному условию (например, присутствие пассажира в «черном списке»). Вот как это делается:

Функция processPassengers получает два параметра. В первом параметре передается массив пассажиров.

Второй параметр содержит функцию, которая умеет проверять некоторое условие для пассажира.

```
function processPassengers(passengers, testFunction) {
    for (var i = 0; i < passengers.length; i++) {
        if (testFunction(passengers[i])) {
            return false;
        }
    }
    return true;
}
```

В цикле последовательно перебираем всех пассажиров.

И вызываем проверочную функцию для каждого пассажира.

Если проверочная функция возвращает true, то мы возвращаем false. Иначе говоря, если пассажир не прошел проверку (не заплатил за билет, входит в «черный список» и т. д.), вылет следует запретить!

Если управление передано в эту точку, значит, все пассажиры прошли проверку, и функция возвращает true.

Теперь осталось написать функции проверки пассажиров (вернее, вы уже написали их в предыдущем упражнении). Вот они:

Обратите внимание: функция получает один объект passenger, а не массив passengers (массив объектов).

```
function checkNoFlyList(passenger) {
    return (passenger.name === "Dr. Evel");
}
```

Эта функция проверяет, входит ли пассажир в «черный список». Наш «черный список» прост: полеты разрешены всем, кроме доктора Зло. Для всех остальных пассажиров возвращается false (то есть пассажир не входит в «черный список»).

```
function checkNotPaid(passenger) {
    return (!passenger.paid);
}
```

А эта функция проверяет, заплатил ли пассажир за билет. Для этого достаточно проверить свойство paid объекта пассажира. Если билет не оплачен, возвращаем true.

Передача функции другой функции

Хорошо, у нас имеется функция, которая готова получить в аргументе другую функцию (`processPassengers`), а также две функции, готовые к передаче в аргументах `processPassengers` (`checkNoFlyList` и `checkNotPaid`).

Пора связать все воедино. Барабанная дробь...

Передать функцию другой функции несложно: просто укажите имя функции как аргумент.

Здесь передается функция checkNoFlyList. Это означает, что функция processPassengers будет проверять каждого пассажира на присутствие в «черном списке».

```
var allCanFly = processPassengers(passengers, checkNoFlyList);
if (!allCanFly) {
    console.log("The plane can't take off: we have a passenger on the no-fly-list.");
}
```

Если хотя бы один из пассажиров присутствует в «черном списке», функция возвращает false и на консоль выводится соответствующее сообщение.

Здесь в аргументе передается функция checkNotPaid. В этом случае processPassengers проверяет, каждый ли пассажир заплатил за билет.

```
var allPaid = processPassengers(passengers, checkNotPaid);
if (!allPaid) {
    console.log("The plane can't take off: not everyone has paid.");
}
```

Если хотя бы один пассажир не заплатил за билет, функция возвращает false и на консоль выводится соответствующее сообщение.

Первым классом всегда лучше... Я про функции, конечно...

Тест-~~драйв~~... Вернее, полет

Чтобы протестировать ваш код, включите следующий фрагмент JavaScript в базовую страницу HTML и загрузите ее в своем браузере.

Консоль JavaScript

```
The plane can't take off: we have a passenger on the no-fly-list.
The plane can't take off: not everyone has paid.
```

Похоже, полет не состоится: у нас проблемы с пассажирами! Хорошо, что проверили заранее...

Ваша очередь: напишите функцию, которая выводит имя пассажира и информацию об оплате, вызовом console.log. Передайте свою функцию processPassengers, чтобы протестировать ее. Ниже мы начали писать код; вам остается завершить его. Прежде чем двигаться дальше, сверьтесь с ответом в конце главы.

```
function printPassenger(passenger) {

}

processPassengers(passengers, printPassenger);
```

Запишите здесь свой код.

Ваш код должен выводить список пассажиров при передаче функции processPassengers.

В: Разве нельзя поместить этот код в processPassengers? С таким же успехом можно включить все нужные проверки в одну итерацию, чтобы при каждом проходе цикла выполнялись все проверки и выводился список. Разве такое решение не будет более эффективным?

О: С простым и коротким кодом — да, это разумный подход. Однако мы в данном случае стремимся к гибкости. Что если в будущем вам придется часто добавлять новые проверки или требования к существующим функциям будут часто изменяться? А если изменится структура данных, представляющая пассажиров? В таких случаях использованное решение позволит вносить изменения способом, который сокращает общую сложность и снижает риск появления ошибок в коде.

В: Что именно передается при передаче функции другой функции?

О: Передается ссылка на функцию. Считайте, что ссылка — это своего рода указатель на внутреннее представление функции. Сама ссылка может храниться в переменной, ее можно присваивать другим переменным и передавать в аргументах функций. Если после ссылки на функцию стоят круглые скобки, то это приводит к вызову функции.

Возьми в руку карандаш

Мы создаем функцию и присваиваем ее переменной fun.

```
function fun(echo) {
    console.log(echo);
};
```

fun

```
function fun(echo) {
  console.log(echo);
}
```

Проанализируйте этот код и запишите результаты его работы на странице. Обязательно проделайте это в уме, прежде чем проверить свои предположения на компьютере.

```
fun("hello");                          ____________

function boo(aFunction) {
    aFunction("boo");
}

boo(fun);                              ____________

console.log(fun);                      ____________

fun(boo);                              ____________

var moreFun = fun;

moreFun("hello again");                ____________

function echoMaker() {
    return fun;
}

var bigFun = echoMaker();
bigFun("Is there an echo?");           ____________
```

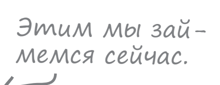
Задание на дополнительный балл! (Дает представление о том, что нас ждет...)

Очень, очень важно: проверьте себя и разберитесь в ответах, прежде чем читать дальше!

Возвращение функций из функций

Мы успели рассмотреть два требования к первоклассным значениям (присваивание функций переменным и передача функций другим функциям), но еще не видели ни одного примера возвращения функций из функций.

- ☑ Присваивать функции переменным.
- ☑ Передавать функции другим функциям.
- ☐ Возвращать функции из функций.



Давайте немного расширим пример с авиарейсом и выясним, почему и где может возникнуть необходимость в возвращении функции из функции. Для этого мы добавим к каждому из пассажиров еще одно свойство ticket, определяющее тип билета, приобретенного пассажиром ("firstclass" — первый класс, "coach" — второй класс):

```
var passengers = [  { name: "Jane Doloop", paid: true, ticket: "coach" },
                    { name: "Dr. Evel", paid: true, ticket: "firstclass" },
                    { name: "Sue Property", paid: false, ticket: "firstclass" },
                    { name: "John Funcall", paid: true, ticket: "coach" } ];
```

С этим дополнением мы напишем код для основных операций, выполняемых стюардессами:

Выбор напитков зависит от класса билета. В первом классе мы предлагаем вино или коктейли; во втором — колу или воду.

Операции, выполняемые стюардессами для обслуживания пассажиров:

```
function serveCustomer(passenger) {
    // Предложить напитки
    // Предложить обед
    // Забрать мусор
}
```

Начнем с реализации заказа напитков.

Как вам известно, уровень обслуживания в первом и втором классе несколько отличается. В первом классе вы сможете заказать коктейль или вино, а во втором предлагают колу или воду.

По крайней мере, так оно выглядит в фильмах...

Код заказа напитков

Первая попытка может выглядеть примерно так:

```
function serveCustomer(passenger) {
    if (passenger.ticket === "firstclass") {
        alert("Would you like a cocktail or wine?");
    } else {
        alert("Your choice is cola or water.");
    }

    // Предложить обед
    // Забрать мусор
}
```

Если пассажир купил билет первого класса, выдаем сообщение с предложением выбрать коктейль или вино.

Купившим билеты второго класса предлагаем колу или воду.

В простых сценариях такое решение работает неплохо: мы проверяем билет пассажира и выводим сообщение в зависимости от типа купленного билета. Но давайте рассмотрим некоторые потенциальные недостатки этого кода. Конечно, код заказа напитков прост, но что произойдет с функцией `serveCustomer`, если задача станет более сложной? Допустим, количество категорий пассажиров увеличилось до трех: первый класс, второй класс и бизнес-класс. А с улучшенным эконом-классом их становится уже четыре!) А если увеличится ассортимент напитков? Или если выбор будет зависеть от аэропорта вылета или назначения?

Например, в рейсах на Гавайи в первом классе обычно подают «Маи Таи» (во всяком случае, нам так рассказывали).

Если нам придется иметь дело с такими сложностями, функция `serveCustomer` быстро разрастется, а ее основной задачей станет принятие решений по напиткам, а не обслуживание клиентов. Функции же рекомендуется проектировать так, чтобы они решали только одну задачу, но делали это очень хорошо.

Перечитайте все потенциальные проблемы, перечисленные в двух последних абзацах на этой странице. Подумайте, какая архитектура кода позволит нам сохранить специализацию serveCustomer, но при этом оставить возможность расширения функциональности предложения напитков на будущее.

Код заказа напитков: другой подход

Первая попытка была неплоха, но со временем усложнение кода заказа напитков может создать проблемы. Давайте немного переработаем этот код, поскольку возможно другое решение: логику заказа напитков можно выделить в отдельную функцию. Это позволит нам скрыть всю логику в одном месте; кроме того, если код заказа напитков изменится, вы совершенно четко знаете, где следует вносить изменения.

Здесь мы создаем новую функцию createDrinkOrder, которой передается объект passenger.

```
function createDrinkOrder(passenger) {
    if (passenger.ticket === "firstclass") {
        alert("Would you like a cocktail or wine?");
    } else {
        alert("Your choice is cola or water.");
    }
}
```

Вся логика заказа напитков размещается в этой функции.

Теперь функция serveCustomer уже не перегружена посторонней логикой выбора напитков.

Вернемся к функции `serveCustomer` и удалим всю логику выбора, заменив ее вызовом новой функции.

```
function serveCustomer(passenger) {
    if (passenger.ticket === "firstclass") {
        alert("Would you like a cocktail or wine?");
    } else {
        alert("Your choice is cola or water.");
    }
    createDrinkOrder(passenger);

    // Предложить обед
    // Забрать мусор
}
```

Удаляем логику из serveCustomer...

Исходная логика, запрограммированная «на месте», заменяется вызовом createDrinkOrder.

Функции createDrinkOrder передается объект пассажира, полученный функцией serveCustomer.

Программа, в которой вся встроенная логика выбора заменяется одним вызовом функции, безусловно, лучше читается. Весь код хранится в одном месте, которое легко найти, и это удобно. Но не торопитесь отправлять код на тестирование — неожиданно возникла еще одна проблема...

Постойте, одного напитка недостаточно!

Нам только что сказали, что одного напитка на рейс явно недостаточно. Стюардесса говорит, что типичный полет выглядит примерно так:

```
function serveCustomer(passenger) {
    createDrinkOrder(passenger);
    // Предложить обед
    createDrinkOrder(passenger);
    createDrinkOrder(passenger);
    // Включить кино
    createDrinkOrder(passenger);
    // Забрать мусор
}
```

Мы внесли изменения, чтобы отразить тот факт, что функция createDrinkOrder вызывается несколько раз за полет.

С одной стороны, наш код спроектирован хорошо, потому что добавление новых вызовов `createDrinkOrder` не создает проблем. С другой стороны, при каждом заказе выполняется лишняя логика определения категории пассажира.

Но это всего несколько строк кода, скажете вы? Безусловно, но ведь это простой учебный пример. А если бы в реальном приложении для проверки вам пришлось бы каждый раз обращаться к веб-службе с мобильного устройства? Такая проверка занимала бы много времени и обходилась слишком дорого.

Не беспокойтесь: первоклассная функция приходит на помощь. Как вы сейчас убедитесь, возможность возвращения функции из функции поможет решить проблему.

Возьми в руку карандаш

Как вы думаете, что делает этот код? Можете привести примеры того, как он должен использоваться?

```
function addN(n) {
    var adder = function(x) {
                    return n + x;
                };
    return adder;
}
```

↑ Запишите ответ.

Заказ напитков с использованием первоклассной функции

Как же первоклассная функция может помочь в этой ситуации? А вот как: вместо того, чтобы многократно вызывать `createDrinkOrder` для каждого пассажира, мы вызываем функцию только один раз. Она должна вернуть функцию, которая знает, как заказывать напитки для этого пассажира. Когда в будущем потребуется заказать напиток, достаточно вызвать эту функцию.

Начнем с переопределения `createDrinkOrder`. При вызове функция упаковывает код заказа напитка в функцию и возвращает эту функцию. В дальнейшем возвращаемая функция используется тогда, когда потребуется заказать напиток.

Новая функция createDrinkOrder возвращает функцию, которая знает, какие напитки следует предлагать пассажиру.

```
function createDrinkOrder(passenger) {
    var orderFunction;

    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like a cocktail or wine?");
        };
    } else {
        orderFunction = function() {
            alert("Your choice is cola or water.");
        };
    }
    return orderFunction;
}
```

Создается переменная для хранения функции, которую мы возвращаем.

Теперь код проверки типа билета пассажира выполняется только один раз.

Если пассажир летит первым классом, создается функция, которая принимает заказы напитков для первого класса.

В противном случае создается функция для заказа напитков пассажирами второго класса.

Наконец, возвращаем функцию.

Теперь переделаем функцию `serveCustomer`. Новая версия сначала вызывает `createDrinkOrder` для получения функции, которая умеет принимать заказ у пассажира, а потом вызывает эту функцию снова и снова для получения заказа у пассажира.

getDrinkOrder теперь возвращает функцию, которая сохраняется в переменной getDrinkOrderFunction.

```
function serveCustomer(passenger) {
    var getDrinkOrderFunction = createDrinkOrder(passenger);
    getDrinkOrderFunction();
    // Предложить обед
    getDrinkOrderFunction();
    getDrinkOrderFunction();
    // Включить кино
    getDrinkOrderFunction();
    // Забрать мусор
}
```

Каждый раз, когда потребуется получить заказ напитков от некоторого пассажира, мы используем функцию, полученную от createDrinkOrder.

Тест-~~драйв~~полет

Давайте протестируем наш новый код. Для этого нужно написать функцию, которая перебирает всех пассажиров и вызывает `serveCustomer` для каждого пассажира. Добавьте код в файл, загрузите его в браузер и примите несколько заказов.

```
function servePassengers(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        serveCustomer(passengers[i]);
    }
}

servePassengers(passengers);
```

По сути мы перебираем пассажиров в массиве passengers и вызываем serveCustomer для каждого пассажира.

И разумеется, нужно вызвать функцию servePassengers, чтобы вся схема заработала. (Приготовьтесь — сообщений будет много!)

The page at localhost says:
Your choice is cola or water.
OK

The page at localhost says:
Would you like a cocktail or wine?
OK

Часто задаваемые вопросы

В: То есть, вызывая createDrinkOrder, мы получаем функцию, которую нужно будет вызвать снова для заказа напитков?

О: Да, верно. Сначала мы вызываем createDrinkOrder для получения функции getDrinkOrderFunction, умеющей запрашивать заказ у пассажира, после чего вызываем эту же функцию каждый раз, когда потребуется принять заказ. Обратите внимание: функция getDrinkOrderFunction намного проще createDrinkOrder; она просто выдает сообщение, в котором предлагается ввести заказ пассажира.

В: Откуда функция getDrinkOrderFunction узнает, какое сообщение нужно вывести?

О: Потому что мы создали ее специально для пассажира на основании его билета. Взгляните на createDrinkOrder еще раз: возвращаемая функция соответствует типу пассажира: если пассажир летит первым классом, то создается версия getDrinkOrderFunction, предлагающая напитки первого класса. Но если пассажир летит вторым классом, то создается getDrinkOrderFunction для напитков этого класса. С возвращением правильной функции для типа билета конкретного пассажира функция заказа остается простой и быстрой, и ее легко вызвать каждый раз, когда потребуется сделать заказ.

В: Этот код предлагает одному пассажиру напиток, включает фильм и т. д. Но разве стюардессы не предлагают напитки всем пассажирам, не включают фильм для всех сразу и т. д.?

О: Ага, мы вас проверяли! И вы прошли проверку. Да, все именно так; этот код применяет всю функцию serveCustomer к одному пассажиру за раз. В реальном мире все не так. Но ведь это всего лишь пример, демонстрирующий сложную тему (возвращение функций), и он не идеален. Но раз уж вы нам *указали на нашу ошибку...* Что ж, возьмите листок бумаги и...

Как бы вы переработали код заказа напитков, обедов и показа фильмов для всех пассажиров без постоянного пересчета их заказов в зависимости от класса билета? Использовали бы вы при этом первоклассные функции?

Возьми в руку карандаш

Добавьте в наш код третий класс обслуживания, допустим, «улучшенный экономический». Пассажирам этого класса кроме лимонада и воды предлагается вино. Также реализуйте функцию заказа обеда getDinnerOrderFunction со следующим меню:

Первый класс: курица или паста

Улучшенный эконом-класс: закуски или сырная тарелка

Второй класс: орешки или сухарики

Сверьтесь с ответами в конце главы! И не забудьте протестировать код.

Обязательно примените первоклассные функции в реализации!

«Веб-кола»

Фирма «Веб-кола» обратилась к нам за помощью в управлении ассортиментом продукции. Для начала взгляните на структуру данных, которая используется для хранения информации о производимых газированных напитках:

Похоже, данные хранятся в массиве объектов. Каждый объект представляет некоторый продукт.

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];
```

Для каждого продукта в объекте хранится название, калорийность, цвет и количество проданных бутылок за месяц.

Помогите нам упорядочить сорта колы. Их нужно отсортировать по всем свойствам: названию, калорийности, цвету, объему продаж. Конечно, сортировка должна выполняться по возможности эффективно; кроме того, она должна быть гибкой, чтобы данные можно было отсортировать множеством разных способов.

Ведущий аналитик фирмы «Веб-кола».

Фрэнк: Парни, нам звонили из «Веб-колы». Нужно помочь им разобраться с данными. Они хотят, чтобы их продукты можно было сортировать по любому свойству — названию, объему продаж, цвету, калорийности и т. д. Но код должен быть гибким на тот случай, если в будущем добавятся новые свойства.

Джо: Как они хранят свои данные?

Фрэнк: Каждый вид газировки представлен объектом в массиве, со свойствами для названия, объема продаж, калорийности...

Джо: Понятно.

Фрэнк: Я думал о простом алгоритме сортировки. У «Веб-колы» продуктов не так много, так что хватит и простого решения.

Джим: У меня есть решение еще проще, но для него нужно знать первоклассные функции.

Фрэнк: Проще — это хорошо, но при чем тут первоклассные функции? По-моему, они только усложняют дело.

Джим: Вовсе нет. Достаточно написать функцию, которая сравнивает два значения, а затем передать ее другой функции, которая выполняет сортировку.

Джо: И как работает функция, которую мы пишем?

Джим: Вместо того чтобы управлять всем процессом сортировки, достаточно написать функцию, которая умеет сравнивать два значения. Допустим, список нужно отсортировать по определенному свойству — скажем, по количеству проданных бутылок. Для этого пишется функция, которая выглядит примерно так:

```
function compareSold(product1, product2) {
  // Код сравнения
}
```

Функция должна получить два продукта, а затем сравнить их.

Подробности мы сейчас рассмотрим, а пока важно другое: такую функцию можно передать функции сортировки, и функция sort выполнит всю работу за вас — ей достаточно знать, как должны сравниваться элементы списка.

Фрэнк: Постой, а где взять эту функцию sort?

Джим: Это метод, который может вызываться для любого массива. Так что ты можешь вызвать метод `sort` для массива продуктов и передать ему функцию сравнения, которую мы собираемся написать. А когда сортировка будет выполнена, массив продуктов будет упорядочен по критерию, использованному `compareSold` при сортировке.

Джо: Хорошо, с сортировкой по объему продаж понятно — это числа. Значит, функция `compareSold` должна просто определить, какое из двух значений меньше (или больше) другого?

Джим: Верно. Давайте-ка познакомимся с тем, как работает сортировка массива...

Как работает метод массивов sort

Массивы JavaScript предоставляют метод `sort`, который получает функцию, сравнивающую два элемента массива, и сортирует массив за вас. Рассмотрим общую концепцию: алгоритмы сортировки хорошо известны, а самое замечательное заключается в том, что код сортировки может повторно использоваться для любого набора элементов. Но возникает одна проблема: чтобы код сортировки мог отсортировать конкретный набор элементов, он должен уметь сравнивать два произвольных элемента. Сопоставьте сортировку набора чисел с сортировкой списка имен или набора объектов. Способ сравнения зависит от типа элементов: для чисел это операторы <, > и == , для строк — алфавит (впрочем, в JavaScript можно использовать операторы <, > и ==), а для объектов может применяться некоторый нестандартный способ сравнения, основанный на значениях их свойств.

Важная подсказка

Массивы JavaScript поддерживают много полезных методов для выполнения разнообразных операций. За перечнем этих методов и примерами их использования обращайтесь к книге *JavaScript. Подробное руководство* Дэвида Фленагана (O'Reilly).

Прежде чем переходить к массиву продуктов «Веб-колы», рассмотрим пример. Мы возьмем простой массив чисел и используем метод sort для упорядочения их по возрастанию. Массив выглядит так:

```
var numbersArray = [60, 50, 62, 58, 54, 54];
```

Теперь нужно написать функцию, которая умеет сравнивать два значения из массива. В нашем случае используется массив чисел, так что функция должна сравнивать два числа. Допустим, числа требуется отсортировать по возрастанию; тогда метод sort ожидает, что если первое число больше второго, функция возвращает положительный результат, при равенстве чисел — 0, а если первое число меньше — отрицательный результат. Примерно так:

Массив состоит из чисел, поэтому функция должна сравнивать два числа.

```
function compareNumbers(num1, num2) {
    if (num1 > num2) {
        return 1;
    } else if (num1 === num2) {
        return 0;
    } else {
        return -1;
    }
}
```

Сначала проверяется условие «num1 больше num2»? Если больше — функция возвращает 1.

Если они равны, функция возвращает 0.

И наконец, если num1 меньше num2, функция возвращает -1.

Возьми в руку карандаш

Функция сравнения, передаваемая sort, возвращает число большее, равное или меньшее 0 в зависимости от сравниваемых значений: если первое значение больше второго, то возвращается число больше 0; если значения равны, возвращается 0; если первое значение меньше второго, то возвращается число меньше 0.

Можно ли на основе этой информации и факта, что мы сравниваем числа в compareNumbers, существенно сократить объем кода?

Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

Все вместе

Теперь, когда функция сравнения написана, остается лишь вызвать метод `sort` для массива `numbersArray` и передать функцию при вызове. Вот как это делается:

```
var numbersArray = [60, 50, 62, 58, 54, 54];
numbersArray.sort(compareNumbers);
console.log(numbersArray);
```

Мы вызываем метод sort для массива и передаем ему функцию compareNumbers.

После вызова sort массив отсортирован по возрастанию. Просто для уверенности выводим его на консоль.

Массив, отсортированный по возрастанию.

Консоль JavaScript

```
[50, 54, 54, 58, 60, 62]
```

Обратите внимание: метод sort является деструктивным, то есть он изменяет исходный массив вместо того, чтобы возвращать новый отсортированный массив.

Упражнение

Метод sort отсортировал numbersArray по возрастанию, потому что при возвращении 1, 0 и -1 мы сообщаем методу sort:

1: первое значение должно располагаться после второго;

0: значения эквивалентны, их можно не переставлять;

-1: первое значение должно располагаться перед вторым.

Переход к сортировке по убыванию сводится к простой инверсии этой логики: 1 означает, что второе значение должно следовать за первым, а -1 — что второе значение должно предшествовать первому (с 0 их порядок не важен). Напишите новую функцию сравнения для сортировки по убыванию:

```
function compareNumbersDesc(num1, num2) {
    if (_____ > _____) {
        return 1;
    } else if (num1 === num2) {
        return 0;
    } else {
        return -1;
    }
}
```

Тем временем в «Веб-коле»

Пора воспользоваться новыми знаниями в области сортировки массивов и помочь «Веб-коле». Конечно, на самом деле достаточно написать функцию сравнения, но сначала давайте еще раз окинем взглядом массив продуктов:

Но говорить им об этом совершенно не обязательно.

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];
```

Помните, что каждый элемент массива products является объектом. Мы не собираемся сравнивать объекты друг с другом — сравниваться должны только конкретные свойства объектов (такие, как sold).

С чего начнем? Допустим, с сортировки по возрастанию количества проданных бутылок. Для этого нужно сравнивать свойство `sold` каждого объекта. При этом необходимо обратить внимание на одно важное обстоятельство: так как речь идет о массиве объектов, функции сравнения будут передаваться два объекта, а не два числа.

compareSold получает два объекта, представляющих виды колы, и сравнивает свойство sold объекта colaA со свойством sold объекта colaB.

```
function compareSold(colaA, colaB) {
    if (colaA.sold > colaB.sold) {
        return 1;
    } else if (colaA.sold === colaB.sold) {
        return 0;
    } else {
        return -1;
    }
}
```

С этой функцией метод sort сортирует данные по возрастанию количества проданных бутылок.

При желании можете упростить этот код по аналогии с предыдущим упражнением.

И конечно, чтобы использовать функцию `compareSold` для сортировки массива `products`, мы просто вызываем метод sort массива `products`:

```
products.sort(compareSold);
```

Помните, что метод sort может использоваться для любых массивов (числа, строки, объекты) и для любых видов сортировки (по возрастанию и по убыванию). Передавая функцию сравнения, мы обеспечиваем гибкость и возможность повторного использования кода.

Тест-драйв сортировки

Пора протестировать первую версию кода «Веб-колы». Ниже собран код с нескольких последних страниц (с небольшими изменениями для тестирования). Итак, создайте простую страницу HTML с приведенным кодом (cola.html) и протестируйте ее:

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
                 { name: "Orange", calories: 160, color: "orange", sold: 12101 },
                 { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
                 { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
                 { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
                 { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
                 { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
                 { name: "Water", calories: 0, color: "clear", sold: 62123 }
               ];

function compareSold(colaA, colaB) {
    if (colaA.sold > colaB.sold) {
        return 1;
    } else if (colaA.sold === colaB.sold) {
        return 0;
    } else {
        return -1;
    }
}

function printProducts(products) {
    for (var i = 0; i < products.length; i++) {
        console.log("Name: " + products[i].name +
                    ", Calories: " + products[i].calories +
                    ", Color: " + products[i].color +
                    ", Sold: " + products[i].sold);
    }
}

products.sort(compareSold);
printProducts(products);
```

Функция сравнения, которая будет передаваться sort...

...а это новая функция, написанная нами для вывода, чтобы список хорошо выглядел на консоли. (Если просто вызвать console.log(products), информация выводится, но выглядит не лучшим образом.)

Сначала сортируем массив products, используя функцию compareSold...

...и выводим результаты.

Консоль JavaScript

```
Name: Grapefruit, Calories: 170, Color: red, Sold: 8200
Name: Raspberry, Calories: 180, Color: pink, Sold: 9427
Name: Root Beer, Calories: 200, Color: caramel, Sold: 9909
Name: Orange, Calories: 160, Color: orange, Sold: 12101
Name: Lemon, Calories: 200, Color: clear, Sold: 14983
Name: Cola, Calories: 210, Color: caramel, Sold: 25412
Name: Diet Cola, Calories: 0, Color: caramel, Sold: 43922
Name: Water, Calories: 0, Color: clear, Sold: 62123
```

Результат выполнения кода с функцией сортировки compareSold. Обратите внимание: продукты упорядочены по объему продаж.

Возьми в руку карандаш

Итак, сортировка по свойству sold успешно заработала. Пора написать функции сравнения для всех остальных свойств объекта продукта: name, calories и color. Внимательно проверьте результаты, выводимые на консоль; убедитесь в том, что для каждого вида сортировки список упорядочивается правильно. Сверьтесь с ответами в конце главы.

Напишите свои решения для трех остальных функций сравнения.

```
function compareName(colaA, colaB) {




}
function compareCalories(colaA, colaB) {




}

function compareColor(colaA, colaB) {




}
products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

Подсказка: операторы <, > и == могут использоваться и для алфавитной сортировки!

Для каждой новой функции сравнения мы вызываем sort и выводим результаты на консоль.

В самую точку!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Существуют два способа определения функций: **объявления функций** и **функциональные выражения**.
- **Ссылка на функцию** — значение, которое может использоваться для обращения к функции.
- Объявления функций обрабатываются до обработки кода.
- Функциональные выражения обрабатываются во время выполнения вместе с остальным кодом.
- Когда браузер обрабатывает объявление функции, он создает функцию и переменную, имя которой совпадает с именем функции, и сохраняет ссылку на функцию в переменной.
- Когда браузер обрабатывает функциональное выражение, он создает функцию, а вы сами решаете, что делать со ссылкой.
- **Первоклассные** значения могут присваиваться переменным, включаться в структуры данных, передаваться функциям и возвращаться ими.
- Ссылка на функцию является первоклассным значением.
- Метод **sort** массивов получает функцию, которая умеет сравнивать два значения из массива.
- Функция, передаваемая методу sort, должна возвращать положительное число, 0 или отрицательное число.

Решение упражнений

Возьми в руку карандаш Решение

Какие выводы можно сделать об объявлениях функций и функциональных выражениях на основании того, как браузер обрабатывает код quack и fly? Пометьте каждое истинное утверждение. Ниже приведены наши ответы.

- [x] Объявления функций обрабатываются раньше остального кода.
- [x] Функциональные выражения обрабатываются вместе с остальным кодом.
- [x] Объявление функции не возвращает ссылку на функцию; вместо этого оно создает переменную с именем функции и присваивает ей новую функцию.
- [x] Функциональное выражение возвращает ссылку на новую функцию, созданную выражением.
- [x] Процесс вызова функции, созданной по объявлению, не отличается от процесса вызова функции, созданной по выражению.
- [x] Ссылки на функции могут храниться в переменных.
- [x] Объявления функций являются командами; функциональные выражения используются в командах.
- [ ] Объявления — традиционный способ создания функций.
- [ ] Используйте объявления функций там, где это возможно, потому что они обрабатываются на более ранней стадии.

Не обязательно!

Стань браузером. Решение

Ниже приведен код JavaScript. Представьте себя на месте браузера, который обрабатывает этот код. Справа запишите каждую функцию при ее создании. Не забудьте, что код должен обрабатываться в два прохода: на первом обрабатываются объявления, а на втором — выражения.

Запишите имена функций в последовательности их создания. Если функция создается функциональным выражением, укажите имя переменной, которой оно присваивается. Первую функцию мы уже записали за вас.

```
var midi = true;
var type = "piano";
var midiInterface;

function play(sequence) {
    // ...
}
var pause = function() {
    stop();
}
function stop() {
    // ...
}

function createMidi() {
   // ...
}

if (midi) {
    midiInterface = function(type) {
        // ...
    };
}
```

play
stop
createMidi
pause
midiInterface

Возьми в руку карандаш
Решение

Чтобы лучше закрепить основные принципы функций в памяти, давайте сыграем в маленькую азартную игру. Знаете, как играют в «наперстки»? Сможете ли вы выиграть? Попробуйте и узнайте. Ниже приведено наше решение.

```
var winner = function() { alert("WINNER!") };
var loser = function() { alert("LOSER!") };
// Простой тест
winner();
// Присваивание ссылок переменным
var a = winner;
var b = loser;
var c = loser;
a();
b();
// Проверяем удачу в игре "наперстки"
c = a;
a = b;
b = c;
c = a;
a = c;
a = b;
b = c;
a();
```

В этих переменных хранятся ссылки на функции winner и loser. Эти ссылки можно присваивать другим переменным, как и любые другие значения.

Помните: функцию можно в любой момент вызвать по ссылке на нее.

c — ссылка на winner
a — ссылка на loser
b — ссылка на winner
c — ссылка на loser
a — ссылка на loser
a — ссылка на winner
b — ссылка на loser
Вызывается... winner!!!

The page at localhost says:
WINNER!
OK

Возьми в руку карандаш. Решение

Пора немного потренироваться в передаче функций другим функциям. Просмотрите приведенный ниже код и попробуйте предсказать, что получится при его выполнении. Наше решение:

```
function sayIt(translator) {
    var phrase = translator("Hello");
    alert(phrase);
}

function hawaiianTranslator(word) {
    if (word == "Hello") return "Aloha";
    if (word == "Goodbye") return "Aloha";
}

sayIt(hawaiianTranslator);
```

Мы определяем функцию, которая получает функцию в аргументе, а затем вызываем эту функцию.

The page at localhost says:
Aloha
OK

Функция hawaiianTranslator передается функции sayIt.

Упражнение. Решение

Ваша очередь: напишите функцию, которая выводит имя пассажира и информацию об оплате, вызовом console.log. Передайте свою функцию processPassengers, чтобы протестировать ее. Мы начали писать код, вам остается завершить его. Ниже приведено наше решение.

```
function printPassenger(passenger) {
    var message = passenger.name;
    if (passenger.paid === true) {
        message = message + " has paid";
    } else {
        message = message + " has not paid";
    }
    console.log(message);
    return false;
}

processPassengers(passengers, printPassenger);
```

Возвращаемое значение не так важно, потому что в данном случае мы игнорируем результат processPassengers.

Консоль JavaScript

```
Jane Doloop has paid
Dr. Evel has paid
Sue Property has not paid
John Funcall has paid
```

Возьми в руку карандаш
Решение

Мы создаем функцию и присваиваем ее переменной fun.

```
function fun(echo) {
    console.log(echo);
};
```

function fun(echo) {
console.log(echo);
}

fun

Предупреждение: ваш браузер может выводить на консоль другие значения для функций fun и boo. Протестируйте это упражнение в паре разных браузеров.

Проанализируйте этот код и запишите результаты его работы на странице. Обязательно проделайте это в уме, прежде чем проверить свои предположения на компьютере.

```
fun("hello");

function boo(aFunction) {
    aFunction("boo");
}

boo(fun);

console.log(fun);

fun(boo);

var moreFun = fun;

moreFun("hello again");

function echoMaker() {
    return fun;
}

var bigFun = echoMaker();
bigFun("Is there an echo?");
```

Ответы:

- fun("hello"); — hello
- boo(fun); — boo
- console.log(fun); — function fun(echo) { console.log(echo); }
- fun(boo); — function boo(aFunction) { aFunction("boo"); }
- moreFun("hello again"); — hello again
- bigFun("Is there an echo?"); — Is there an echo?

Задание на дополнительный балл! (Дает представление о том, что нас ждет...)

Очень, очень важно: проверьте и поймите ответы, прежде чем читать дальше!

Добавьте в наш код третий класс обслуживания, допустим, «улучшенный экономический». Пассажирам этого класса кроме лимонада и воды предлагается вино. Также реализуйте функцию заказа обеда getDinnerOrderFunction со следующим меню:

Первый класс: курица или паста

Улучшенный эконом-класс: закуски или сырная тарелка

Второй класс: орешки или сухарики

Вот наш ответ.

```
var passengers = [  { name: "Jane Doloop", paid: true, ticket: "coach" },
                    { name: "Dr. Evel", paid: true, ticket: "firstclass" },
                    { name: "Sue Property", paid: false, ticket: "firstclass" },
                    { name: "John Funcall", paid: true, ticket: "premium" } ];

function createDrinkOrder(passenger) {
    var orderFunction;
    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like a cocktail or wine?");
        };
    } else if (passenger.ticket === "premium") {
        orderFunction = function() {
            alert("Would you like wine, cola or water?");
        };
    } else {
        orderFunction = function() {
            alert("Your choice is cola or water.");
        };
    }
    return orderFunction;
}
```

Переводим этого пассажира в эконом-класс на этот рейс (для тестирования кода).

Новый код для обработки пассажиров эконом-класса. Теперь мы возвращаем одну из трех разных функций заказа в зависимости от типа билета, купленного пассажиром.

Посмотрите, как удобно объединить всю эту логику в одной функции, которая умеет создавать правильную функцию заказа для пассажира.

При оформлении заказа вся эта логику уже не нужна; у нас уже имеется функция заказа, предназначенная для конкретного пассажира.

Упражнение
Решение

```
function createDinnerOrder(passenger) {
    var orderFunction;
    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like chicken or pasta?");
        };
    } else if (passenger.ticket === "premium") {
        orderFunction = function() {
            alert("Would you like a snack box or cheese plate?");
        };
    } else {
        orderFunction = function() {
            alert("Would you like peanuts or pretzels?");
        };
    }
    return orderFunction;
}
```

Мы добавляем новую функцию createDinnerOrder, которая генерирует функцию заказа обеда для пассажира.

Функция работает по тому же принципу, что и createDrinkOrder: проверяем тип билета пассажира и возвращаем функцию заказа, адаптированную для пассажира.

```
function serveCustomer(passenger) {
    var getDrinkOrderFunction = createDrinkOrder(passenger);
    var getDinnerOrderFunction = createDinnerOrder(passenger);

    getDrinkOrderFunction();

    // Предложить обед
    getDinnerOrderFunction();

    getDrinkOrderFunction();
    getDrinkOrderFunction();
    // Включить кино
    getDrinkOrderFunction();
    // Забрать мусор
}

function servePassengers(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        serveCustomer(passengers[i]);
    }
}

servePassengers(passengers);
```

Мы создаем функцию заказа обеда, предназначенную для этого пассажира...

...а затем вызываем ее всюду, где нужно будет предложить пассажиру обед.

Возьми в руку карандаш
Решение

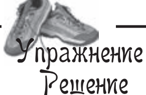
Как вы думаете, что делает этот код? Можете привести примеры того, как он должен использоваться? Ниже приведен наш ответ.

```
function addN(n) {
    var adder = function(x) {
                    return n + x;
                };
    return adder;
}
```

Эта функция получает один аргумент n. Затем она создает функцию, которая также получает один аргумент x, и суммирует n и x. Сгенерированная функция возвращается как результат вызова.

Мы используем ее для создания функции, которая всегда увеличивает число на 2:

```
var add2 = addN(2);
console.log(add2(10));
console.log(add2(100));
```



Упражнение
Решение

Метод sort отсортировал numbersArray по возрастанию, потому что при возвращении 1, 0 и -1 мы сообщаем методу sort:

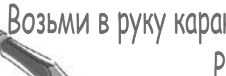
1: первое значение располагается после второго;

0: значения эквивалентны, их можно не переставлять;

-1: первое значение располагается перед вторым.

Переход к сортировке по убыванию сводится к простой инверсии этой логики: 1 означает, что второе значение должно следовать за первым, а -1 — что второе значение должно предшествовать первому (с 0 их порядок не важен). Напишите новую функцию сравнения для сортировки по убыванию:

```
function compareNumbersDesc(num1, num2) {
    if (num2 > num1) {
        return 1;
    } else if (num1 == num2) {
        return 0;
    } else {
        return -1;
    }
}
```



Возьми в руку карандаш
Решение

Функция сравнения, передаваемая sort, возвращает число большее, равное или меньшее 0 в зависимости от сравниваемых значений: если первое значение больше второго, то возвращается число больше 0; если значения равны, возвращается 0; если первое значение меньше второго, то возвращается число меньше 0.

Можно ли на основе этой информации и факта, что мы сравниваем числа в compareNumbers, существенно сократить объем кода?

Вот наше решение:

```
function compareNumbers(num1, num2) {
    return num1 - num2;
}
```

Чтобы сократить эту функцию до одной строки кода, достаточно вернуть результат вычитания num2 из num1. Проверьте на паре примеров — вы поймете, как работает это решение. И помните, что функция сравнения должна возвращать число больше 0, 0 или меньше 0, а не конкретно 1, 0 и -1 (хотя часто встречается код, возвращающий именно эти значения).

Упражнение
Решение

Итак, сортировка по свойству sold успешно заработала. Пора написать функции сравнения для всех остальных свойств объекта продукта: name, calories и color. Внимательно проверьте результаты, выводимые на консоль; убедитесь в том, для каждого вида сортировки список упорядочивается правильно. Ниже приведено наше решение.

Наши реализации всех функций сравнения.

```
function compareName(colaA, colaB) {
    if (colaA.name > colaB.name) {
        return 1;
    } else if (colaA.name === colaB.name) {
        return 0;
    } else {
        return -1;
    }
}

function compareCalories(colaA, colaB) {
    if (colaA.calories > colaB.calories) {
        return 1;
    } else if (colaA.calories === colaB.calories) {
        return 0;
    } else {
        return -1;
    }
}

function compareColor(colaA, colaB) {
    if (colaA.color > colaB.color) {
        return 1;
    } else if (colaA.color === colaB.color) {
        return 0;
    } else {
        return -1;
    }
}

products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

Безусловно!

В самую точку!

Для каждой новой функции сравнения мы вызываем sort и выводим результаты на консоль.

Серьезные функции

Мы узнали много нового о функциях, но это далеко не всё. В этой главе мы пойдем дальше и разберемся в темах, которыми обычно занимаются профессионалы. Вы научитесь **действительно эффективно** работать с функциями. Глава будет не слишком длинной, но с довольно интенсивным изложением материала, так что к концу главы выразительность вашего кода JavaScript превзойдет все ожидания. Вы также будете готовы к тому, чтобы взяться за анализ кода коллеги или заняться изучением библиотеки JavaScript с открытым кодом, потому что мы заодно изучим некоторые распространенные идиомы и соглашения, связанные с использованием функций. А если вы никогда не слышали об **анонимных функциях** и **замыканиях**, то это самое подходящее место для знакомства!

А если вы слыхали о замыканиях, но толком не знаете, что это такое, — так это еще более подходящее место!

Посмотрим на функции с другой стороны...

Вы уже видели две стороны функций — формальную, декларативную сторону объявлений функций и более свободную и выразительную сторону функциональных выражений. Пришло время открыть еще одну интересную сторону: *анонимную*.

Анонимными называются функции, которые *определяются без имени*. Как такое возможно? Когда вы определяете функцию в объявлении функции, у этой функции *безусловно существует имя*. Но при определении функции с использованием функционального выражения *присваивать функции имя не обязательно*.

Что вы говорите? «Факт, конечно, интересный, но что из того?» Дело в том, что анонимные функции часто делают код более лаконичным, более четким, удобочитаемым и эффективным и даже упрощают его сопровождение.

Итак, давайте посмотрим, как создавать и использовать анонимные функции. Начнем с кода, который уже встречался ранее, и посмотрим, как улучшить его с помощью анонимных функций:

Обработчик события загрузки страницы — назначается точно так же, как мы это делали в прошлом.

Сначала определяется функция. У этой функции есть имя handler.

```
function handler() { alert("Yeah, that page loaded!"); }
window.onload = handler;
```

Затем мы назначаем функцию свойству onload объекта window, указывая ее имя handler.

При загрузке страницы будет вызываться функция handler.

Возьми в руку карандаш

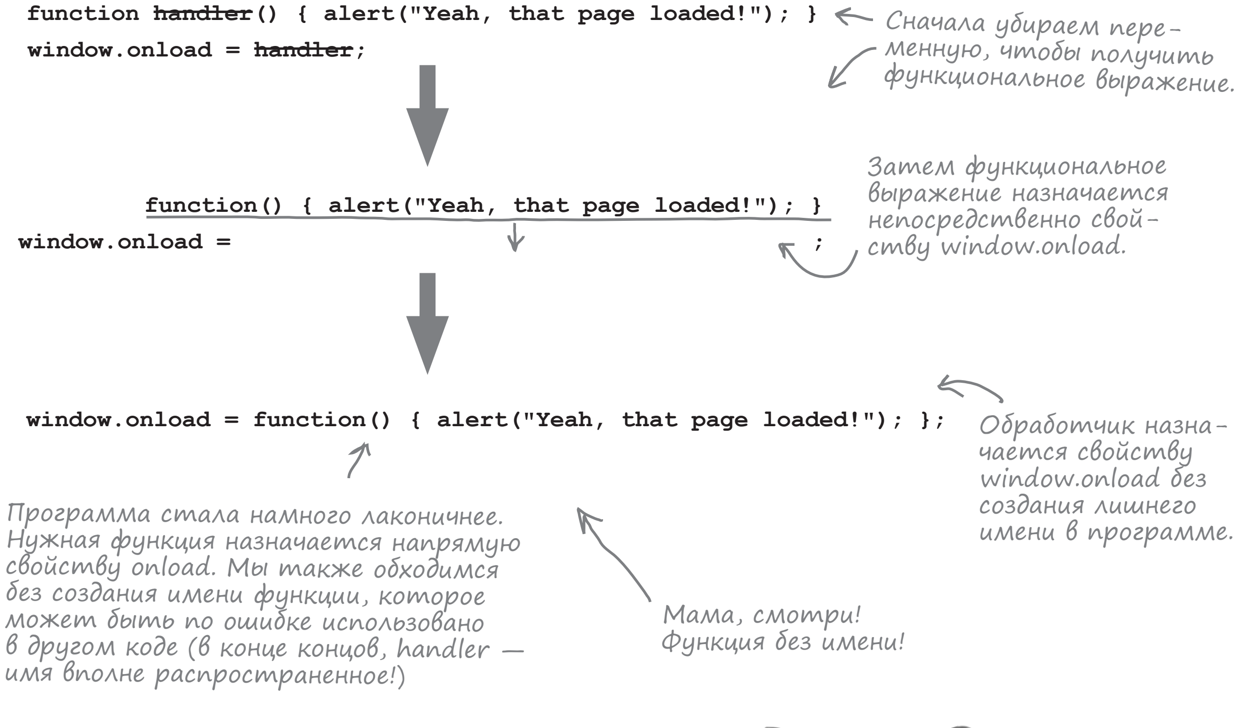
Используя свои знания в области функций и переменных, пометьте истинные утверждения.

- ❑ В переменной handler хранится ссылка на функцию.
- ❑ При назначении обработчика свойству window.onload присваивается ссылка на функцию.
- ❑ Переменная handler существует только для того, чтобы быть назначенной свойству window.onload.
- ❑ Переменная handler больше нигде не используется — этот код должен выполняться только при исходной загрузке страницы.
- ❑ Повторный вызов обработчиков onload нежелателен — он может создать проблемы, поскольку обработчики обычно выполняют операции, относящиеся к инициализации всей страницы.
- ❑ Функциональные выражения создают ссылки на функции.
- ❑ А мы упоминали о том, что при назначении обработчика window.onload присваивается ссылка на функцию?

Как использовать анонимную функцию?

Итак, мы создаем функцию для обработки события загрузки страницы, но эта функция совершенно точно будет «одноразовой» — ведь событие происходит только один раз для каждой загрузки страницы. Также мы видим, что свойству `window.onload` присваивается ссылка на функцию, а именно ссылка, хранящаяся в `handler`. Но поскольку функция `handler` является одноразовой, создавать для нее переменную излишне, потому что она используется для единственной цели: для присваивания ссылки, хранящейся в этой переменной, свойству `window.onload`.

Анонимные функции упрощают подобный код. Анонимная функция представляет собой функциональное выражение без имени, которое используется там, где обычно должна располагаться ссылка на функцию. Посмотрите, как функциональное выражение используется в коде для создания анонимной функции:



Не вспомните ли вы, где ранее мы использовали анонимные функции, не подозревая об этом?

Подсказка: может, они прятались где-то в объектах?

Возьми в руку карандаш

В приведенном ниже фрагменте есть несколько мест, в которых уместно воспользоваться анонимными функциями. Переработайте этот код и внесите необходимые исправления. Там, где потребуется, вычеркните старый код и запишите новые фрагменты. Да, и еще одна задача: обведите кружком все анонимные функции, используемые в этом коде.

```
window.onload = init;
var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
              console.log("Baking the cookies.");
              setTimeout(done, time);
          }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

Нельзя ли покороче...

Ладно, нам не хочется поднимать этот вопрос, потому что вы значительно продвинулись в работе с функциями — вы умеете передавать их, присваивать переменным, возвращать их из функций — и все же ваши программы остаются чуть более многословными, чем необходимо (можно сказать, они не так выразительны, как могли бы быть). Пример:

Вполне обычная функция с именем cookieAlarm выводит сообщение о том, что печенье пора вынимать из духовки.

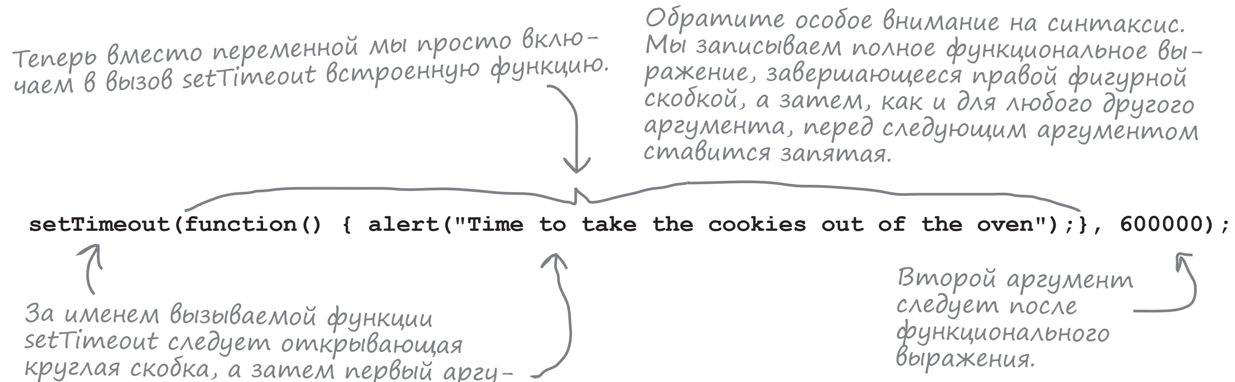
```
function cookieAlarm() {
    alert("Time to take the cookies out of the oven");
};

setTimeout(cookieAlarm, 600000);
```

Ставим таймер на 10 минут.

Здесь мы берем функцию и передаем ее в аргументе setTimeout.

На всякий случай напомним: значение задается в миллисекундах, 1000 * 60 * 10 = 600 000.

Код смотрится нормально, но анонимные функции позволяют сделать его немного короче. Как? Взгляните на переменную `cookieAlarm`, использованную при вызове `setTimeout`. Эта переменная содержит ссылку на функцию, передаваемую при вызове `setTimeout`. Да, для получения ссылки на функцию можно воспользоваться переменной, но как и в примере с `window.onload` пару страниц назад, с таким же успехом можно применить функциональное выражение. Давайте перепишем этот код с функциональным выражением:



Кого вы пытаетесь обмануть? Да здесь же ничего не поймешь. Кто захочет читать одну длинную строку? А если функция будет длинной и сложной?

Для таких коротких функций запись в одну строку подойдет. Но в целом вы правы, читать длинную функцию будет неудобно. Но как вы знаете, в код JavaScript можно включать дополнительные пробелы; мы можем использовать отступы и переносы строк, чтобы код лучше читался. Вот как выглядит переформатированная версия кода `setTimeout` с предыдущей страницы.

Мы всего лишь добавили несколько пробелов.

```
setTimeout(function() {
              alert("Time to take the cookies out of the oven");
          }, 600000);
```

Хорошо, что вы спросили об этом, потому что теперь код читается намного лучше.

Погодите — кажется, я начинаю понимать. Так как в результате обработки функционального выражения получается ссылка на функцию, я могу подставить функциональное выражение в любом месте, где должна находиться ссылка?

Многословно, но в целом верно. На самом деле это один из ключевых моментов для понимания роли функций как первоклассных значений. Если ваш код ожидает получить ссылку на функцию, вы всегда можете разместить на этом месте функциональное выражение — потому что в результате его вычисления получится ссылка на функцию. Как вы уже видели, если функция должна передаваться в аргументе — не проблема, передайте функциональное выражение (которое пересчитывается в ссылку перед передачей). Если функция должна возвращаться из функции, ситуация та же — просто верните функциональное выражение.

Упражнение

Давайте убедимся в том, что вы правильно поняли синтаксис передачи анонимных функциональных выражений другим функциям. Преобразуйте следующий код, чтобы вместо переменной (в данном случае vaccine) в нем использовалось анонимное функциональное выражение.

```
function vaccine(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}

administer(patient, vaccine, time);
```

Запишите здесь свою версию. И обязательно проверьте свой ответ, прежде чем двигаться дальше!

Часто Задаваемые Вопросы

В: Все эти анонимные функции — какая-то экзотика. Мне действительно необходимо это знать?

О: Да. Анонимные функциональные выражения часто встречаются в коде JavaScript, поэтому если вы хотите читать код других людей или понимать библиотеки JavaScript, вы должны знать, как работает этот синтаксис, и узнавать его при использовании.

В: Анонимные фукциональные выражения действительно улучшают код? Мне кажется, что они только усложняют его и затрудняют его чтение и понимание.

О: Не торопитесь. Со временем вы начнете быстрее понимать такой код, а во многих случаях анонимный синтаксис упрощает код, более наглядно выражает его смысл и делает его более элегантным. Впрочем, злоупотребление анонимными функциями бесспорно приводит к появлению кода, сложного для понимания. Будьте настойчивы — когда вы уловите суть, такой код станет более вразумительным и полезным. Код, использующий анонимные функции, вам будет встречаться очень часто, так что стоит включить этот прием в ваш творческий арсенал.

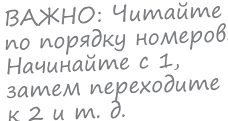
В: Если первоклассные функции настолько полезны, то почему их нет в других языках?

О: Вообще-то есть (и даже там, где нет, их собираются добавить). Например, в таких языках, как Scheme и Scala, реализована поддержка первоклассных функций на уровне JavaScript. Другие языки — такие, как PHP, Java (в последней версии), C# и Objective C, — поддерживают некоторые (или многие) возможности первоклассных функций, реализованные в JavaScript. Чем больше людей понимают полезность поддержки первоклассных функций в языке программирования, тем большим количеством языков они поддерживаются. Впрочем, каждый язык делает это по-своему, так что приготовьтесь к тому, что в других языках эти средства будут реализованы несколько иначе.

Когда определяется функция? Здесь возможны варианты...

С функциями связана еще одна тонкость. Помните, что браузер обрабатывает код JavaScript в два прохода: на первом проходе разбираются все объявления функций; на втором проходе браузер выполняет код от начала к концу, тогда же определяются функциональные выражения. Из-за этого функции, созданные по объявлениям, определяются *до* функций, созданных с использованием функциональных выражений. А этот факт, в свою очередь, определяет, где и как вы сможете вызывать функции в своем коде.

Чтобы понять, что это означает на практике, рассмотрим конкретный пример. Ниже приведен код из предыдущей главы, расположенный в слегка ином порядке. Давайте проанализируем его:



1 Начинаем от первой строки кода и находим все объявления функций.

4 Снова начинаем с первой строки — на этот раз происходит выполнение кода.

```
var migrating = true;
```

5 Создать переменную migrating и присвоить ей true.

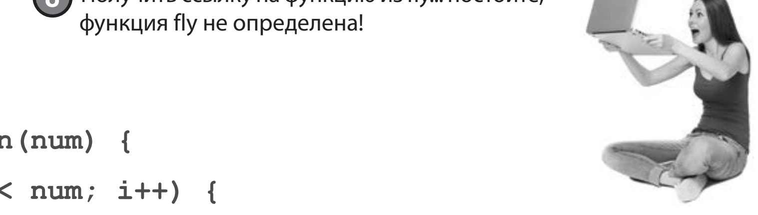
Обратите внимание: это условие переместилось наверх от конца кода.

```
if (migrating) {
    quack(4);
    fly(4);
}
```

6 Условие истинно, выполняем программный блок.

7 Получить ссылку на функцию из quack, вызвать ее с аргументом 4.

8 Получить ссылку на функцию из fly... постойте, функция fly не определена!



```
var fly = function(num) {
   for (i = 0; i < num; i++) {
      console.log("Flying!");
   }
};
```

2 Найдено объявление функции. Создаем функцию и присваиваем ее переменной quack.

```
function quack(num) {
   for (i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
```

3 Достигнут конец кода. Обнаружено только одно объявление функции.

Что произошло? Почему функция fly не определена?

Итак, при попытке вызвать функцию `fly` выясняется, что она не определена, но почему? Ведь с `quack` все прошло нормально. Как вы, вероятно, уже поняли, в отличие от функции `quack`, — определяемой на первом проходе, поскольку это объявление функции, — функция `fly` определяется при стандартном выполнении кода от начала к концу. Посмотрите еще раз:

При попытке вызова quack в ходе выполнения этого кода все работает как ожидалось, потому что функция quack определяется при первом проходе.

```
var migrating = true;
if (migrating) {
  quack(4);
  fly(4);
}
var fly = function(num) {
  for (var i = 0; i < num; i++) {
    console.log("Flying!");
  }
};
function quack(num) {
  for (var i = 0; i < num; i++) {
    console.log("Quack!");
  }
}
```

Но при попытке вызова функции fly происходит ошибка, потому что функция fly еще не определена...

...потому что функция fly остается неопределенной вплоть до выполнения этой команды, которая следует уже после вызова fly.

```
Консоль JavaScript
Quack!
Quack!
Quack!
Quack!
TypeError: undefined is not a function
```

Что происходит при попытке вызова неопределенной функции.

Возможно, вы уже видели подобные ошибки ранее (в зависимости от используемого браузера): TypeError: свойство 'fly' объекта [object Object] не является функцией.

Что же это означает? Прежде всего то, что вы можете размещать объявления функций в любой точке кода — в начале, в конце, в середине — и вызывать их тогда, когда считаете нужным. Объявления функций создают функции, которые могут определяться в любой точке кода.

С функциональными выражениями дело обстоит иначе, потому что они остаются неопределенными до момента их обработки. Итак, даже если функциональное выражение присваивается глобальной переменной, как это было сделано с `fly`, вы не сможете использовать эту переменную для вызова до того, как функция будет определена.

В этом примере обе функции имеют *глобальную область действия* — то есть обе функции после определения видны в любой точке кода. Но нам также необходимо учитывать вложенные функции (то есть функции, определяемые внутри других функций), потому что они влияют на область действия этих функций. Давайте посмотрим, о чем идет речь.

Как создаются вложенные функции

Ничто не мешает нам определять функции внутри других функций; это означает, что объявления функций или функциональные выражения можно использовать только внутри этих функций. Как работает вложение? Очень просто: различия между функцией, определяемой на верхнем уровне кода, и функцией, определяемой внутри другой функции, сводятся к области действия имен. Другими словами, размещение функции внутри другой функции влияет на видимость этой функции внутри вашего кода.

Чтобы лучше понять сказанное, давайте немного расширим наш пример и добавим несколько вложенных объявлений функций и функциональных выражений.

```
var migrating = true;
var fly = function(num) {
    var sound = "Flying";
    function wingFlapper() {
        console.log(sound);
    }
    for (var i = 0; i < num; i++) {
        wingFlapper();
    }
};
function quack(num) {
    var sound = "Quack";
    var quacker = function() {
        console.log(sound);
    };
    for (var i = 0; i < num; i++) {
        quacker();
    }
}
if (migrating) {
     quack(4);
     fly(4);
}
```

Здесь добавляется объявление функции с именем wingFlapper внутри функционального выражения fly.

А здесь эта функция вызывается.

Здесь добавляется функциональное выражение, присваиваемое переменной quacker внутри объявления функции quack.

А здесь эта функция вызывается.

Этот код перемещен в конец, чтобы избежать ошибки при вызове fly.

Упражнение

Возьмите карандаш и пометьте в этом коде границы области действия функций fly, quack, wingFlapper и quacker. Кроме того, отметьте те места, где, по вашему мнению, эти функции находятся в области действия, но при этом остаются неопределенными.

Как вложение влияет на область действия

Функции, определяемые на верхнем уровне кода, обладают глобальной областью действия, тогда как функции, определяемые внутри других функций, обладают локальной видимостью. Сейчас мы пройдемся по этому коду и рассмотрим область действия каждой функции, а заодно и уточним, в каких частях кода каждая функция определена (или не определена, если хотите):

```
var migrating = true;
var fly = function(num) {
   var sound = "Flying";
   function wingFlapper() {
      console.log(sound);
   }
   for (var i = 0; i < num; i++) {
      wingFlapper();
   }
};
function quack(num) {
   var sound = "Quack";
   var quacker = function() {
      console.log(sound);
   };
   for (var i = 0; i < num; i++) {
      quacker();
   }
}
if (migrating) {
    quack(4);
    fly(4);
}
```

Все, что определяется на верхнем уровне кода, имеет глобальную область действия. Следовательно, переменные fly и quack являются глобальными.

Но помните: функция fly определяется только после того, как это функциональное выражение будет обработано.

Функция wingFlapper определяется объявлением в функции fly. Ее областью действия является вся функция fly, и она определена в любой точке тела функции fly.

Функция quacker определяется функциональным выражением в функции quack. Ее областью действия является вся функция quack, однако определена она только на участке от функционального выражения до конца тела функции.

Функция quacker определена только здесь.

Для функций, определяемых внутри функций, по возможности обращения к функции действуют те же правила, что и на верхнем уровне. Другими словами, внутри функции при определении вложенной функции *посредством объявления* эта вложенная функция определена в любой точке тела функции. С другой стороны, если при создании вложенной функции используется *функциональное выражение*, то эта вложенная функция определена только после обработки функционального выражения.

Часто задаваемые вопросы

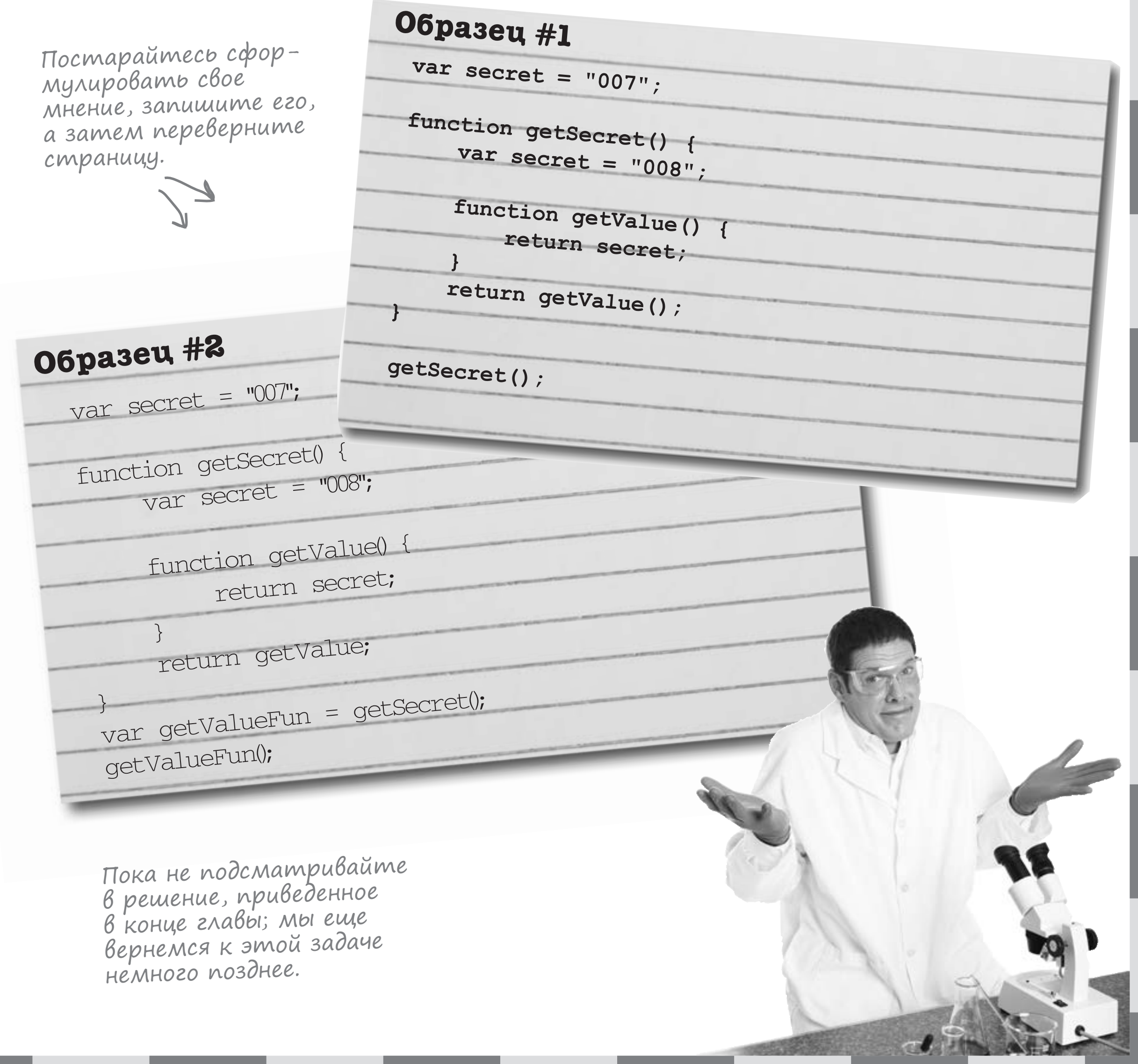
В: При передаче функционального выражения другой функции эта функция сохраняется в параметре, а затем рассматривается как локальная переменная в той функции, которой она передается. Верно?

О: Совершенно верно. При передаче функции в аргументе передаваемая ссылка копируется в переменную-параметр вызываемой функции. И параметр, содержащий ссылку на функцию, является локальной переменной (как, впрочем, и любой другой параметр).

ДЛЯ НАСТОЯЩИХ ПРОФЕССИОНАЛОВ

Нам нужен эксперт по первоклассным функциям — и нам порекомендовали вас! Ниже приведены два фрагмента кода; помогите нам разобраться с тем, что делает каждый фрагмент. Мы в полном тупике — эти фрагменты нам кажутся совершенно одинаковыми, только в одном используется первоклассная функция, а в другом нет. На основании того, что мы знаем об областях действия JavaScript, мы ожидали, что образец №1 даст результат 008, а образец №2 — результат 007. Тем не менее в обоих случаях результат равен 008! Помогите нам разобраться, почему это происходит?

Постарайтесь сформулировать свое мнение, запишите его, а затем переверните страницу.



Образец #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}

getSecret();
```

Образец #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Пока не подсматривайте в решение, приведенное в конце главы; мы еще вернемся к этой задаче немного позднее.

В двух словах о лексической области действия

Этот термин означает, что для определения области действия переменной достаточно прочитать структуру кода (то есть для ее определения не нужно дожидаться выполнения программы).

Раз уж речь зашла об областях действия, давайте посмотрим, как работает лексическая область действия:

Глобальная переменная с именем justAVar.

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    return justAVar;
}

var result = whereAreYou();
console.log(result);
```

Эта функция определяет новую лексическую область действия...

...в которой существует локальная переменная justAVar, замещающая глобальную переменную с тем же именем.

При вызове этой функции возвращается переменная justAVar. Но какая? Мы используем лексическую область действия, поэтому значение justAVar определяется областью действия ближайшей функции. И если найти переменную там не удается, то поиск переходит к глобальной области действия.

Итак, при вызове whereAreYou возвращается значение локальной переменной justAVar, а не глобальной.

```
Консоль JavaScript
Just an every day LOCAL
```

Теперь добавим вложенную функцию:

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner();
}

var result = whereAreYou();
console.log(result);
```

Та же функция.

И замещающая переменная.

Но теперь мы имеем дело с вложенной функцией, которая обращается к переменной justAVar. Но какой именно? И снова выбирается переменная из ближайшей внешней функции. Итак, будет использоваться та же переменная, что и в прошлый раз.

Здесь мы вызываем внутреннюю функцию и возвращаем ее результат...

Следовательно, при вызове whereAreYou вызывается внутренняя функция, которая возвращает значение локальной переменной justAVar, а не глобальной.

```
Консоль JavaScript
Just an every day LOCAL
```

Чем интересна лексическая область действия

Давайте внесем еще одно исправление. Следите внимательно:

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    function inner() {
         return justAVar;
    }

    return inner;
}

var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

Здесь ничего не изменилось, те же переменные и функции.

Но вместо того чтобы вызывать внутреннюю функцию, мы возвращаем ее.

При вызове whereAreYou мы получаем ссылку на внутреннюю функцию inner, которая присваивается переменной innerFunction. Затем мы вызываем innerFunction, сохраняем вывод в переменной result и выводим ее значение.

Какая же переменная justAVar будет использована здесь при вызове inner (под личиной innerFunction)? Локальная или глобальная?

Здесь важен момент вызова функции. Мы вызываем inner после возврата управления, когда в области действия находится глобальная версия justAVar, поэтому будет выдано сообщение «Oh don't worry about it, I'm GLOBAL».

Не так быстро... С лексической областью действия важна программная структура, в которой определяется функция, поэтому результатом должно быть значение локальной переменной, то есть «Just an everyday LOCAL».

Фрэнк: Что значит «ты права»? Это нарушение законов физики... или чего-нибудь в этом роде. Локальная переменная уже не существует... Я хочу сказать, при выходе из области действия локальная переменная прекращает существовать. Она дезинтегрируется! Фильм «ТРОН» видела?

Джуди: Может, в ваших немощных языках C++ и Java... но не в JavaScript.

Джим: Серьезно? Функция `whereAreYou` пропала, а локальная версия `justAVar` больше не существует.

Джуди: Если бы ты послушал, что я тебе только что сказала... В JavaScript все работает не так.

Фрэнк: Так подскажи нам, Джуди. Как же оно работает?

Джуди: При определении внутренней функции локальная переменная `justAVar` находится в области действия этой функции. С лексической областью действия *каждый раз, когда вызывается* `inner`, локальная переменная находится поблизости на тот случай, если она понадобится функции.

Фрэнк: Понятно, но как я уже сказал, это нарушает законы физики. Функция `whereAreYou`, определившая локальную версию переменной variable, больше не существует.

Джуди: Верно. Функции `whereAreYou` нет, но ее область действия вокруг `inner` может использоваться.

Джим: Это как?

Джуди: Давайте посмотрим, что происходит при определении и возвращении функции...

ОТ РЕДАКТОРА: Как Джо успел поменять рубашку, пока перелистнул страницу?!

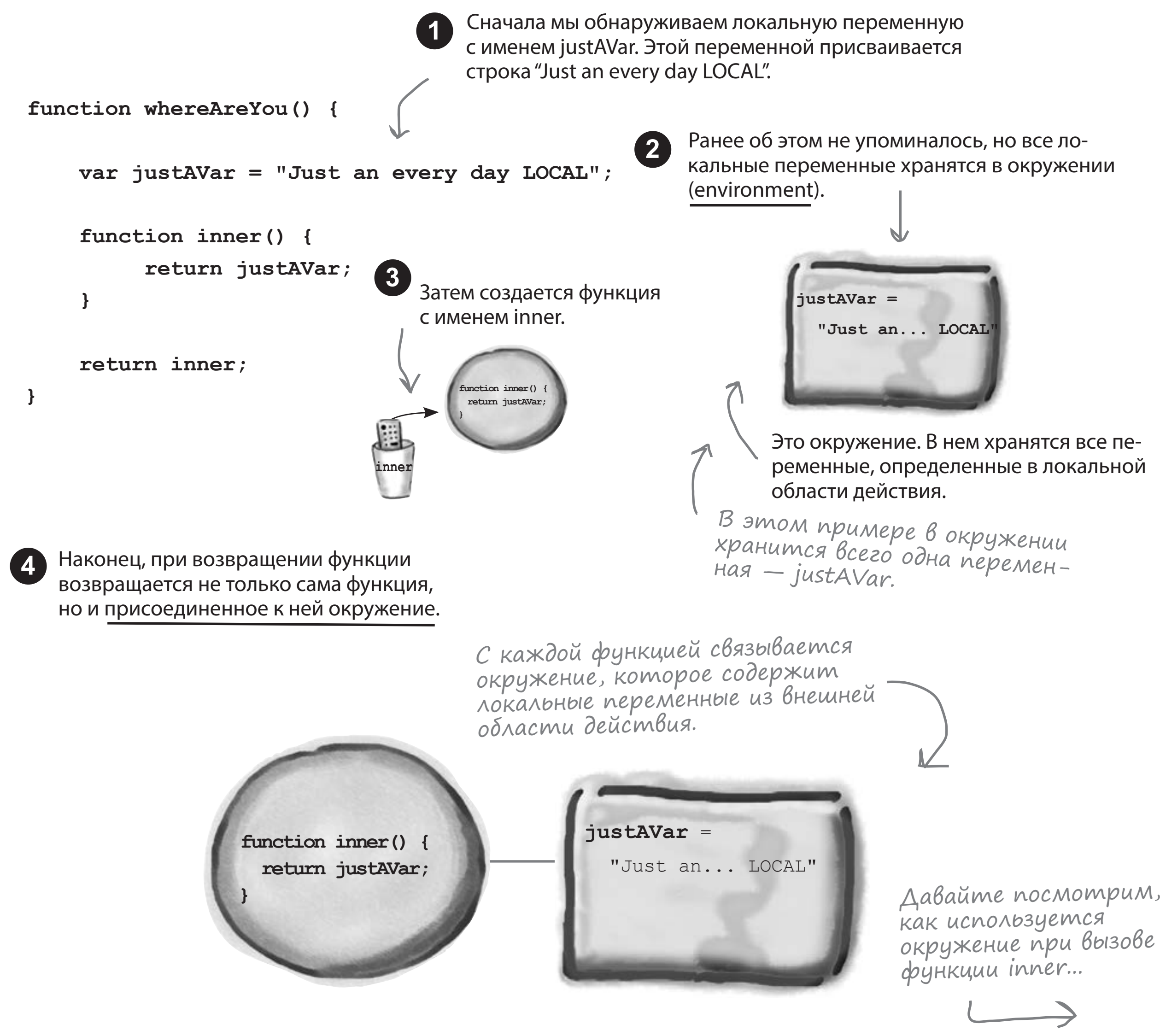
Снова о функциях

Должны вам признаться: мы до сих пор еще не рассказали *всего* о функциях. Даже когда вы спросили: «На что на самом деле указывает ссылка на функцию?», мы постарались обойти этот вопрос. «Лучше думать о них как об указателях на “замороженный” код, который можно в любой момент извлечь и пустить в дело», — сказали мы.

Пришло время раскрыть все секреты.

Для этого мы посмотрим, что же в действительности происходит во время выполнения с этим кодом, начиная с функции `whereAreYou`:

```
function whereAreYou() {

    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner;
}
```

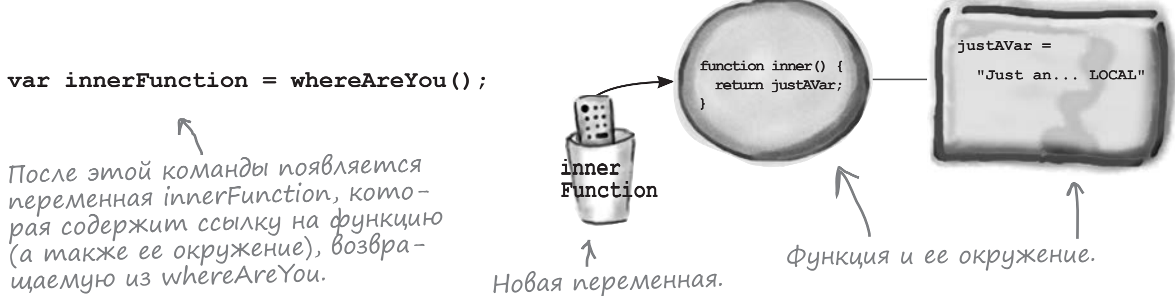


Вызовы функций (снова)

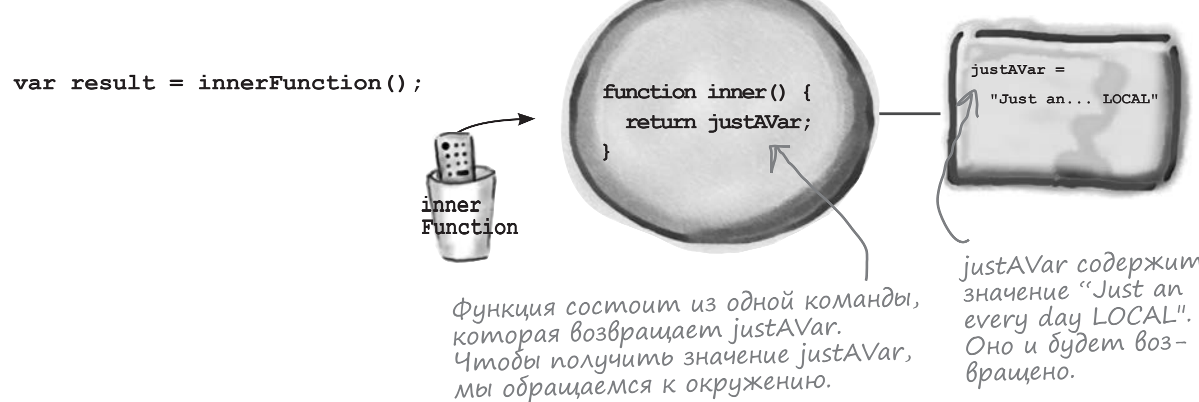
Итак, у нас имеется функция `inner` и ее окружение. Давайте вызовем `inner` и посмотрим, что при этом происходит. Мы хотим выполнить следующий фрагмент:

```
var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

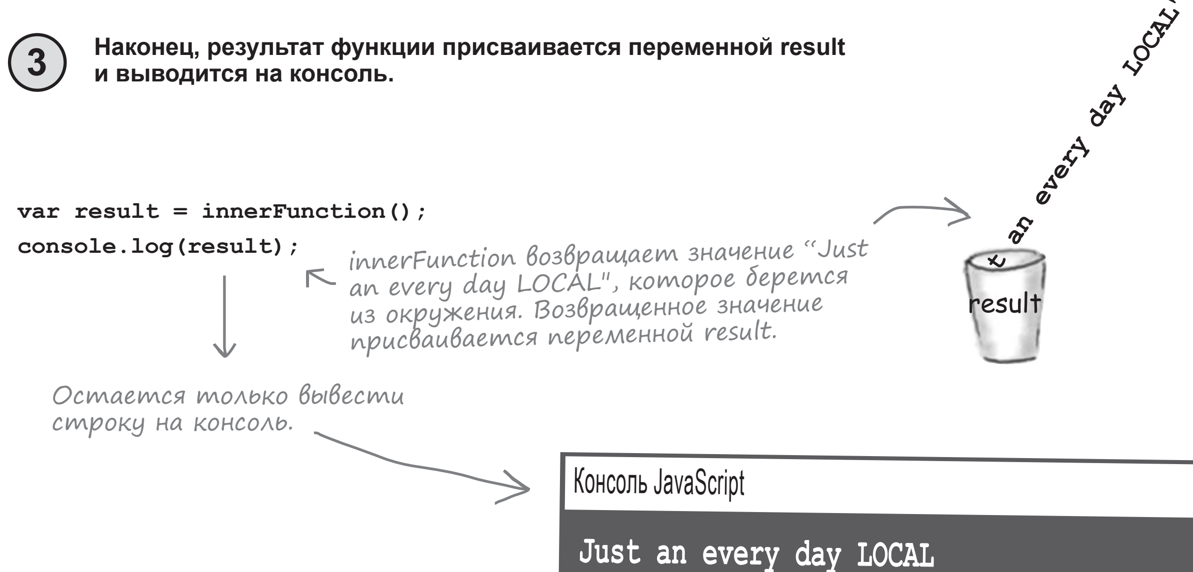
1 **Сначала вызывается whereAreYou. Мы уже знаем, что она возвращает ссылку на функцию, поэтому создаем переменную innerFunction и присваиваем ей полученную функцию. Помните, что ссылка на функцию связана со своим окружением?**



2 **Затем вызывается innerFunction. Для этого мы обрабатываем код тела функции, причем делаем это в контексте ее окружения:**



3 **Наконец, результат функции присваивается переменной result и выводится на консоль.**



Вот черт! Джуди снова права.

Погодите секунду... Джуди не упоминала о замыканиях? Кажется, они как-то связаны с тем, что мы делаем. Давайте-ка изучим эту тему и поквитаемся с ней.

Кстати, парни... это называется ЗАМЫКАНИЕ. Вам стоит что-нибудь почитать на эту тему.

В: Что вы имеете в виду, когда говорите, что место определения переменной определяется лексической областью действия?

О: Под лексической областью действия имеется в виду, что правила областей действия JavaScript зависят исключительно от структуры кода (а не от каких-то динамических свойств времени выполнения). Это означает, что место определения переменной можно узнать, просто изучив структуру своего кода. Также следует помнить, что в JavaScript только функции вводят новые области действия. Итак, располагая ссылкой на переменную, ищите место ее определения в функциях от наибольшего уровня вложенности (где она используется) к наименьшему. А если не удастся найти определение в функции, значит, переменная является глобальной или неопределенной.

В: Как работает окружение для функций, вложенных на несколько уровней?

О: Мы использовали для объяснения упрощенную модель окружения, но вы можете считать, что каждая вложенная функция имеет собственное окружение с собственными переменными. Далее остается лишь создать цепочку окружений для всех вложенных функций, от внутренних к внешним.

Итак, поиск переменной начинается с ближайшего окружения, а затем проходит по цепочке, пока переменная не будет найдена. А если найти ее не удалось — поиск переходит к глобальному окружению.

В: Зачем нужны лексические области действия и окружения? Я думал, что в том простом примере ответом будет строка «Oh, don't you worry about it, I'm GLOBAL». По крайней мере это логично. Правильный ответ выглядит странно и противоестественно.

О: Да, на первых порах действительно может сложиться такое впечатление, однако у лексической области действия есть одно важное преимущество: вы всегда можете просмотреть код, узнать область действия, в которой определяется переменная, и по ней определить значение переменной. И как вы уже видели, это утверждение истинно даже при возвращении функции и ее последующем вызове за пределами исходной области действия. Впрочем, есть и другой довод в пользу лексической области действия. Вскоре мы доберемся до него.

В: Переменные-параметры тоже включаются в окружение?

О: Да. Как было сказано ранее, с параметрами в функциях можно работать как с локальными переменными, поэтому они тоже включаются в окружение.

В: Обязательно ли во всех подробностях понимать, как работает окружение?

О: Нет. Нужно понимать правила лексической области действия для переменных — мы о них уже рассказали. А еще вы знаете, что функция, возвращаемая из функции, несет с собой все свое окружение.

Помните, что функции JavaScript всегда выполняются в том же окружении, в котором они определяются. Чтобы внутри функции узнать, откуда появилась переменная, проведите поиск в ее внешних функциях, от наибольшего уровня вложенности к наименьшему.

Что такое «замыкание»?

Конечно, все вокруг говорят о замыканиях как об *обязательной* возможности языка, но сколько людей действительно понимают, что собой представляют замыкания и как ими пользоваться? Очень немного.

И тут возникает проблема: по мнению многих грамотных специалистов, *замыкания – весьма сложная тема.* Вот только у вас с ними проблем быть не должно. Хотите знать, почему? Нет, нет, вовсе не потому, что эта книга «учитывает особенности работы вашего мозга», и не потому, что мы подготовили суперприложение, которое достаточно построить для понимания замыканий. А потому что *вы их уже знаете.* Просто мы не называли их замыканиями.

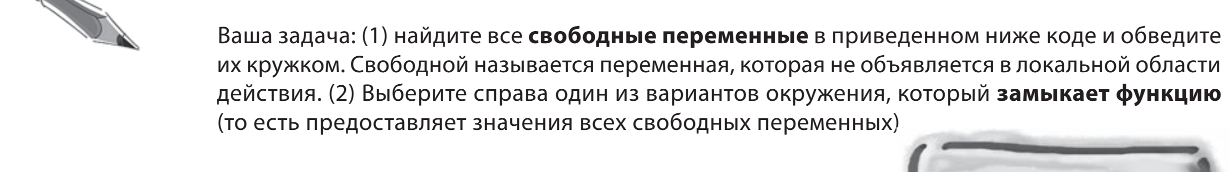
Впрочем, довольно разговоров — мы приведем формальное определение.

> **Замыкание (сущ.)**: функция вместе с сопутствующим окружением.

Если вы внимательно читали эту книгу, то должны сейчас воскликнуть: «Ух ты! Это, как говорится, "Почувствуйте разницу"».

Ладно, мы согласны, что это определение выглядит не слишком вразумительно. И почему замыкания называются *замыканиями*? Давайте разберем этот момент, потому что, честное слово, он может стать одним из решающих вопросов на собеседовании при приеме на работу или будущем повышении.

Чтобы понять смысл термина *замыкание*, необходимо понять концепцию «замкнутости» функции.



Ваша задача: (1) найдите все **свободные переменные** в приведенном ниже коде и обведите их кружком. Свободной называется переменная, которая не объявляется в локальной области действия. (2) Выберите справа один из вариантов окружения, который **замыкает функцию** (то есть предоставляет значения всех свободных переменных)

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
       ending = " -- I'm just sayin!";
    } else if (notSoMuch) {
       ending = " -- Not so much.";
    }
    alert(phrase + ending);
}
```

Обведите кружком свободные переменные. Свободные переменные не определяются в локальной области действия.

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

```
beingFunny = true;
justSayin = false;
oocoder = true;
```

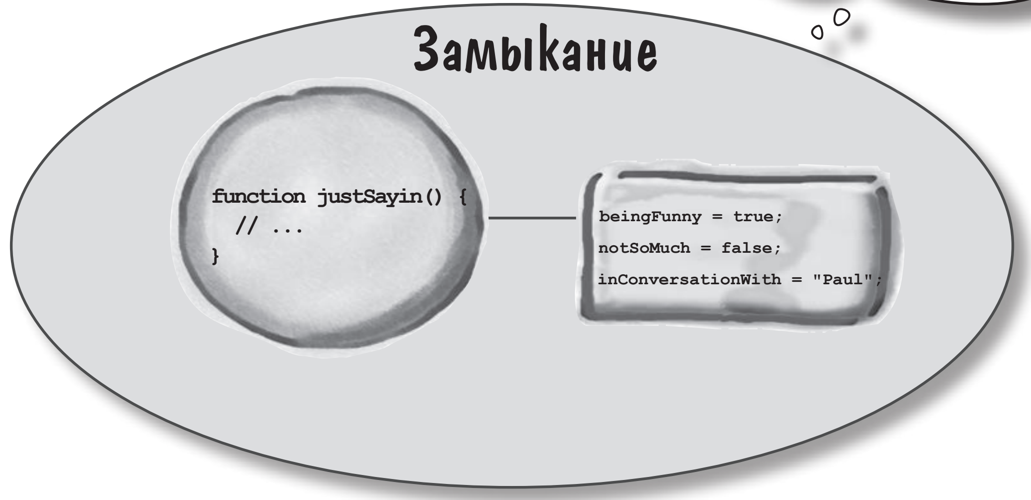
```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

Выберите один из вариантов окружения, который замыкает функцию.

Как замкнуть функцию

Вероятно, вы это уже поняли из предыдущего упражнения, но давайте повторим еще раз: в коде тела функции обычно присутствуют *локальные переменные* (включая все параметры функции), а также могут присутствовать переменные, не определяемые локально, — такие переменные называются *свободными*. Термин «*свободный*» появился потому, что в теле функции свободные переменные не связываются ни с какими значениями (иначе говоря, они объявляются где-то за пределами функции). Теперь в окружении, содержащем значение для каждой свободной переменной, функция становится *замкнутой*. А если взять функцию вместе с ее окружением, получится *замыкание*.

Если переменная в теле моей функции не определяется локально и не является глобальной, значит, она получена из функции, в которую вложена моя функция, и доступна в моем окружении.



Замыкание возникает тогда, когда функция, содержащая свободные переменные, объединяется с окружением, которое предоставляет привязки для всех свободных переменных.

Уже страниц десять на эту тему...
Мы когда-нибудь вернемся к реальному
программированию на JavaScript?
Или так и останемся в стране теорий?
Зачем мне знать, как работают все эти
низкоуровневые особенности функций?
Мне ведь просто нужно писать функции
и вызывать их, верно?

Мы бы согласились, не будь замыкания невероятно полезными. Жаль, что вам приходится напрягаться с изучением замыканий, но уверяем вас — они того стоят. Видите ли, замыкания — это не просто теоретическая конструкция функциональных языков, а весьма мощный инструмент реального программирования. Теперь, когда вы знаете, как работают замыкания (а мы не шутили, говоря, что понимание замыканий повысит вашу репутацию у начальства и коллег), пора научиться применять замыкания на практике.

И еще одно: замыкания сплошь и рядом применяются в программировании. Скоро они станут вашей второй натурой, и вы сами начнете широко использовать их в своем коде. Как бы то ни было, давайте рассмотрим пример кода с замыканиями, и вы поймете, о чем идет речь.

Использование замыканий для реализации счетчика

Как бы вы реализовали функцию-счетчик? Обычно это делается так:

```
var count = 0;          ← Глобальная переменная count.

function counter() {    ← При каждом вызове функция
    count = count + 1;     counter увеличивает глобаль-
    return count;          ную переменную count
}                          и возвращает новое значение.
```

Пример использования функции-счетчика:

```
console.log(counter());  ← Последовательно уве-
console.log(counter());    личиваем и выводим
console.log(counter());    значение счетчика.
```

Консоль JavaScript

```
1
2
3
```

Единственный недостаток такого решения заключается в том, что для `count` используется глобальная переменная, а это может создать проблемы при разработке в составе группы (программисты часто используют одинаковые имена, что приводит к конфликтам).

Нельзя ли реализовать счетчик с полностью локальной и защищенной переменной `count`? В этом случае она заведомо не будет конфликтовать с другими переменными, а увеличить ее можно будет только вызовом функции.

В реализации счетчика на базе замыкания можно воспользоваться большей частью уже приведенного кода. Смотрите и удивляйтесь:

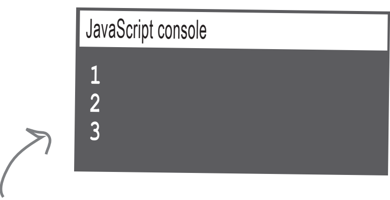
```
function makeCounter() {
    var count = 0;          ← Переменная count помещается в функцию
                              makeCounter. Теперь count становится
                              локальной, а не глобальной переменной.

    function counter() {    ← Создаем функцию counter, которая
        count = count + 1;    увеличивает переменную count.
        return count;
    }
    return counter;         ← И возвращаем функцию counter.
}
      ↑ Это замыкание: значение count
        сохраняется в его окружении.
```

Думаете, этот фокус сработает? Давайте попробуем...

Тест-драйв волшебного счетчика

Мы добавили небольшой фрагмент тестового кода для тестирования счетчика. Проверим, как он работает!

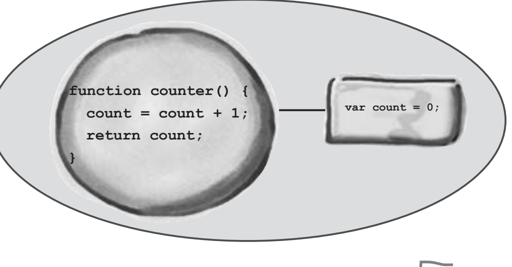
```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
var doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```



Счетчик работает как и положено...

Взгляд за кулисы

А теперь проанализируем код шаг за шагом, чтобы понять, как работает счетчик.



1. Мы вызываем функцию makeCounter, которая создает функцию counter и возвращает ее вместе с окружением, содержащим свободную переменную count. Другими словами, она создает замыкание. Функция, возвращенная из makeCounter, сохраняется в doCount.

2. Вызываем функцию doCount. Это приводит к выполнению тела функции counter.

3. Обнаружив переменную count, мы ищем ее в окружении и получаем ее значение. Переменная count увеличивается, новое значение сохраняется в окружении, после чего функция возвращает это новое значение в точку вызова doCount.

4. Шаги 2 и 3 повторяются при каждом вызове doCount.

```
function makeCounter() {
    var count = 0;

    function counter() {
③       count = count + 1;
        return count;
    }
    return counter;
}
① var doCount = makeCounter();
② console.log(doCount());
   console.log(doCount());
④ console.log(doCount());
```

Это замыкание.

При вызове makeCounter мы получаем замыкание: функцию с окружением.

При вызове doCount (переменной, содержащей ссылку на функцию counter) значение count берется из переменной count, находящейся в окружении замыкания. Для внешнего мира (то есть глобальной области действия) переменная count остается невидимой. Тем не менее мы можем использовать ее при каждом вызове doCount. К count невозможно обратиться иначе, кроме как при вызове doCount.

Ваша очередь. Попробуйте создать замыкания, описанные ниже. Мы понимаем, что эта задача не из простых, поэтому при необходимости можете подсматривать в ответы. Важно, чтобы вы самостоятельно поработали над примерами и в какой-то момент начали полностью понимать их.

Первая задача (10 очков): функция makePassword получает пароль в аргументе и возвращает функцию, которая принимает введенную строку и возвращает true, если введенная строка совпадает с паролем (иногда, чтобы понять суть замыкания, приходится перечитывать описание несколько раз):

```
function makePassword(password) {
    return ________________________ {
        return (passwordGuess === password);
    };
}
```

Следующая задача (20 очков): функция multN получает число (назовем его n) и возвращает функцию. Эта функция также получает число, умножает его на n и возвращает результат.

```
function multN(n) {
    return ________________________ {
        return ____________;
    };
}
```

Последняя задача (30 очков): разновидность счетчика, созданного ранее в этой главе. Функция makeCounter не получает аргументов, но определяет переменную count. Затем она создает и возвращает объект с единственным методом increment. Этот метод увеличивает переменную count и возвращает ее.

Создание замыкания с передачей функционального выражения в аргументе

Возвращение функции из функции — не единственный способ создания замыканий. Они создаются везде, где появляется ссылка на функцию со свободными переменными, и эта функция выполняется вне контекста, в котором она была создана.

Замыкания также могут создаваться передачей функций при вызове функций. Передаваемая функция выполняется в контексте, отличном от контекста ее создания. Пример:

```
function makeTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

}

makeTimer("Cookies are done!", 1000);
```

Имеется функция...

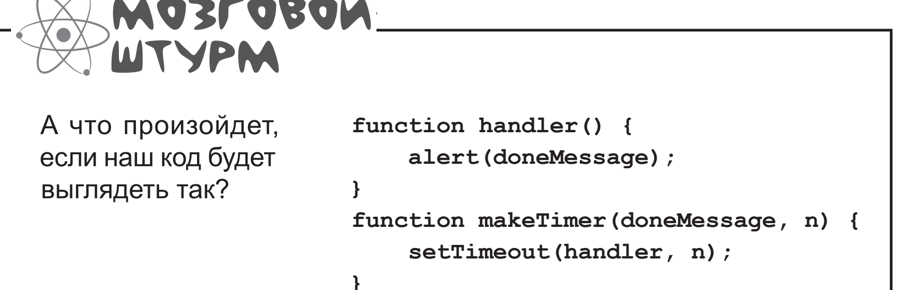
...со свободной переменной...

...которая используется как обработчик при вызове setTimeout.

...эта функция будет выполнена через 1000 миллисекунд, когда функция makeTimer давно завершится.

Функциональное выражение, содержащее свободную переменную `doneMessage`, передается функции `setTimeout`. Как вам известно, при этом функциональное выражение обрабатывается для получения ссылки на функцию, которая затем передается `setTimeout`. Метод `setTimeout` сохраняет функцию (а вернее, функцию вместе с окружением — иначе говоря, замыкание), после чего через 1000 миллисекунд вызывает ее.

И снова функция, передаваемая `setTimeout`, представляет собой замыкание, потому что вместе с ней передается окружение, связывающее свободную переменную `doneMessage` со строкой “Cookies are done!”.



Мозговой штурм

А что произойдет, если наш код будет выглядеть так?

```
function handler() {
    alert(doneMessage);
}
function makeTimer(doneMessage, n) {
    setTimeout(handler, n);
}
makeTimer("Cookies are done!", 1000);
```

Вернитесь к коду на с. 434 главы 9. Сможете ли вы изменить код, чтобы в нем использовалось замыкание, и обойтись без передачи третьего аргумента setTimeout?

Замыкание содержит непосредственное окружение, а не его копию

Многие разработчики при изучении замыканий ошибочно полагают, что окружение в замыкании должно содержать копию всех переменных и их значений. Это не так. На самом деле окружение обращается к «живым» переменным, используемым в вашем коде, так что если значение будет изменено за пределами функции замыкания, то последняя «увидит» это новое значение во время выполнения.

Давайте изменим наш пример и посмотрим, что это означает.

```
function setTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "OUCH!";
}
setTimer("Cookies are done!", 1000);
```

Здесь создается замыкание.

Значение doneMessage изменяется после вызова setTimeout.

1. Когда мы вызываем функцию setTimeout и передаем ей функциональное выражение, создается замыкание, которое содержит функцию вместе со ссылкой на окружение.

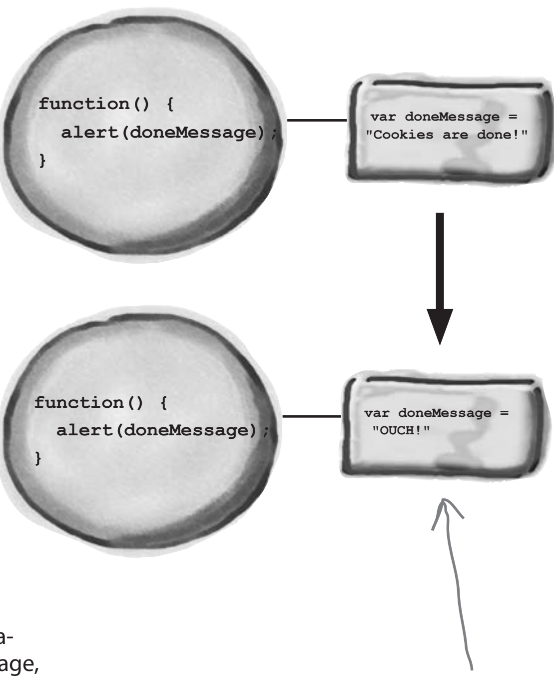
```
setTimeout(function() {
    alert(doneMessage);
}, n);
```

2. Затем при изменении значения doneMessage на "OUCH!" за пределами замыкания оно также изменяется в окружении, используемом замыканием.

```
doneMessage = "OUCH!";
```

3. Через 1000 миллисекунд вызывается функция в замыкании. Эта функция обращается к переменной doneMessage, которая теперь содержат строку "OUCH!" в окружении, поэтому в сообщении будет выведен текст "OUCH!".

```
function() { alert(doneMessage); }
```



При вызове функция использует значение doneMessage, хранящееся в окружении, — то есть новое значение, присвоенное переменной ранее в setTimer.

Создание замыкания в обработчике события

Рассмотрим еще один способ создания замыканий. Мы создадим замыкание в обработчике события — кстати, этот способ довольно часто встречается в коде JavaScript. Для начала мы создадим простую веб-страницу с кнопкой и элементом `<div>` для вывода сообщения. Программа будет отслеживать количество нажатий кнопки, а результат будет отображаться в `<div>`.

Ниже приведена разметка HTML и небольшой фрагмент CSS для создания страницы. Сохраните приведенную разметку HTML и CSS в файле divClosure.html.

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    // Код JavaScript
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```

Типичная, простая веб-страница...

Немного CSS для стилевого оформления элементов страницы.

Здесь будет размещаться наш код.

Кнопка и элемент <div> для сообщения, которое будет обновляться при каждом нажатии кнопки.

Наша цель: при каждом нажатии кнопки сообщение в <div> обновляется и в нем выводится количество нажатий.

Click me!
localhost/~Beth/HFJSe/chapter...
Click me!
You clicked me 10 times!

Перейдем к написанию кода. Вообще-то код для этого примера можно написать и без замыканий, но, как вы убедитесь, с замыканием код становится более компактным и даже чуть более эффективным.

Программа без замыкания

Сначала посмотрим, как бы вы реализовали этот пример *без* замыкания.

```
var count = 0;

window.onload = function() {
    var button = document.getElementById("clickme");
    button.onclick = handleClick;
};

function handleClick() {
    var message = "You clicked me ";
    var div = document.getElementById("message");
    count++;
    div.innerHTML = message + count + " times!";
}
```

Переменная count должна быть глобальной, потому что если сделать ее локальной для handleClick (обработчик события щелчка на кнопке — см. ниже), эта переменная будет повторно инициализироваться при каждом нажатии кнопки.

В функции обработчика события load мы получаем элемент кнопки и добавляем обработчик щелчка в свойство onclick.

Функция-обработчик события щелчка на кнопке.

Мы определяем переменную message...

...получаем элемент <div> из страницы...

...увеличиваем счетчик...

...и обновляем <div> сообщением с количеством нажатий.

Программа с замыканием

Версия без замыкания выглядит вполне разумно, кроме глобальной переменной, с которой могут возникнуть проблемы. Давайте перепишем код с использованием замыкания и сравним его с исходным вариантом. Сейчас мы только приведем сам код, а потом проанализируем его после тестирования.

```
window.onload = function() {
    var count = 0;
    var message = "You clicked me ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " times!";
    };
};
```

Теперь все переменные локальны по отношению к window.onload. Никаких проблем с конфликтами имен.

Обработчик назначается как функциональное выражение, присваиваемое свойству onclick кнопки, поэтому мы можем обращаться к div, message и count в функции. (Помните про лексическую область действия!)

Функция содержит три свободные переменные: div, message и count, поэтому для функции-обработчика click создается замыкание. Таким образом, свойству onclick кнопки назначается замыкание.

Тест-драйв счетчика нажатий

Итак, соберите HTML и код в файле divClosure.html и проверьте, как он работает. Загрузите страницу и щелкните на кнопке, чтобы увеличить счетчик. Сообщение в элементе <div> должно обновиться. Снова просмотрите код и убедитесь в том, что вы понимаете, как он работает. После этого переверните страницу, и мы разберем его вместе.

Click me!

You clicked me 10 times!

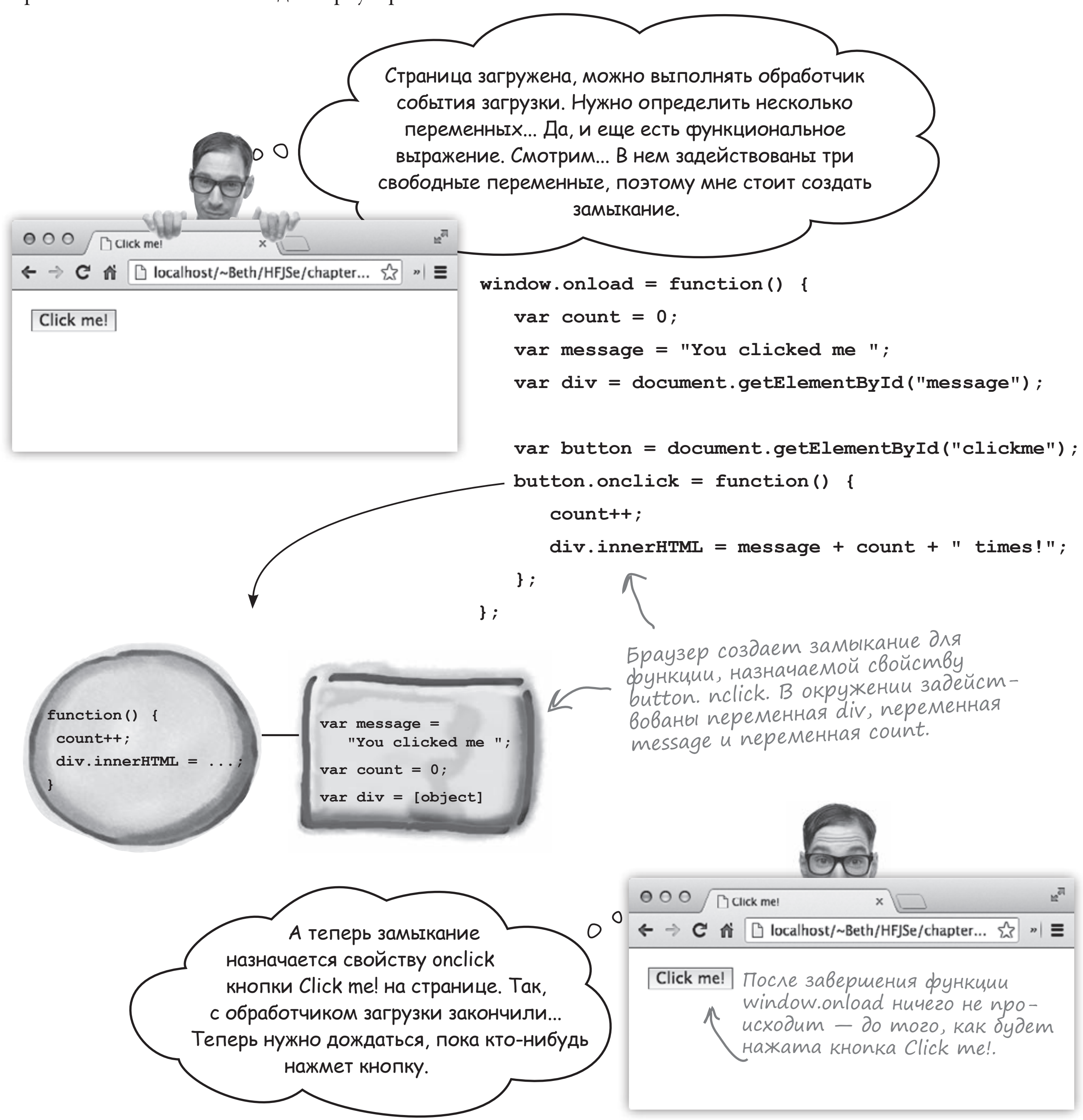
Так выглядит страница.

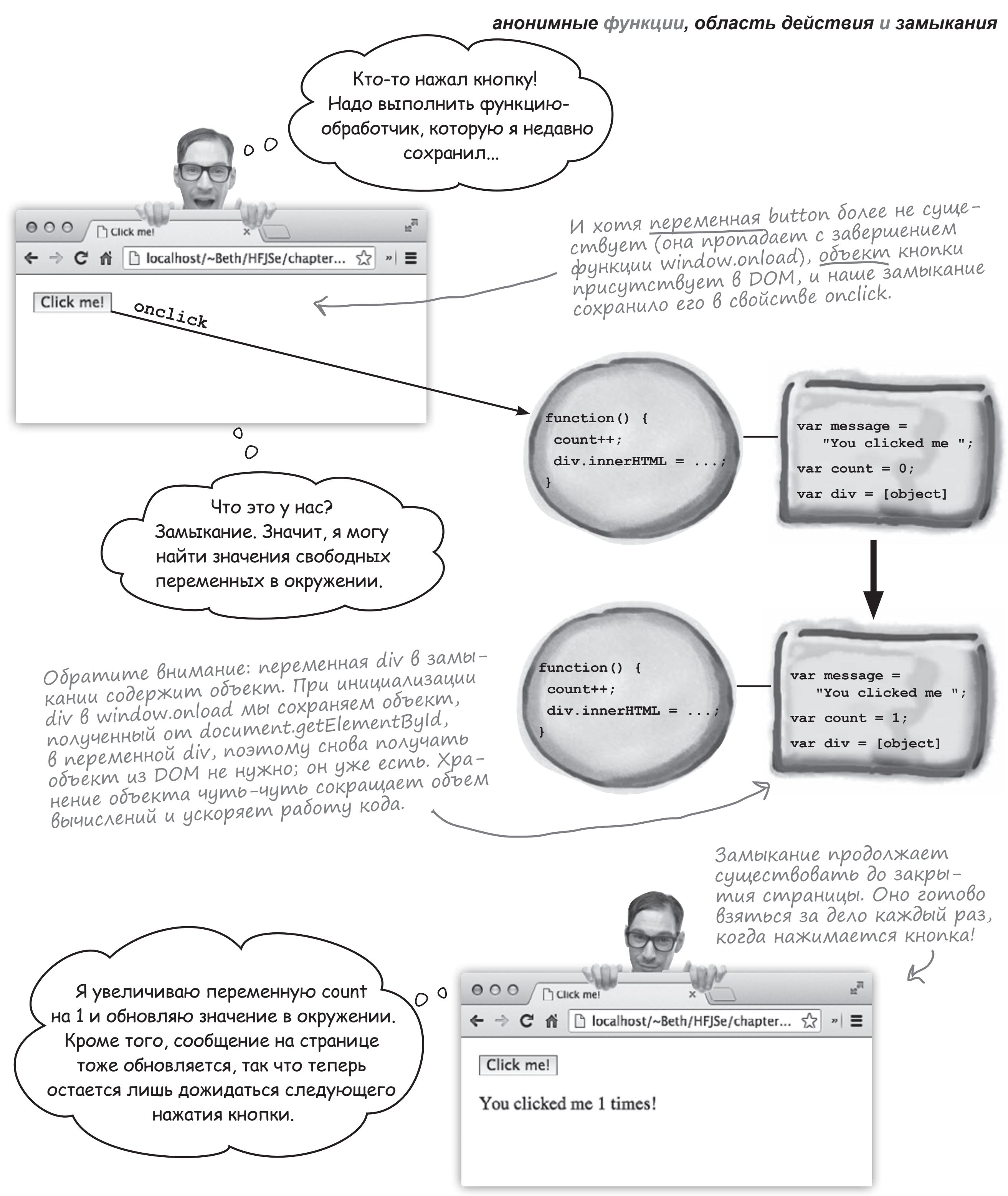
```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    window.onload = function() {
        var count = 0;
        var message = "You clicked me ";
        var div = document.getElementById("message");

        var button = document.getElementById("clickme");
        button.onclick = function() {
            count++;
            div.innerHTML = message + count + " times!";
        };
    };
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```

Обновите свой файл divClosure.html.

Как работает замыкание

Чтобы понять, как работает замыкание, мы снова проследим за тем, что происходит при выполнении этого кода в браузере...




Кто-то нажал кнопку! Надо выполнить функцию-обработчик, которую я недавно сохранил...
Click me!
localhost/~Beth/HFJSe/chapter...
onclick
И хотя переменная button более не существует (она пропадает с завершением функции window.onload), объект кнопки присутствует в DOM, и наше замыкание сохранило его в свойстве onclick.
function() {
count++;
div.innerHTML = ...;
}
var message = "You clicked me ";
var count = 0;
var div = [object]
Что это у нас? Замыкание. Значит, я могу найти значения свободных переменных в окружении.
Обратите внимание: переменная div в замыкании содержит объект. При инициализации div в window.onload мы сохраняем объект, полученный от document.getElementById, в переменной div, поэтому снова получать объект из DOM не нужно; он уже есть. Хранение объекта чуть-чуть сокращает объем вычислений и ускоряет работу кода.
function() {
count++;
div.innerHTML = ...;
}
var message = "You clicked me ";
var count = 1;
var div = [object]
Замыкание продолжает существовать до закрытия страницы. Оно готово взяться за дело каждый раз, когда нажимается кнопка!
Я увеличиваю переменную count на 1 и обновляю значение в окружении. Кроме того, сообщение на странице тоже обновляется, так что теперь остается лишь дожидаться следующего нажатия кнопки.
Click me!
localhost/~Beth/HFJSe/chapter...
Click me!
You clicked me 1 times!

ДЛЯ НАСТОЯЩИХ ПРОФЕССИОНАЛОВ

ВТОРАЯ ПОПЫТКА

Нам нужен эксперт по первоклассным функциям — и нам порекомендовали вас! Теперь, когда вы знаете, как работают замыкания, сможете ли вы объяснить, почему оба фрагмента дают результат 008? Запишите все переменные, сохраняемые в окружении, для приведенных ниже функций. Учтите, что окружение вполне может быть пустым. Сверьтесь с ответом в конце главы.

Образец #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

Окружение

Образец #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Окружение

Возьми в руку карандаш

Просмотрите следующий фрагмент кода:

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})("cookies");
```

Использование функционального выражения вместо ссылки, доведенное до крайности.

Ваша задача — не только определить, что делает этот код, но и *как* он это делает. Для этого проще пойти в обратном направлении, то есть возьмите анонимную функцию, присвойте ее переменной, а затем используйте эту переменную повсюду, где раньше было функциональное выражение. Станет ли смысл кода более понятным? И что же делает этот код?

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Анонимная функция** представляет собой функциональное выражение без имени.
- Анонимные функции могут сделать код более элегантным.
- **Объявление функции** определяется до обработки остального кода.
- **Функциональное выражение** обрабатывается во время выполнения вместе с остальным кодом, поэтому функциональные выражения остаются неопределенными до выполнения команд, в которых они находятся.
- Функциональное выражение можно передать другой функции или вернуть его из функции.
- Результатом обработки функционального выражения является **ссылка на функцию**, поэтому функциональные выражения могут использоваться всюду, где могут использоваться ссылки на функции.
- **Вложенные функции** определяются внутри других функций.
- Вложенная функция имеет локальную область видимости, как и другие переменные.
- **Лексическая область действия** означает, что для определения области действия переменных достаточно прочитать код.
- Чтобы получить значение переменной во вложенной функции, используйте значение, определяемое в ближайшей внешней функции. Если найти значение не удалось, обращайтесь к глобальной области действия.
- **Замыкание** состоит из функции и используемого ею окружения.
- В замыкании отражены значения переменных, находившихся в области действия на момент создания замыкания.
- **Свободными переменными** называются переменные в теле функции, не связанные с данной функцией.
- При выполнении замыкания в контексте, отличном от контекста его создания, значения свободных переменных определяются окружением.
- Замыкания часто используются для сохранения состояния в обработчиках событий.

Давайте убедимся в том, что вы правильно поняли синтаксис передачи анонимных функциональных выражений другим функциям. Преобразуйте следующий код, чтобы вместо переменной (в данном случае vaccine) в нем использовалось анонимное функциональное выражение.

Ниже приведено наше решение:

```
administer(patient, function(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}, time);
```

Обратите внимание: ничто не мешает использовать многострочные функциональные выражения в аргументах. Будьте внимательны с синтаксисом; ошибиться проще простого!

Упражнение
Решение

Ваша очередь. Попробуйте создать замыкания, описанные ниже. Мы понимаем, что эта задача не из простых, поэтому при необходимости можете подсматривать в ответы. Важно, чтобы вы самостоятельно поработали над примерами и в какой-то момент начали полностью понимать их.

Наши решения:

Первая задача (10 очков): функция makePassword получает пароль в аргументе и возвращает функцию, которая принимает введенную строку и возвращает true, если введенная строка совпадает с паролем (иногда, чтобы понять суть замыкания, приходится перечитывать описание несколько раз):

```
makePassword(password) {
    return function guess(passwordGuess) {
        return (passwordGuess === password);
    };
}
var tryGuess = makePassword("secret");
console.log("Guessing 'nope': " + tryGuess("nope"));
console.log("Guessing 'secret': " + tryGuess("secret"));
```

Функция, возвращаемая из makePassword, представляет собой замыкание с окружением, содержащим свободную переменную password.

Мы передаем makePassword значение “secret”, которое сохраняется в окружении замыкания.

При вызове tryGuess переданное слово (“nope” или “secret”) сравнивается со значением password в окружении tryGuess.

Обратите внимание: используется именованное функциональное выражение. Это не обязательно, но удобно для ссылок на имя внутренней функции. Возвращаемая функция _должна_ вызываться tryGuess (а не guess).

См. продолжение на следующей странице…

Давайте убедимся в том, что вы правильно поняли синтаксис передачи анонимных функциональных выражений другим функциям. Преобразуйте следующий код, чтобы вместо переменной (в данном случае vaccine) в нем использовалось анонимное функциональное выражение.

Ниже приведено наше решение (продолжение):

Следующая задача (20 очков): функция multN получает число (назовем его n) и возвращает функцию. Эта функция также получает число, умножает его на n и возвращает результат.

```
function multN(n) {
    return function multBy(m) {
        return n*m;
    };
}
var multBy3 = multN(3);
console.log("Multiplying 2: " + multBy3(2));
console.log("Multiplying 3: " + multBy3(3));
```

Функция, возвращаемая из multN, представляет собой замыкание с окружением, содержащим свободную переменную n.

Итак, вызов multN(3) возвращает функцию, которая умножает любое переданное число на 3.

Последняя задача (30 очков): разновидность счетчика, созданного ранее в этой главе. Функция makeCounter не получает аргументов, но определяет переменную count. Затем она создает и возвращает объект с единственным методом increment. Этот метод увеличивает переменную count и возвращает ее.

```
function makeCounter() {
    var count = 0;
    return {
        increment: function() {
            count++;
            return count;
        }
    };
}
var counter = makeCounter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

Функция почти не отличается от предыдущей функции makeCounter — не считая того, что метод increment возвращает объект, а не функцию.

Метод increment использует свободную переменную count. Итак, increment — замыкание с окружением, содержащим переменную count.

Теперь мы вызываем makeCounter и получаем объект с методом (который представляет собой замыкание).

Метод вызывается как обычно; при этом метод обращается к переменной count в своем окружении.

Возьми в руку карандаш. Решение

Используя свои знания в области функций и переменных, пометьте истинные утверждения. Наше решение:

- ☑ В переменной handler хранится ссылка на функцию.
- ☑ При назначении обработчика свойству window.onload присваивается ссылка на функцию.
- ☑ Переменная handler существует только для того, чтобы быть назначенной свойству window.onload.
- ☑ Переменная handler больше нигде не используется — этот код должен выполняться только при исходной загрузке страницы.
- ☑ Повторный вызов обработчиков onload нежелателен — он может создать проблемы, поскольку обработчики обычно выполняют операции, относящиеся к инициализации всей страницы.
- ☑ Функциональные выражения создают ссылки на функции.
- ☑ А мы упоминали о том, что при назначении обработчика window.onload присваивается ссылка на функцию?

Возьми в руку карандаш. Решение

Ваша задача: (1) найдите все **свободные переменные** в приведенном ниже коде и обведите их кружком. "Свободной" называется переменная, которая не объявляется в локальной области действия. (2) Выберите справа один из вариантов окружения, который **замыкает функцию** (то есть предоставляет значения всех свободных переменных). Ниже приведено наше решение.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
       ending = " -- I'm just sayin!";
    } else if (notSoMuch) {
       ending = " -- Not so much.";
    }
    alert(phrase + ending);
}
```

Обведите кружком свободные переменные в этом коде. Свободные переменные не определяются в локальной области действия.

Возьми в руку карандаш
Решение

В приведенном ниже фрагменте есть несколько мест, в которых уместно воспользоваться анонимными функциями. Переработайте этот код и внесите необходимые исправления. Там, где потребуется, вычеркните старый код и запишите новые фрагменты. Да, и еще одна задача: обведите кружком все анонимные функции, уже используемые в этом коде. Ниже приводится наше решение.

```
window.onload = init;
var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
        console.log("Baking the cookies.");
        setTimeout(done, time);
    }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

Мы переработали код и создали два анонимных функциональных выражения — для функции init и для функции handleButton.

Теперь функциональное выражение назначается свойству window.onload...

...а другое выражение назначается свойству button.onclick.

```
window.onload = function() {
    var button = document.getElementById("bake");
    button.onclick = function() {
        console.log("Time to bake the cookies.");
        cookies.bake(2500);
    };
};

var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
            console.log("Baking the cookies.");
            setTimeout(done, time);
        }
};

function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

Получите дополнительный балл, если догадались напрямую передать функцию cool при вызове setTimeout:

```
setTimeout(function() {
    alert("Cookies are cool, time to eat!");
}, 1000);
```

Для настоящих профессионалов

Нам нужен эксперт по первоклассным функциям — и нам порекомендовали вас! Теперь, когда вы знаете, как работают замыкания, сможете ли вы объяснить, почему оба фрагмента дают результат 008? Запишите все переменные, сохраняемые в окружении, для приведенных ниже функций. Учтите, что окружение вполне может быть пустым. Ниже приведено наше решение.

Образец #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

Окружение

```
secret = "008"
```

secret — свободная переменная в getValue...

...поэтому она сохраняется в окружении getValue. Но мы не возвращаем getValue из getSecret, поэтому замыкание никогда не становится видимым за пределами контекста, в котором оно было создано.

Образец #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Окружение

```
secret = "008"
```

secret — свободная переменная в getValue...

...а здесь создается замыкание, возвращаемое из getSecret. Соответственно, при вызове getValueFun (getValue) в другом контексте (глобальной области действия) используется значение secret из окружения.

Возьми в руку карандаш
Решение

А вот и решение нашей головоломки!

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})("cookies");
```

Ваша задача — не только определить, что делает этот код, но и *как* он это делает. Для этого проще пойти в обратном направлении — то есть возьмите анонимную функцию, присвойте ее переменной, а затем используйте эту переменную повсюду, где раньше было функциональное выражение. Станет ли смысл кода более понятным? И что же делает этот код?

Выделяем функцию из приведенного кода — здесь мы назвали ее eat. При желании также можно было использовать объявление функции.

```
var eat = function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
};
(eat)("cookies");
```

Конечно, запись eat (“cookies”) выглядит привычнее, но здесь мы показываем, как подставить eat в приведенное выше функциональное выражение.

Итак, этот код всего лишь определяет встроенное функциональное выражение, после чего немедленно вызывает его с аргументами.

По сути здесь функция eat вызывается для значения “cookies”. Но откуда взялись лишние круглые скобки?

Помните, что объявление функции начинается с ключевого слова function, за которым следует имя? И что функциональное выражение должно находиться в команде? Если не заключить функциональное выражение в круглые скобки, интерпретатор JavaScript будет рассматривать его как объявление, а не как функциональное выражение. Но для вызова eat круглые скобки не нужны, поэтому их можно убрать.

Да, и он возвращает строку “More please”.

12 нетривиальное создание объектов

Создание объектов

До настоящего момента мы создавали объекты вручную. Для каждого объекта использовался **объектный литерал,** который задавал все без исключения свойства. Для небольших программ это допустимо, но для серьезного кода потребуется что-то получше, а именно **конструкторы объектов**. Конструкторы упрощают создание объектов, причем вы можете создавать объекты по единому **шаблону** — иначе говоря, конструкторы позволяют создавать серии объектов, обладающих одинаковыми свойствами и содержащих одинаковые методы. Код, написанный с использованием конструкторов, получается гораздо более **компактным** и снижает риск ошибок при создании большого количества объектов. Уверяем, что после изучения этой главы вы будете пользоваться конструкторами так, словно занимались этим всю сознательную жизнь.

Создание объектов с использованием объектных литералов

Пока в этой книге для создания объектов использовались *объектные литералы*. По сути, объектный литерал представляет собой полное описание объекта:

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,

    start: function() { this.started = true;},
    stop: function() { this.started = false;},
    drive: function {
         // Код управления машиной
    }
};
```

При использовании объектного литерала все части объекта записываются в фигурных скобках. Результат представляет собой объект JavaScript, который обычно присваивается переменной для использования в будущем.

Объектные литералы позволяют легко создавать объекты в любой точке кода. Но представьте, что вам потребовалось создать действительно много объектов — скажем, объекты, представляющие такси для целого автопарка. Захочется ли вам вводить сотни разных объектных литералов?

МОЗГОВОЙ ШТУРМ

Представьте, что вам пришлось создать объекты такси для целого автопарка. Какие проблемы могут возникнуть из-за использования объектных литералов?

- [ ] Пальцы устанут набирать такой код!
- [ ] Можно ли гарантировать, что все объекты имеют одинаковые свойства? А если сделать ошибку, опечатку, пропустить свойство?
- [ ] Много объектных литералов — много кода. Разве это не замедлит загрузку страницы в браузере?
- [ ] Код методов start, stop и drive придется повторять снова и снова.
- [ ] А если вы решите добавить или удалить свойство (или изменить процедуру запуска или остановки двигателя)? Придется вносить изменения во всех объектах.
- [ ] Кому нужны такси, если есть Uber?

О сходстве и различии между объектами

До настоящего момента мы создавали объекты *по соглашению*. Мы объединяли набор свойств и методов и говорили: «Это машина!» или «Это собака!», но сходство объектов машин (или собак) зависело от соблюдения установленных соглашений.

Этот способ неплох, но создает проблемы при большом количестве объектов или разработчиков, работающих над общим кодом и не знающих (не соблюдающих) соглашений.

Впрочем, не надо верить на слово. Взгляните на некоторые объекты, которые уже встречались нам ранее в этой книге. Тогда мы договорились, что эти объекты представляют машины:

Похоже на объекты машин, с которыми мы уже имели дело, — те же свойства и методы.

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,

    start: function() {
        this.started = true;
    },

    stop: function() {
        this.started = ...
```

Да, это похоже на другие объекты машин... Постойте. Реактивный двигатель? Хм, не знаю, можно ли назвать это машиной.

Превосходная машина, но в ней отсутствуют некоторые базовые свойства — пробег или цвет. Зато есть другие, лишние, это может создать проблемы...

```
var rocketCar = {
    make: "Galaxy",
    model: "4000",
    year: 2001,
    color: "white",
    passengers: 6,
    convertible: false,
    mileage: 60191919,
    started: false,

    start: function() {
        this.started = true;
    },

    stop: function() {
        this.started = false;
    },

    drive: function() {
        // Код управления машиной
    },

    thrust: function(amount) {
        // Код включения
реактивного двигателя
    }
};
```

```
var toyCar = {
    make: "Mattel",
    model: "PeeWee",
    color: "blue",
    type: "wind up",
    price: "2.99"
};
```

Может, это и машина, но она не похожа на другие машины. У нее есть модель, производитель, цвет... но ведь это игрушка. Что она здесь делает?

```
var tbird = {
    make: "Ford",
    model: "Thunderbird",
    year: 1957,
    passengers: 4,
    convertible: true,
    started: false,
    oilLevel: 1.0,

    start: function() {
        if (oilLevel > .75) {
            this.started = true;
        }
    },

    stop: function() {
        this.started = false;
    },

    drive: function() {
        // Код управления машиной
    }

};
```

Если бы только существовал механизм **создания объектов**, обладающих единой базовой структурой. Тогда все мои объекты выглядели бы одинаково: они имели бы одинаковые наборы свойств, а все методы определялись бы в одном месте... Как формочка для печенья, которая вместо печений нарезает копии объекта. Но это, конечно, только мечты.

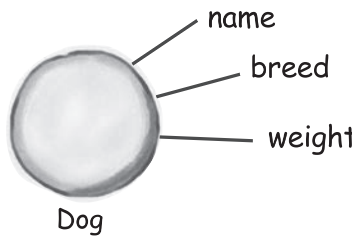
Конструкторы

Конструкторы объектов, или сокращенно «конструкторы», открывают путь к улучшенному созданию объектов. Конструктор можно представить себе как маленькую фабрику, способную создать бесконечное множество похожих объектов.

В программном коде конструктор напоминает функцию, возвращающую объект: он определяется один раз, а затем вызывается везде, где потребуется создать новый объект. Но как вы вскоре увидите, создание конструктора потребует чуть больших усилий.

Как проще всего увидеть работу конструктора? Конечно, создать его. Вернемся к нашему старому знакомому, объекту, представляющему собаку, и напишем конструктор для создания таких объектов. Ниже приведена версия объекта собаки, которую мы использовали ранее — с кличкой (name), породой (breed) и весом (weight).

Конструкторы объектов и функции тесно связаны друг с другом. Помните об этом, когда будете учиться создавать и использовать конструкторы.



Если мы захотим определить объект собаки в виде литерала, он будет выглядеть примерно так:

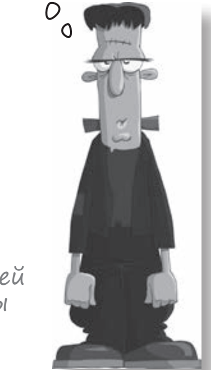
```
var dog = {
    name: "Fido",
    breed: "Mixed",
    weight: 38
};
```

Простой объект, создаваемый на основе литерала. Теперь нужно придумать, как создать целую свору таких объектов.

Однако нам нужна не просто конкретная собака с кличкой *Fido* — нам нужен механизм создания объекта *произвольной собаки* с заданной кличкой, породой и весом. Для этого мы снова напишем код, который внешне напоминает функцию, но с добавлением щепотки объектного синтаксиса.

Вероятно, после такого введения вас разбирает любопытство? Переверните страницу — там вы узнаете все о конструкторах и об их использовании.

Лично мне конструкторы напоминают моего создателя Франкенштейна. Они тоже строят нечто новое из составных частей функций и объектов. Разве не замечательно?



На следующей странице вы поймете, почему.

Как создать конструктор

Использование конструкторов проходит в два этапа: сначала мы создаем конструктор, а потом используем его для создания объектов. Начнем с создания конструктора.

Итак, нам нужен конструктор для создания объектов собак, а конкретнее — собак с заданной кличкой, породой и весом. Для этого мы определим функцию, которая умеет создавать нужные объекты. Выглядит она так:

Обратите внимание: имя конструктора начинается с прописной буквы. Это не обязательно, но разработчики обычно следуют этому правилу.

Конструктор выглядит как обычная функция.

Параметры функции соответствуют свойствам, которые должны передаваться при вызове для каждого создаваемого объекта.

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
```

Эта часть больше похожа на объект: значения параметров присваиваются переменным, которые похожи на свойства.

Здесь не используются локальные переменные, как во многих функциях. Вместо них используется ключевое слово this, которое ранее нам встречалось только внутри объектов.

Имена свойств и параметров не обязаны совпадать, но часто совпадают — это еще одно удобное соглашение.

Обратите внимание: конструктор ничего не возвращает.

Оставайтесь с нами: сейчас мы объясним, как использовать конструкторы, и все встанет на свои места.

Возьми в руку карандаш

Нам нужна ваша помощь. Мы использовали объектные литералы для создания объектов, представляющих уток. После всего, что сказано выше, сможете ли вы написать конструктор для создания таких объектов? Ниже приведен объектный литерал, который следует взять за основу для написания вашего конструктора:

```
var duck = {
    type: "redheaded",
    canFly: true
}
```

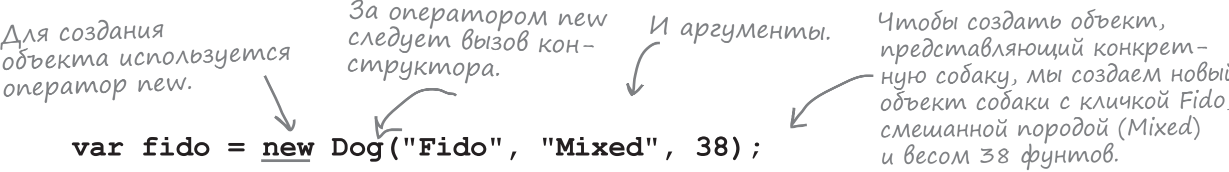
Пример объектного литерала.

Напишите конструктор для создания уток.

P. S. Мы знаем, что вы еще не до конца понимаете смысл происходящего, так что пока сосредоточимся на синтаксисе.

Как использовать конструктор

Мы сказали, что конструктор используется в два этапа: сначала мы создаем конструктор, а затем используем его. Ранее мы создали конструктор Dog, теперь воспользуемся им. Вот как это делается:



Итак, чтобы создать новый объект собаки с кличкой "Fido", породой "Mixed" и весом 38 фунтов, мы ставим ключевое слово `new`, за которым следует вызов конструктора с нужными аргументами. После обработки этой команды в переменной `fido` будет храниться ссылка на созданный объект.

Определив конструктор для объектов собак, мы можем создавать их и далее:

```
var fluffy = new Dog("Fluffy", "Poodle", 30);

var spot = new Dog("Spot", "Chihuahua", 10);
```

Гораздо удобнее, чем с литералами, верно? К тому же при таком создании объектов мы знаем, что все объекты собак содержат необходимый набор свойств: name, breed и weight.

Упражнение

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    var size = "small";
    if (dogs[i].weight > 10) {
        size = "large";
    }
    console.log("Dog: " + dogs[i].name
                + " is a " + size
                + " " + dogs[i].breed);
}
```

Маленькое практическое упражнение для закрепления новых знаний: включите этот код в страницу и попробуйте выполнить его. Запишите здесь полученный результат.

Как работают конструкторы

Вы уже знаете, как объявить конструктор и использовать его для создания объектов, но чтобы понять, что на самом деле происходит при вызове конструктора, необходимо заглянуть «за кулисы». Самое главное: чтобы понять, как работают конструкторы, нужно знать, что же делает оператор `new`.

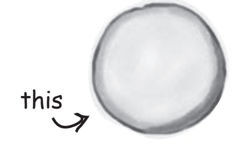
Начнем с команды, использованной для создания объекта в переменной `fido`:

```
var fido = new Dog("Fido", "Mixed", 38);
```

Посмотрите на правую часть команды присваивания, где, собственно, все и происходит. Проследим за ее выполнением шаг за шагом:

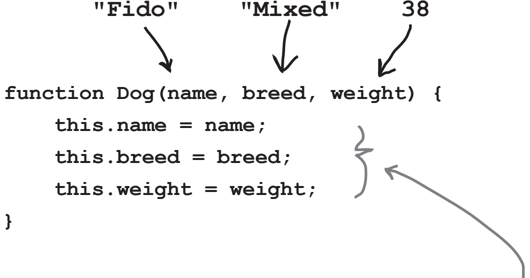
1. Сначала `new` создает новый, пустой объект:

2. Затем `new` заносит в `this` ссылку на новый объект.



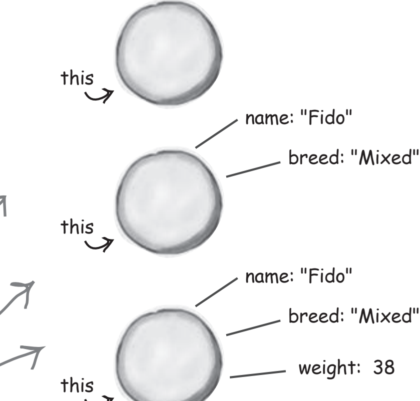
Вспомните главу 5: в this хранится ссылка на текущий объект, с которым связан выполняемый код.

3. После подготовки `this` вызывается функция Dog, которой передаются аргументы "Fido", "Mixed" и 38.



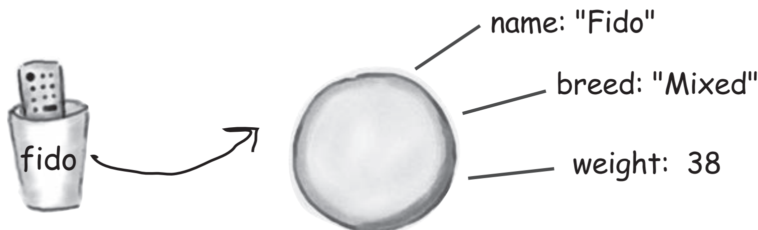

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
```

4. Затем вызывается тело функции. Как и большинство конструкторов, Dog задает значения свойств только что созданного объекта `this`.



При выполнении тела функции Dog новый объект заполняется тремя свойствами, которым присваиваются значения соответствующих параметров.

5. Наконец, после того как функция Dog будет выполнена, оператор `new` возвращает `this`, то есть ссылку на только что созданный объект. Обратите внимание: ссылка this возвращается автоматически, вам не придется явно возвращать ее в своем коде. И после того как новый объект будет возвращен, эта ссылка присваивается переменной `fido`.



СТАНЬ браузером

Ниже приведен код JavaScript, содержащий ошибки. Представьте себя на месте браузера и найдите ошибки в этом коде. А когда упражнение будет выполнено, загляните в конец главы и посмотрите, удалось ли вам отыскать все ошибки до единой. Кстати, мы уже дошли до главы 12, если хотите, можете делать замечания по стилю программирования. Вы заслужили это право.

```
function widget(partNo, size) {
    var this.no = partNo;
    var this.breed = size;
}
function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}

var widgetA = widget(100, "large");
var widgetB = new widget(101, "small");
var formFactorA = newFormFactor("plastic", widgetA);
var formFactorB = new ForumFactor("metal", widgetB);
```

В конструкторы также можно добавить методы

Объекты, созданные конструктором Dog, практически ничем не отличаются от объектов, создававшихся ранее... только они не содержат метод bark. Проблема легко решается, потому что наряду со свойствами в конструкторе могут задаваться методы объекта. Расширим код конструктора Dog и включим в него метод bark:

Как вы уже знаете, методы объекта также являются свойствами — просто таким свойствам назначаются функции.

Чтобы добавить в создаваемые объекты метод bark, мы просто назначаем функцию (в данном случае анонимную) свойству this.bark:

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
```

Отныне каждый создаваемый объект также содержит метод bark, который можно будет вызвать в програме.

Как и во всех остальных объектах, которые мы создавали в прошлом, объект, метод которого вызывается в программе, обозначается ключевым словом this.

Простой тест-драйв метода bark

Но довольно разговоров о конструкторах — включите приведенный выше код в страницу HTML, а затем добавьте следующий фрагмент для его тестирования:

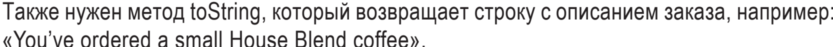
```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    dogs[i].bark();
}
```

Как видите, метод bark работает так, как задумано: выдаваемое сообщение зависит от конкретного объекта.

У нас есть конструктор для создания объектов, представляющих разные кофейные напитки, но в этом конструкторе нет методов.

Нам понадобится метод getSize, который возвращает строку в зависимости от объема напитка (в унциях):

- 8 унций — малый (small);
- 12 унций — средний (medium);
- 16 унций — большой (large).



Также нужен метод toString, который возвращает строку с описанием заказа, например: «You've ordered a small House Blend coffee».

Запишите свой код внизу, а затем протестируйте его в браузере. Попробуйте создать несколько напитков с разными объемами. Прежде чем двигаться дальше, сверьтесь с ответами.

```
function Coffee(roast, ounces) {
    this.roast = roast;
    this.ounces = ounces;









}

var houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());

var darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

Запишите здесь два метода, которые добавляются в конструктор.

Консоль JavaScript

```
You've ordered a medium House Blend coffee.
You've ordered a large Dark Roast coffee.
```

Наш результат; ваш должен выглядеть так же.

Часто Задаваемые Вопросы

В: Почему имена конструкторов начинаются с прописной буквы?

О: Это соглашение, которое используется разработчиками JavaScript, помогает отличить конструкторы от обычных функций. Почему? Потому что при вызове конструкторов должен использоваться оператор new. В общем случае такое выделение имени конструктора упрощает его идентификацию при чтении кода.

В: Если не считать задания свойств объекта, конструктор ничем не отличается от обычной функции?

О: С точки зрения программирования — ничем. С конструктором можно делать все то же, что и с обычной функцией: объявлять и использовать переменные, включать циклы for, вызывать другие функции и т. д. Единственное, чего не следует делать в конструкторе, так это возвращать значение (отличное от **this**), потому что в этом случае конструктор не вернет объект, который ему положено конструировать.

В: Имена параметров конструктора должны соответствовать именам свойств?

О: Нет. Вы можете назначить параметрам любые имена на свое усмотрение. Параметры всего лишь используются для хранения значений, которые назначаются свойствам объекта. По-настоящему важны только имена свойств объекта. Впрочем, для наглядности часто используются одинаковые имена, чтобы по определению конструктора сразу было видно, какие свойства он задает.

В: Объект, созданный конструктором, ничем не отличается от объекта, созданного на базе литерала?

О: Да, по крайней мере пока мы не займемся более сложными объектами. Но это будет только в следующей главе.

В: Почему для создания объектов необходим оператор new? Почему нельзя создать объект обычной функцией и вернуть его (так, как это делалось в makeCar из главы 5)?

О: Да, объекты можно создавать и так, но, как мы упоминали ранее, при использовании new выполняются некоторые дополнительные операции. Эта тема более подробно рассматривается позднее в этой главе, а также в главе 13.

В: Я еще не до конца понимаю смысл `this` в конструкторе. Мы используем `this` при назначении свойств объекта, а также включаем `this` в методы объекта. Это одно и то же?

О: Когда мы вызываем конструктор (для создания объекта), ссылка **this** связывается с только что созданным объектом, чтобы весь код, выполняемый в конструкторе, мог применяться к этому конкретному объекту.

Позднее при вызове метода ссылка this указывает на объект, метод которого был вызван. Таким образом, this в ваших методах всегда обозначает объект, метод которого был вызван.

В: Создавать объекты конструктором лучше, чем создавать их с использованием объектных литералов?

О: Каждый вариант по-своему полезен. Конструктор удобен при создании множества объектов, содержащих одноименные методы и свойства; он обеспечивает возможность повторного использования кода и обеспечивает единую структуру объектов.

Но иногда требуется создать объект «на скорую руку», который, возможно, будет использоваться всего один раз. Компактность и выразительность литералов хорошо подходят для этой задачи.

Итак, на самом деле все зависит от ваших потребностей. Каждый способ пригодится в подходящей ситуации.

Вскоре мы рассмотрим хороший пример.

ОПАСНАЯ ЗОНА

У конструкторов есть одна особенность, к которой нужно относиться очень внимательно: не забывайте про ключевое слово **new**. О нем часто забывают, в конце концов, конструктор — всего лишь функция, и его можно вызывать без **new**. *Но если вы забудете использовать **new** с конструктором, в коде появятся коварные ошибки, которые трудно обнаружить.* Давайте посмотрим, что произойдет, если забыть ключевое слово **new**...

```
function Album(title, artist, year) {
    this.title = title;
    this.artist = artist;
    this.year = year;
    this.play = function() {
        // ...
    };
}
var darkside = Album("Dark Side of the Cheese","Pink Mouse", 1971);
darkside.play();
```

Вроде бы нормальный конструктор.

Ой — забыли использовать new!

С другой стороны, Album — это функция... Может, ничего страшного?

А что произойдет, если все равно вызвать метод play? Ой, ничего хорошего...

```
Uncaught TypeError: Cannot call method 'play' of undefined
```

ТЕХНИКА БЕЗОПАСНОСТИ

Итак, почему же это могло произойти?

Помните, что **new** создает новый объект, прежде чем присвоить его **this** (а затем вызывает конструктор). Без оператора **new** новый объект создан не будет.

- Это означает, что любые ссылки **this** в конструкторе будут относиться не к новому объекту альбома, а к глобальному объекту вашего приложения.
- Без использования **new** не существует объекта, возвращаемого из конструктора; это означает, что переменной darkside не будет присвоен объект и она останется неопределенной. Из-за этого при вызове play выдается ошибка с сообщением о том, что объект, для которого мы пытаемся вызвать метод, не определен.

Глобальный объект — объект верхнего уровня, в котором хранятся глобальные переменные. В браузерах это объект window.

Если вы используете конструктор для создания объектов, а при обращении к ним объекты оказываются неопределенными — проверьте свой код и убедитесь в том, что конструктор используется с оператором **new**.
А еще не держите наклонно открытую пробирку над дорогим ноутбуком — это тоже плохо кончится!

Head First: Привет, `new`, где вы скрывались? Как мы добрались до главы 12, ни разу не увидев вас?

new: Существует великое множество сценариев, авторы которых не используют меня или используют, толком не понимая.

Head First: Почему?

new: Многие разработчики просто используют объектные литералы или копируют готовый код с моим участием без понимания того, как я работаю.

Head First: Это верно... Объектные литералы удобны, а мне еще не совсем ясно, когда и как вас стоит использовать.

new: Пожалуй, вы правы, я отношусь к категории нетривиальных возможностей. Хтобы уметь правильно использовать меня, необходимо понимать, как работают объекты, как работают функции, как работает `this`... Даже просто для того, чтобы узнать о моем существовании, потребуется изрядная подготовка!

Head First: Можете рассказать о себе пару слов? Наши читатели уже знают об объектах, функциях и `this`, самое время им побольше узнать о вас.

new: Дайте подумать... Хорошо, вот: я — оператор, который работает совместно с функциями-конструкторами для создания новых объектов.

Head First: Ммм... Даже неудобно говорить, но это явно не «в двух словах».

new: Ладно вам, я ведь все-таки оператор, а не пиарщик.

Head First: Хорошо, у меня возникло несколько вопросов. Прежде всего — вы оператор?

new: Да! Я оператор. Поставьте меня перед вызовом функции, и я полностью изменю ситуацию. Оператор выполняет действия со своими операндами. В моем случае операнд только один, и это вызов функции.

Head First: Тогда расскажите в подробностях, как именно вы работаете.

new: Сначала я создаю новый объект. Все думают, что объект создается конструктором, но это делаю я. Совершенно неблагодарное занятие.

Head First: Продолжайте...

new: Потом я вызываю функцию-конструктор и делаю так, чтобы новый объект, который я создал, был связан с ключевым словом `this` в теле функции.

Head First: Зачем вы это делаете?

new: Чтобы команды в теле функции могли получить доступ к объекту. В конце концов, конструктор нужен именно для того, чтобы расширять созданный объект новыми свойствами и методами. Если вы используете конструктор для создания объектов, представляющих собак или машины, вы захотите дополнить их свойствами, верно?

Head First: Верно. И что потом?

new: Потом я делаю так, чтобы вновь созданный объект был возвращен из конструктора. Это делается просто для удобства, чтобы разработчику не приходилось возвращать его вручную.

Head First: Действительно удобно. Неужели кому-то захочется использовать объектные литералы после знакомства с вами?

new: Вообще-то мы с объектным литералом — старые друзья. Он отличный парень, и я сам бы использовал его для создания объекта «на скорую руку». А я лучше подхожу для создания множества похожих объектов, когда вы хотите организовать повторное использование кода, обеспечить единство структуры объектов... а также чтобы использовать некоторые нетривиальные возможности, когда будете больше знать по теме.

Head First: Нетривиальные? Расскажите!

new: Давайте не будем отвлекаться. Мы подробнее поговорим об этом в следующей главе.

Head First: Надо будет перечитать это интервью. До встречи...

Даешь массовое производство!

Вы вовремя научились создавать объекты — нам поступил большой заказ на сборку машин, и собирать их вручную нереально. Чтобы не нарушить график, нужно воспользоваться конструктором. Мы возьмем литералы объектов машин, которые использовались ранее в книге, и воспользуемся ими для создания конструктора.

Внизу представлены разнообразные машины, которые нам нужно построить. Мы решили определить единый набор свойств и методов, чтобы они совпадали у всех машин. Пока не нужно беспокоиться о специальной комплектации, игрушечных машинках и монстрах с реактивными двигателями (мы займемся ими позднее). Немного осмотритесь, вспомните, и займемся конструктором для создания объектов любых машин с заданным набором свойств и методов.

```
var chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,

    start: function() {
      this.started = true;
    },

    stop: function() {
      this.started = false;
    },

    drive: function() {
      if (this.started) {
        console.log(this.make + " " +
              this.model + " goes zoom zoom!");
      } else {
        console.log("Start the engine first.");
      }
    }
};
```

```
var cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
          : false,
          function() {...},
          unction() {...},
          function() {...}
```

```
var fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,
    start: function() {...},
    stop: function() {
    drive: function()
};
```

```
var taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start: function() {...},
    stop: function() {...},
    drive: function() {...}
};
```

Упражнение

Используйте все, что вы узнали на последних страницах, для создания конструктора Car. Мы рекомендуем делать это в следующем порядке:

1. Начните с ключевого слова function (мы сделали это за вас), за которым следует имя конструктора. Затем приведите список параметров; нам понадобится один параметр для каждого свойства, которому должно присваиваться начальное значение.

2. Затем задайте начальное значение каждого свойства объекта (не забудьте поставить `this` перед именем свойства).

3. Добавьте три метода объектов-машин: start, drive и stop.

```
function ________(___________________________________) {

}
```

Запишите здесь результат.

Сверьтесь с ответами в конце главы, прежде чем двигаться дальше!

Тест-драйв на новых машинах

У нас появился механизм массового производства объектов машин. Создадим несколько объектов и проверим результат. Включим конструктор Car в страницу HTML, а затем добавим тестовый код.

Примечание: у вас ничего не получится, если вы не выполнили упражнение на предыдущей странице!

Ниже приведен код, который использовали мы; пожалуйста, не стесняйтесь изменять и дополнять его:

Сначала мы используем конструктор для создания всех машин в главе 5.

```
var chevy = new Car("Chevy", "Bel Air", 1957, "red", 2, false, 1021);
var cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
var taxi = new Car("Webville Motors", "Taxi", 1955, "yellow", 4, false, 281341);
var fiat = new Car("Fiat", "500", 1957, "Medium Blue", 2, false, 88000);
```

Но зачем ограничиваться?

```
var testCar = new Car("Webville Motors", "Test Car", 2014, "marine", 2, true, 21);
```

Создадим тестовую машину специально для этой книги!

Вы можете добавить собственные машины, реальные или вымышленные.

Видите, как легко создавать новые объекты с конструкторами? А теперь проведем тест-драйв:

```
var cars = [chevy, cadi, taxi, fiat, testCar];

for(var i = 0; i < cars.length; i++) {
    cars[i].start();
    cars[i].drive();
    cars[i].drive();
    cars[i].stop();
}
```

То, что получилось у нас. А как выглядит ваш результат — может, вы добавили собственную машину? Попробуйте изменить тестовый код (например, вызвать drive до запуска двигателя методом start). А может, вам удастся сделать количество вызовов метода drive случайным?

Консоль JavaScript

```
Chevy Bel Air goes zoom zoom!
Chevy Bel Air goes zoom zoom!
GM Cadillac goes zoom zoom!
GM Cadillac goes zoom zoom!
Webville Motors Taxi goes zoom zoom!
Webville Motors Taxi goes zoom zoom!
Fiat 500 goes zoom zoom!
Fiat 500 goes zoom zoom!
Webville Motors Test Car goes zoom zoom!
Webville Motors Test Car goes zoom zoom!
```

Не спешите расставаться с объектными литералами

Мы уже сравнивали конструкторы с объектными литералами и упоминали о том, что литералы тоже достаточно полезны, но еще не привели ни одного убедительного примера. Давайте немного переработаем код конструктора Car — вы увидите, что объектные литералы иногда упрощают код, делают его более понятным и упрощают его сопровождение.

Вернемся к конструктору Car и посмотрим, как будет выглядеть код с объектными литералами.

Посмотрите, сколько параметров! Мы насчитали семь.

С каждым добавленным параметром (а они всегда добавляются с ростом требований к объектам) читать этот код становится все труднее.

```
function Car(make, model, year, color, passengers, convertible, mileage) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
    this.passengers = passengers;
    this.convertible = convertible;
    this.mileage = mileage;
    this.started = false;

    this.start = function() {
        this.started = true;
    };
    // Остальные методы
}
```

А когда мы пишем код, который вызывает этот конструктор, приходится следить за тем, чтобы параметры передавались строго в указанном порядке.

Проблема в том, что конструктор Car получает слишком много параметров; это усложняет его чтение и сопровождение кода. Также усложняется написание кода вызова этого конструктора. То, что на первый взгляд кажется простым неудобством, в итоге создает больше ошибок, чем вы можете себе представить, — причем часто ошибок коварных и трудноуловимых.

Такие ошибки трудно обнаружить. Если поменять местами две переменные, код остается синтаксически правильным, но скорее всего, работать будет с ошибками.

А если пропустить какое-нибудь значение, возможны самые неожиданные последствия!

Однако существует стандартный прием, который может использоваться при передаче аргументов как конструктору, так и любой другой функции. Он работает так: возьмите все аргументы, объедините их в объектный литерал, а затем передайте этот литерал своей функции. В этом случае все значения передаются в одном контейнере (литерал), а вам не нужно беспокоиться о порядке аргументов и параметров.

Давайте перепишем код вызова конструктора Car, а затем слегка переработаем код конструктора и посмотрим, как работает этот прием.

Преобразование аргументов в объектный литерал

Возьмем вызов конструктора Car и преобразуем набор его аргументов в объектный литерал:

Просто берем каждый аргумент и включаем его в объектный литерал с указанием имени свойства. Мы используем те же имена свойств, которые используются в конструкторе.

```
var cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);

var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};
```

В нашем примере сохранен исходный порядок аргументов, но это ни в коем случае не обязательно.

Затем вызов конструктора Car переписывается следующим образом:

```
var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};

var cadi = new Car(cadiParams);
```

И вроде бы небольшое изменение, а код становится намного чище и понятнее (во всяком случае, на наш взгляд).

Теперь конструктору Car передается один аргумент.

Впрочем, работа еще не закончена — сам конструктор ожидает получить семь аргументов, а не один объект. Давайте переработаем код конструктора, а потом протестируем его.

Преобразование конструктора Car

Теперь нужно удалить все отдельные параметры из конструктора Car и заменить их свойствами передаваемого объекта. Новому параметру присваивается имя `params`. Также необходимо слегка переработать код для использования этого объекта. Вот как это делается:

```
var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};

var cadi = new Car(cadiParams);

function Car(params) {
    this.make = params.make;
    this.model = params.model;
    this.year = params.year;
    this.color = params.color;
    this.passengers = params.passengers;
    this.convertible = params.convertible;
    this.mileage = params.mileage;
    this.started = false;

    this.start = function() {
        this.started = true;
    };
    this.stop = function() {
        this.started = false;
    };
    this.drive = function() {
        if (this.started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    };
}
```

Здесь без изменений — мы просто повторяем объектный литерал и вызов конструктора Car со следующей страницы.

Прежде всего, семь параметров конструктора Car заменяются одним параметром для передаваемого объекта.

Затем каждая ссылка на параметр заменяется соответствующим свойством объекта, переданного функции.

В нашем методе параметр конструктора нигде не используется, поскольку мы работаем со свойствами объекта (для обращения к которым используется переменная this). Следовательно, в этом коде ничего изменять вообще не нужно.

Тест-драйв

Обновите `cadi` и другие машины. Протестируйте свой код.

```
cadi.start();
cadi.drive();
cadi.drive();
cadi.stop();
```

Упражнение

Скопируйте конструкторы Car и Dog в один файл, добавьте приведенный ниже код. Запустите программу и запишите результат.

Конструктор Dog приведен на с. 548.

```
var limoParams = {make: "Webville Motors",
                  model: "limo",
                  year: 1983,
                  color: "black",
                  passengers: 12,
                  convertible: true,
                  mileage: 21120};

var limo = new Car(limoParams);
var limoDog = new Dog("Rhapsody In Blue", "Poodle", 40);

console.log(limo.make + " " + limo.model + " is a " + typeof limo);
console.log(limoDog.name + " is a " + typeof limoDog);
```

Результат запишите здесь.

Допустим, кто-то передал вам объект и вы хотите узнать тип этого объекта (Car? Dog? Superman?) или проверить, относится ли он к одному типу с другим объектом. Поможет ли вам в этом оператор typeof?

Часто задаваемые вопросы

В: Напомните, что делает typeof?

О: Оператор typeof возвращает тип операнда. Передавая ему строку, вы получите результат “string”; для объекта возвращается результат “object” и т. д. Передавать можно любой тип: число, строку, булевское значение или более сложный тип (например, объект или функцию). Но оператор typeof не настолько конкретен, чтобы сообщить, что представляет объект — машину или собаку.

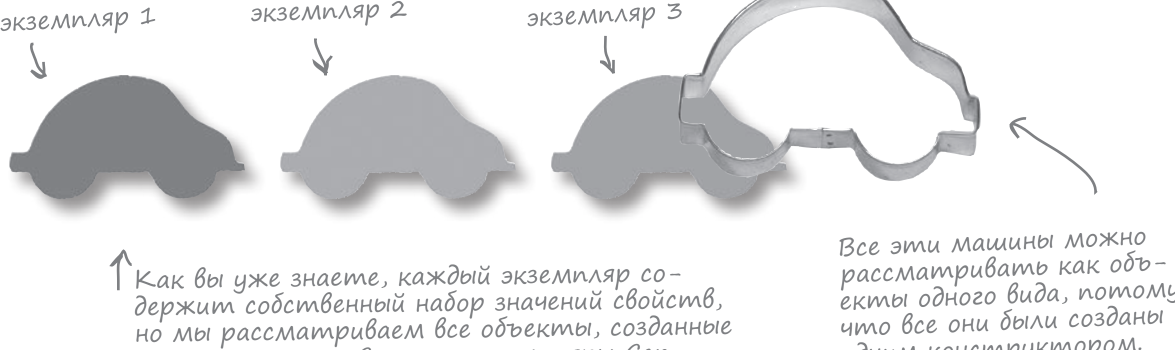
В: Если typeof не может сказать, представляет ли мой объект машину или собаку, то как мне определить, что есть что?

О: Во многих объектно-ориентированных языках — таких, как Java или C++, — существует сильная типизация объектов. В этих языках вы можете проанализировать объект и точно определить его тип. В JavaScript объекты и их типы интерпретируются динамически, то есть более свободно. Из-за этого многие разработчики пришли к выводу, что объектная система JavaScript слабее, но в действительности она обладает большей универсальностью и гибкостью. Динамизм системы типов JavaScript немного усложняет проверку типа — все зависит от того, как разработчик представляет себе собаку или машину. Однако в языке предусмотрен другой оператор, который может предоставить чуть больше информации... Читайте дальше.

Экземпляры

Вы не сможете взять объект JavaScript и определить, к какому типу он относится (то есть представляет ли собаку, машину или что-нибудь еще). В JavaScript объекты являются динамическими структурами и относятся к типу “object” независимо от того, какие свойства или методы они содержат. Тем не менее мы можем получить некоторую информацию об объекте, если будем знать, каким конструктором был создан этот объект.

При каждом вызове конструктора с оператором `new` вы создаете новый экземпляр объекта. И если в каком-то случае для этого использовался конструктор Car, то мы можем неформально считать, что созданный объект представляет машину. А формально созданный объект будет представлять собой экземпляр Car.



Теперь утверждение о том, что объект является экземпляром некоторого конструктора, — не просто слова. Мы можем написать код для определения конструктора, создавшего объект, оператором `instanceof`. Рассмотрим пример:

```
var cadiParams = {make: "GM", model: "Cadillac", year: 1955, color: "tan",
                  passengers: 5, convertible: false, mileage: 12892};

var cadi = new Car(cadiParams);

if (cadi instanceof Car) {
    console.log("Congrats, it's a Car!");
};
```

Оператор instanceof возвращает true, если объект был создан указанным конструктором.

В нашем случае это означает: «Является ли объект cadi экземпляром, созданным конструктором Car?»

Оператор `new` при создании объекта незаметно сохраняет служебную информацию, по которой можно легко определить, каким конструктором был создан объект. Оператор `instanceof` использует эту информацию для проверки того, является ли объект экземпляром, созданным указанным конструктором.

На самом деле все несколько сложнее, чем мы описываем, но об этом будет рассказано в следующей главе.

Упражнение

Нам нужна функция с именем dogCatcher, которая возвращает true, если переданный ей объект является собакой (конструктор Dog), и false в противном случае. Напишите эту функцию и проверьте ее с остальным кодом. Не забудьте свериться с ответом в конце главы, прежде чем читать дальше.

```
function dogCatcher(obj) {




}
```

Добавьте здесь свой код реализации функции dogCatcher.

А это тестовый код.

```
function Cat(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
var meow = new Cat("Meow", "Siamese", 10);
var whiskers = new Cat("Whiskers", "Mixed", 12);

var fido = {name: "Fido", breed: "Mixed", weight: 38};

function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [meow, whiskers, fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    if (dogCatcher(dogs[i])) {
        console.log(dogs[i].name + " is a dog!");
    }
}
```

Итак, объект представляет собаку, если он был создан конструктором Dog, и не иначе?

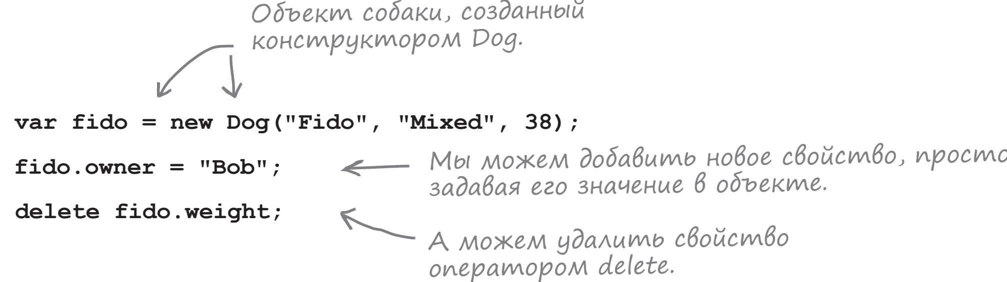
Да, так работает наша программа. В JavaScript не существует сильной концепции типа обекта, поэтому чтобы узнать, представляют ли два объекта кошек или собак, приходится проверять, были ли они созданы одинаково — то есть одной функцией-конструктором. Как уже говорилось ранее, объект считается кошкой, если он был создан констуктором Cat, и собакой, если он был создан конструктором Dog.

В следующей главе будут рассомтрены конструкторы JavaScript и объекты, обладающие еще большей гибкостью. Например, объект такси может быть создан конструктором Taxi, но при этом мы также будем знать, что он одновременно является машиной. Впрочем, пока просто отложите эту идею в дальнем уголке своего мозга — мы вернемся к ней позднее.

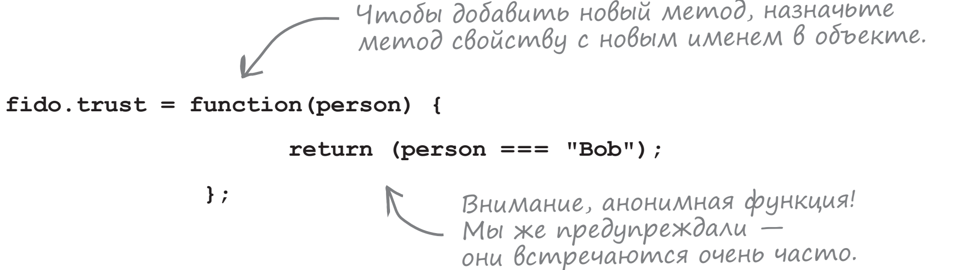
Даже сконструированные объекты могут содержать независимые свойства

Мы много говорили о том, как использовать конструкторы для создания согласованных объектов — объектов, которые имеют одинаковые свойства и одинаковые методы. Но мы не упоминали, что конструкторы не запрещают изменять объект, поскольку это можно проделать после того, как объект создан.

Помните объектные литералы? На их примере мы узнали, как добавлять и удалять свойства созданного объекта. То же самое можно сделать и с объектом, который был создан с помощью конструктора.



В объекты даже можно добавлять новые методы:



Обратите внимание: здесь изменяется только объект fido. Если добавить метод в fido, то метод появится только в этом объекте. У других объектов, представляющих собак, его не будет:

```
var notBite = fido.trust("Bob");

var spot = new Dog("Spot", "Chihuahua", 10);
notBite = spot.trust("Bob");
```

Эта команда работает, потому что метод trust определен в объекте fido. Переменной notBite присваивается значение true.

Этот код не работает, потому что объект spot не содержит метод trust. Выводится сообщение об ошибке: «TypeError: Object #<Dog> has no method 'trust'».

Значит, если я изменю объект, представляющий машину, после его создания, он все равно останется машиной?

Да, машина остается машиной, даже если позднее вы ее измените. Мы имеем в виду, что если вы проверите, остается ли ваш объект экземпляром Car, — он им остается. Допустим, создан следующий объект:

```
var cadiParams = {make: "GM", model: "Cadillac",
                  year: 1955, color: "tan",
                  passengers: 5, convertible: false,
                  mileage: 12892};
var cadi = new Car(cadiParams);
```

В этот объект добавлено новое свойство chrome, и было удалено свойство convertible:

```
cadi.chrome = true;
delete cadi.convertible;
```

cadi остается автомобилем:

```
cadi instanceof Car
```

Дает true.

Вот что мы имели в виду ранее, когда сказали, что в JavaScript используется динамическая система типов.

Остается ли он машиной с практической точки зрения? А если удалить все свойства объекта, останется ли он машиной? Оператор instanceof даст утвердительный ответ, но с житейской точки зрения вы с ним вряд ли согласитесь.

Скорее всего, вам нечасто придется создавать объект конструктором, чтобы позднее превратить его в нечто совершенно неузнаваемое. Как правило, конструкторы используются для создания объектов, которые остаются более или менее постоянными. Но если вам нужен более гибкий объект — что ж, JavaScript предоставляет такую возможность. На стадии проектирования вы должны решить, как использовать конструкторы и объекты наиболее логичным для вас образом (и не забывайте о коллегах).

Конструкторы в реальном мире

Встроенные объекты здорово экономят время. Пожалуй, сегодня я вернусь домой пораньше и успею посмотреть любимый сериал.

JavaScript поставляется с набором готовых конструкторов для создания экземпляров полезных объектов, например объектов для работы с датой и временем, объектов для поиска текста по шаблону и даже объектов, расширяющих возможности работы с массивами. Теперь, когда вы умеете использовать конструкторы и ключевое слово `new`, вы сможете эффективно использовать эти конструкторы, и что еще важнее — созданные ими объекты. Рассмотрим пару примеров, а после них вы будете готовы заняться самостоятельными исследованиями.

Начнем со встроенного объекта даты. Для создания этого объекта используется следующий конструктор:

```
var now = new Date();
```

Создает новый объект, представляющий текущую дату и время.

Вызов конструктора Date возвращает экземпляр Date, представляющий текущее местное время и дату. Располагая таким объектом, вы сможете использовать его методы для выполнения операций с датой (и временем), а также получения различных свойств даты и времени. Несколько примеров:

```
var dateString = now.toString();

var theYear = now.getFullYear();

var theDayOfWeek = now.getDay();
```

Возвращает строковое представление даты вида "Thu Feb 06 2014 17:29:29 GMT-0800 (PST)".

Возвращает год из даты.

Возвращает номер дня недели, представленного объектом даты, например 1 (понедельник).

Объекты даты, представляющие произвольную дату и время, легко создаются с передачей дополнительных аргументов конструктору Date. Допустим, вам понадобился объект даты, представляющий 1 мая 1983 года; он создается так:

```
var birthday = new Date("May 1, 1983");
```

Конструктору можно передать простую строку с датой.

А можно дополнить дату временем:

```
var birthday = new Date("May 1, 1983 08:03 pm");
```

К строке добавляется время.

Конечно, такое описание возможностей объекта даты нельзя назвать даже поверхностным; полный список его свойств и методов приведен в книге *JavaScript: Подробное руководство*.

Объект Array

Еще один интересный встроенный объект предназначен для работы с массивами. Хотя ранее мы создавали массивы в синтаксисе с квадратными скобками [1, 2, 3], массивы также можно создавать конструктором:

```
var emptyArray = new Array();
```

Создаем пустой массив нулевой длины.

Здесь создается новый, пустой объект массива. В него можно в любой момент добавить новые элементы:

```
emptyArray[0] = 99;
```

Эта команда выглядит знакомо: так мы всегда добавляли элементы в массив.

Также можно создать объект массива заданного размера. Допустим, нам нужен массив для трех элементов:

```
var oddNumbers = new Array(3);
oddNumbers[0] = 1;
oddNumbers[1] = 3;
oddNumbers[2] = 5;
```

Создаем массив длины 3 и заполняем данными уже после создания.

Здесь создается массив длиной 3. Изначально три элемента `oddNumbers` не определены, но затем мы присваиваем каждому элементу значение. При желании вы можете легко добавить в массив дополнительные элементы.

Все это не так уж сильно отличается от того, к чему мы привыкли. Объект массива интересен в основном своим набором методов. Метод sort вам уже известен, еще несколько интересных примеров:

```
oddNumbers.reverse();

var aString = oddNumbers.join(" - ");

var areAllOdd = oddNumbers.every(function(x) {
    return ((x % 2) !== 0);
});
```

Перестановка значений массива в обратном порядке (теперь массив oddNumbers содержит элементы 5, 3, 1). Обратите внимание: метод изменяет исходный массив, а не копию.

Метод join создает строку из значений oddNumbers, разделяя их последовательностью " - ", и возвращает полученную строку. В данном случае будет возвращена строка "5 - 3 - 1".

Метод every получает функцию, и для каждого значения в массиве проверяет, какой результат вернет функция для этого значения — true или false. Если функция возвращает true для всех элементов массива, то и метод every возвращает true.

И снова это всего лишь вершина айсберга, а за подробной информацией об объектах массивов обращайтесь к книге *JavaScript: Подробное руководство*. Сейчас вы знаете все необходимое для самостоятельного изучения темы.

Верное замечание. Синтаксис с квадратными скобками [ ], который мы использовали для создания массива, в действительности является сокращенной записью для прямого использования конструктора Array. Рассмотрим два эквивалентных способа создания пустых массивов:

```
var items = new Array();

var items = [];
```

Эти команды делают одно и то же. Запись с квадратными скобками поддерживается в языке JavaScript для того, чтобы упростить работу программиста при создании массивов.

А следующая команда:

```
var items = ["a", "b", "c"];
```

Такой синтаксис массивов называется «литеральным».

является сокращенной записью для использования конструктора:

```
var items = new Array("a", "b", "c");
```

При передаче нескольких аргументов создается массив с элементами, содержащими переданные значения.

Объекты, созданные на базе литералов и с прямым вызовом конструкторов, ничем не отличаются, так что методы могут использоваться в обоих случаях.

Возникает логичный вопрос — в каких случаях стоит использовать конструктор вместо литерала? Конструкторы удобны при создании массивов, размер которых определяется на стадии выполнения, с последующим добавлением в них элементов:

```
var n = getNumberOfWidgetsFromDatabase();
var widgets = new Array(n);
for(var i=0; i < n; i++) {
    widgets[i] = getDatabaseRecord(i);
}
```

В этом коде используются большие массивы, а конкретный размер такого массива неизвестен до выполнения программы.

Итак, синтаксис с литералами прекрасно подходит для создания объектов массивов «на скорую руку», а применение конструктора Array может оказаться более эффективным при построении массива на программном уровне. Используйте любой способ — или их комбинацию по своему усмотрению.

Другие встроенные объекты

Кроме дат и массивов, в JavaScript существуют и другие встроенные объекты. Язык поддерживает множество объектов, которые могут пригодиться время от времени. Ниже приведен неполный список, если вас заинтересует эта тема, ищите информацию в Интернете!

Object Конструктор Object создает объекты. Как и в случае с массивами, литеральный синтаксис объектов {} эквивалентен new Object(). Вскоре эта тема будет рассмотрена более подробно.

Math Объект содержит свойства и методы математических операций (например, Math.PI и Math.random()).

RegExp Конструктор предназначен для создания объектов регулярных выражений, выполняющих поиск текста по шаблону (достаточно сложному).

Error Конструктор создает стандартные объекты ошибок, которые будут полезны, если вы перехватываете ошибки в своем коде.

Часто задаваемые вопросы

В: Не понимаю, как работают конструкторы Date и Array, похоже, они поддерживают нуль и более аргументов. В случае Date при вызове без аргументов я получаю объект текущей даты, но я могу передавать аргументы для получения других дат. Как это делается?

О: Верное замечание. Вы можете писать функции, которые работают по-разному в зависимости от количества аргументов. Если конструктор Array вызывается без аргументов, он знает, что нужно создать пустой массив; если аргумент один — конструктор рассматривает его как размер массива; если аргументов больше, то все дополнительные аргументы задают начальные значения элементов.

В: Я могу делать то же самое в своих конструкторах?

О: Конечно. Мы эту тему еще не рассматривали, но каждая функция получает объект со всеми аргументами, переданными этой функции. По этому объекту конструктор может определить, какая информация была передана при вызове, и среагировать соответствующим образом. Также существуют другие способы, основанные на проверке параметров на undefined.

В: Мы использовали объект Math ранее в книге. Почему мы включали вызов "new Math", чтобы создать экземпляр перед использованием?

О: Хороший вопрос. Вообще-то Math не является конструктором и даже функцией — это объект. И как вы знаете, встроенный объект Math используется для таких операций, как получение числа «пи» (`Math. PI`) или генерирование случайного числа (`Math.random`). Рассматривайте Math как объектный литерал с набором полезных свойств и методов, готовый к использованию в любом коде JavaScript. Его имя начинается с прописной буквы только для того, чтобы вы знали, что это встроенный объект JavaScript.

В: Оператор instanceof позволяет узнать, является ли объект экземпляром заданного конструктора, но как понять, были ли два объекта созданы одним конструктором?

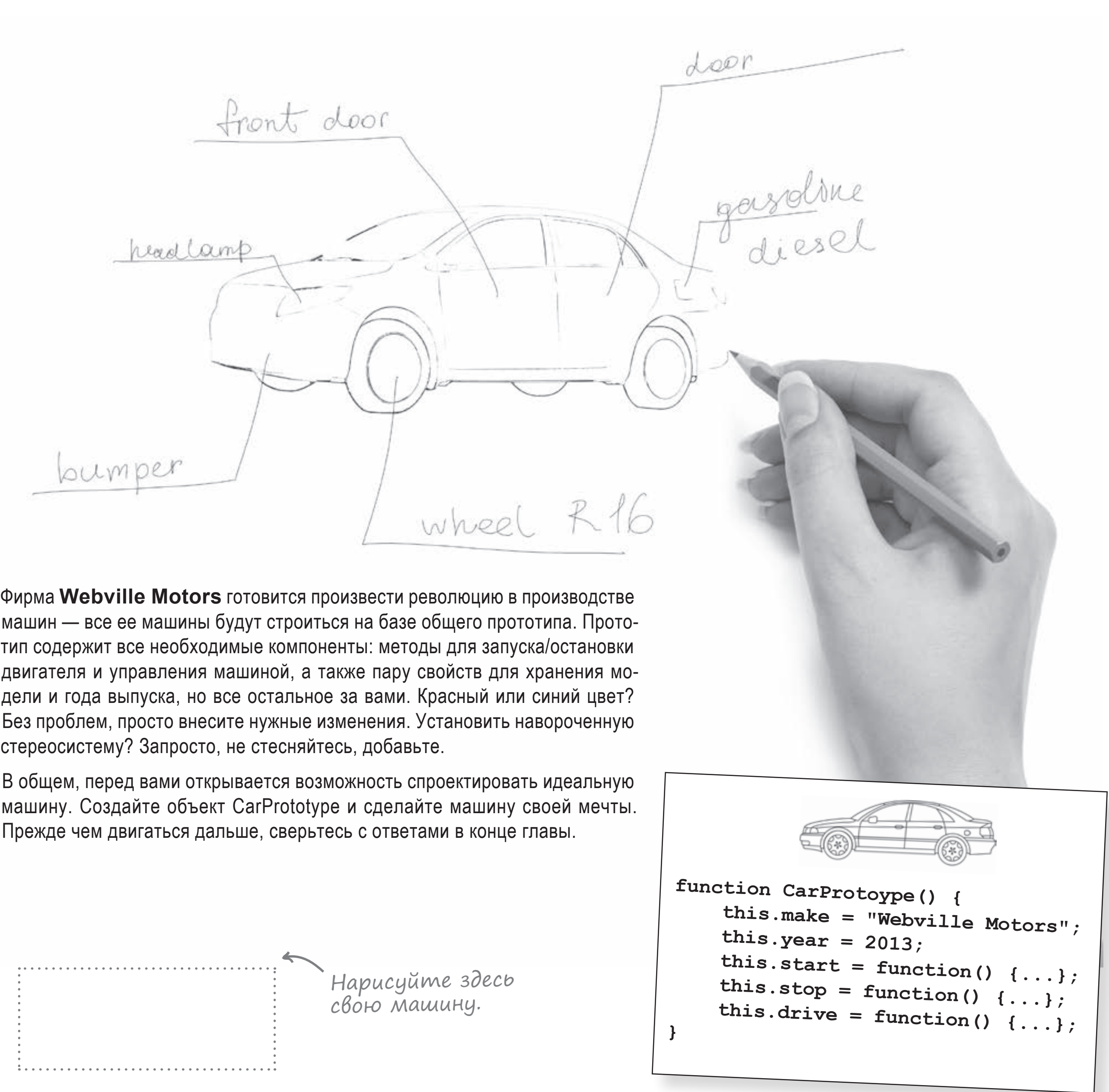
О: Это легко проверить:

```
((fido instanceof Dog) &&
        (spot instanceof Dog))
```

Если результат выражения равен true, значит, объекты fido и spot действительно были созданы одним конструктором.

В: Если я создаю объект при помощи объектного литерала, экземпляром чего он будет? А может, он вообще не является экземпляром?

О: Объектный литерал является экземпляром Object. Считайте, что конструктор Object создает наиболее обобщенные объекты в JavaScript. В следующей главе место Object в системе объектов JavaScript будет рассмотрено более подробно.



Фирма **Webville Motors** готовится произвести революцию в производстве машин — все ее машины будут строиться на базе общего прототипа. Прототип содержит все необходимые компоненты: методы для запуска/остановки двигателя и управления машиной, а также пару свойств для хранения модели и года выпуска, но все остальное за вами. Красный или синий цвет? Без проблем, просто внесите нужные изменения. Установить навороченную стереосистему? Запросто, не стесняйтесь, добавьте.

В общем, перед вами открывается возможность спроектировать идеальную машину. Создайте объект CarPrototype и сделайте машину своей мечты. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
function CarProtoype() {
    this.make = "Webville Motors";
    this.year = 2013;
    this.start = function() {...};
    this.stop = function() {...};
    this.drive = function() {...};
}
```

Нарисуйте здесь свою машину.

Здесь настройте прототип.

И что нам это дает? Узнаете в следующей главе! Кстати говоря, эта глава подходит к концу... Но еще остался список ключевых моментов!

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Объектные литералы** хорошо подходят для создания небольшого количества объектов.
- **Конструкторы** хорошо подходят для создания множества похожих объектов.
- Конструктор — функция, которая должна использоваться в сочетании с оператором **new**. Имена конструкторов по общепринятому соглашению начинаются с прописной буквы.
- Конструктор создает объекты, имеющие одинаковую структуру, с общими именами свойств и методов.
- Используйте оператор **new** в сочетании с вызовом конструктора для создания объекта.
- При использовании **new** с конструктором создается новый пустой объект, который назначается **this** в теле конструктора.
- Используйте **this** в конструкторе для обращения к создаваемому объекту и добавления свойств в объект.
- Новый объект автоматически возвращается конструктором.
- Если вы забудете использовать **new** с конструктором, объект не будет создан. Это приведет к появлению трудноуловимых ошибок в коде.
- Аргументы, передаваемые конструктору, используются для инициализации свойств создаваемого объекта.
- Если конструктор содержит слишком много параметров, возможно, вам стоит объединить их в один параметр-объект.
- Чтобы проверить, был ли объект создан конкретным конструктором, используйте оператор **instanceof**.
- Объекты, созданные конструктором, можно изменять — как и объекты, созданные на базе литералов.
- JavaScript предоставляет набор готовых конструкторов для создания полезных объектов (даты, регулярные выражения, массивы и т. д.).

Возьми в руку карандаш Решение

Нам нужна ваша помощь. Мы использовали объектные литералы для создания объектов, представляющих уток. Сможете ли вы написать конструктор для создания таких объектов? Ниже приведен объектный литерал, который следует взять за основу для написания вашего конструктора. Ознакомьтесь с нашим решением.

```
var duck = {
    type: "redheaded",
    canFly: true
}
```

Пример объектного литерала.

Напишите конструктор для создания уток.

```
function Duck(type, canFly) {
    this.type = type;
    this.canFly = canFly;
}
```

P. S. Мы знаем, что вы еще не до конца понимаете смысл происходящего, так что пока сосредоточимся на синтаксисе.

Упражнение Решение

Маленькое практическое упражнение для закрепления новых знаний: включите этот код в страницу и попробуйте выполнить его. Запишите полученный результат.

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    var size = "small";
    if (dogs[i].weight > 10) {
        size = "large";
    }
    console.log("Dog: " + dogs[i].name
                + " is a " + size
                + " " + dogs[i].breed);
}
```

Консоль JavaScript

```
Dog: Fido is a large Mixed
Dog: Fluffy is a large Poodle
Dog: Spot is a small Chihuahua
```

СТАНЬ браузером. Решение

Ниже приведен код JavaScript, содержащий ошибки. Представьте себя на месте браузера и найдите ошибки в этом коде. Сверьте свое решение с нашим.

```
function widget(partNo, size) {
    var this.no = partNo;
    var this.breed = size;
}

function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}

var widgetA = widget(100, "large");
var widgetB = new widget(101, "small");
var formFactorA = newFormFactor("plastic", widgetA);
var formFactorB = new ForumFactor("metal", widgetB);
```

Если функция widget станет конструктором, ее имя должно начинаться с прописной буквы W. Ошибкой это не является, но это полезное соглашение, которое стоит соблюдать.

Ставить "var" перед this не обязательно. Мы не объявляем новые переменные, а добавляем свойства в объект.

Кроме того, по общепринятому соглашению имена параметров обычно совпадают с именами свойств — поэтому, вероятно, лучше использовать запись this.partNo и this.size.

Вместо символов «;» используются запятые. Помните, что конструктор должен содержать обычные команды вместо разделенных запятыми пар «имя/значение».

Мы возвращаем this, но это не обязательно — конструктор сделает это за нас. Эта команда не приведет к ошибке, но она не нужна.

Забыли new!

Между new и именем конструктора должен стоять пробел.

Имя конструктора записано с ошибкой.

У нас есть конструктор для создания объектов, представляющих разные кофейные напитки, но в этом конструкторе нет методов.

Нам понадобится метод getSize, который возвращает строку в зависимости от объема напитка (в унциях):

- 8 унций — малый (small);
- 12 унций — средний (medium);
- 16 унций — большой (large).

Также нужен метод toString, который возвращает строку с описанием заказа, например: «You've ordered a small House Blend coffee».

Запишите код и протестируйте его в браузере. Попробуйте создать напитки с разными объемами. Ниже приведено наше решение.

```
function Coffee(roast, ounces) {
    this.roast = roast;
    this.ounces = ounces;
    this.getSize = function() {
        if (this.ounces === 8) {
            return "small";
        } else if (this.ounces === 12) {
            return "medium";
        } else if (this.ounces === 16) {
            return "large";
        }
    };
    this.toString = function() {
        return "You've ordered a " + this.getSize() + " "
               + this.roast + " coffee.";
    };
}

var houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());

var darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

Метод getSize проверяет свойство ounces объекта и возвращает соответствующую строку.

Помните: this представляет объект, метод которого мы вызываем. Следовательно, при вызове houseBlend.size ссылка this будет соответствовать объекту houseBlend.

Метод toString возвращает строковое описание объекта. Для получения информации об объеме напитка используется метод getSize.

Мы создаем два объекта, вызываем для них метод toString и выводим полученные строки.

Консоль JavaScript

```
You've ordered a medium House Blend coffee.
You've ordered a large Dark Roast coffee.
```

Наш результат; ваш должен выглядеть примерно так же.

Используйте все, что вы узнали на последних страницах, для создания конструктора Car. Мы рекомендуем делать это в следующем порядке:

1. Начните с ключевого слова function (мы сделали это за вас), за которым следует имя конструктора. Затем приведите список параметров, понадобится один параметр для каждого свойства, которому должно присваиваться начальное значение.
2. Затем задайте начальное значение каждого свойства объекта (не забудьте поставить `this` перед именем свойства).
3. Добавьте три метода объектов-машин: start, drive и stop.

Ниже приводится наше решение.

Конструктору назначается имя Car.

Семь параметров, по одному для каждого инициализируемого свойства.

```
(1)
function Car(make, model, year, color, passengers, convertible, mileage) {

  (2)
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
    this.passengers = passengers;
    this.convertible = convertible;
    this.mileage = mileage;
    this.started = false;

  (3)
    this.start = function() {
        this.started = true;
    };
    this.stop = function() {
        this.started = false;
    };
    this.drive = function() {
        if (this.started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    };
}
```

Каждое свойство объекта car инициализируется значением параметра. Обратите внимание на совпадение имен свойств и параметров, как того требует общепринятое соглашение.

Свойство started инициализируется значением false.

Методы остались теми же, но теперь они назначаются свойствам объекта в несколько ином синтаксисе, потому что здесь используется конструктор, а не объектный литерал.

Скопируйте конструкторы Car и Dog в один файл, добавьте приведенный ниже код. Запустите программу и запишите результат. Вот что получилось у нас:

```
var limoParams = {make: "Webville Motors",
                  model: "limo",
                  year: 1983,
                  color: "black",
                  passengers: 12,
                  convertible: true,
                  mileage: 21120};

var limo = new Car(limoParams);
var limoDog = new Dog("Rhapsody In Blue", "Poodle", 40);

console.log(limo.make + " " + limo.model + " is a " + typeof limo);
console.log(limoDog.name + " is a " + typeof limoDog);
```

Консоль JavaScript

```
Webville Motors limo is a object
Rhapsody In Blue is a object
```

Упражнение
Решение

Нам нужна функция с именем dogCatcher, которая возвращает true, если переданный ей объект является собакой (конструктор Dog), и false в противном случае. Напишите эту функцию и проверьте ее с остальным кодом. Наше решение:

```
function dogCatcher(obj) {
    if (obj instanceof Dog) {
        return true;
    } else {
        return false;
    }
}
```

Или более компактно:

```
function dogCatcher(obj) {
    return (obj instanceof Dog);
}
```

```
function Cat(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
var meow = new Cat("Meow", "Siamese", 10);
var whiskers = new Cat("Whiskers", "Mixed", 12);

var fido = {name: "Fido", breed: "Mixed", weight: 38};

function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [meow, whiskers, fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    if (dogCatcher(dogs[i]) {
        console.log(dogs[i].name + " is a dog!");
    }
}
```

Консоль JavaScript

```
Fluffy is a dog!
Spot is a dog!
```

Фирма **Webville Motors** готовится произвести революцию в производстве машин — все ее машины будут строиться на базе общего прототипа. Прототип содержит все необходимые компоненты: методы для запуска/остановки двигателя и управления машиной, а также пару свойств для хранения модели и года выпуска, но все остальное за вами. Красный или синий цвет? Без проблем, просто внесите нужные изменения. Установить навороченную стереосистему? Запросто, не стесняйтесь, добавьте.

В общем, перед вами открывается возможность спроектировать идеальную машину. Ниже приведено наше решение.

```
function CarProtoype() {
    this.make = "Webville Motors";
    this.year = 2013;
    this.start = function() {...};
    this.stop = function() {...};
    this.drive = function() {...};
}
```

Нарисуйте здесь свою машину.

```
var taxi = new CarPrototype();

taxi.model = "Delorean Remake";

taxi.color = "silver";

taxi.currentTime = new Date();

taxi.fluxCapacitor = {type: "Mr. Fusion"};

taxi.timeTravel = function(date) {...};
```

Здесь настройте прототип.

И что нам это дает? Узнаете в следующей главе! Кстати говоря, эта глава подошла к концу.

13 использование прототипов

Сильные объекты

Научиться создавать объекты — только начало. Пришло время «накачать мышцы» — изучить расширенные средства определения **отношений** между объектами и организовать **совместное использование кода**. Кроме того, нам понадобятся механизмы расширения существующих объектов. Иначе говоря, нам нужно расширить свой инструментарий работы с объектами. В этой главе вы увидите, что в JavaScript реализована достаточно мощная **объектная модель**, но она немного отличается от модели традиционных объектно-ориентированных языков. Вместо типичных объектно-ориентированных систем на базе классов JavaScript использует модель **прототипов** — объектов, способных наследовать и расширять поведение других объектов. Какая в этом польза для вас? Вскоре узнаете. Итак, за дело...

А если вы не изучали классическое наследование, считайте, вам повезло — забывать ничего не придется!

Если вы привыкли к Java, C++ или другим языкам, основанным на классической модели ООП, давайте поговорим.

А если не привыкли... что, вы куда-то торопитесь? Устраивайтесь поудобнее, возможно, вы узнаете что-то полезное.

Скажем прямо: в JavaScript не существует классической объектно-ориентированной модели, в которой объекты создаются на базе классов. Более того, *в JavaScript классов вообще нет*. В JavaScript объекты наследуют поведение *от других объектов*; этот механизм называется *наследованием через прототипы*.

Ситуация может измениться: в следующей версии JavaScript могут добавиться классы. Новейшую информацию по этой теме можно найти на сайте wickedlysmart.com/hfjs.

Разработчики с опытом объектно-ориентированного программирования часто жалуются на JavaScript, но знайте: языки с наследованием через прототипы более универсальны, чем классические объектно-ориентированные языки. Они обладают большей гибкостью, эффективностью и выразительностью... Причем такой выразительностью, что при желании можно использовать JavaScript для реализации классического наследования.

Остается читателю для самостоятельных упражнений.

Итак, если вы обучены искусству классического объектно-ориентированного программирования — расслабьтесь, постарайтесь избавиться от предубеждений и приготовьтесь увидеть нечто новое. А если вы понятия не имеете, что такое «классическое объектно-ориентированное программирование», значит, вы начинаете с самого начала, что может оказаться вашим преимуществом.

Представление объектов на диаграммах

Диаграммы объектов, которые мы использовали ранее, были вполне симпатичны, но в этой главе рассматриваются *серьезные объекты*, поэтому и диаграммы должны стать более серьезными. На самом деле в старых диаграммах нет ничего плохого, но в этой главе они будут настолько сложными, что нам просто не удастся вместить в них все необходимое.

Итак, без лишних разговоров познакомимся с новым форматом:

КЛАССИКА

name: "Fido"
breed: "Mixed"
weight: 38
bark()
Dog

Это свойства.
Конструктор.
А это методы.

НОВЫЙ И УЛУЧШЕННЫЙ ВАРИАНТ

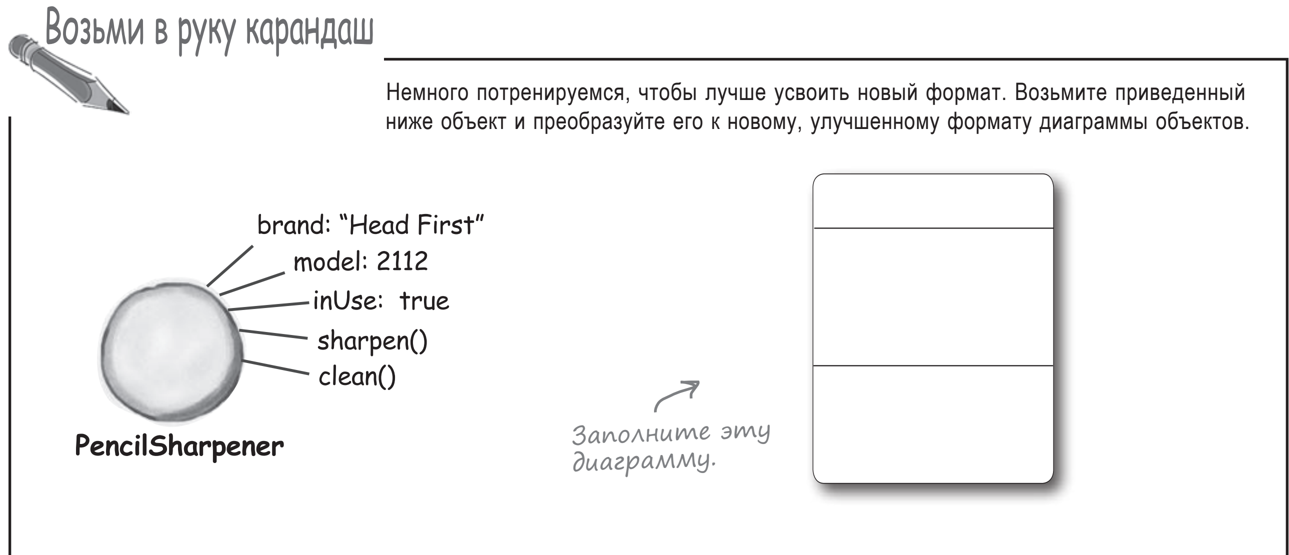
Конструктор.

Dog
name: "Fido" breed: "Mixed" weight: 38
bark()

Это свойства.
А это методы.

Возьми в руку карандаш

Немного потренируемся, чтобы лучше усвоить новый формат. Возьмите приведенный ниже объект и преобразуйте его к новому, улучшенному формату диаграммы объектов.



Снова о конструкторах: код используется повторно, но насколько эффективно?

Помните конструктор Dog, который мы создали в предыдущей главе? Давайте посмотрим еще раз, что нам дает использование конструктора:

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
```

Каждый объект инициализируется своими значениями и содержит единый набор свойств.

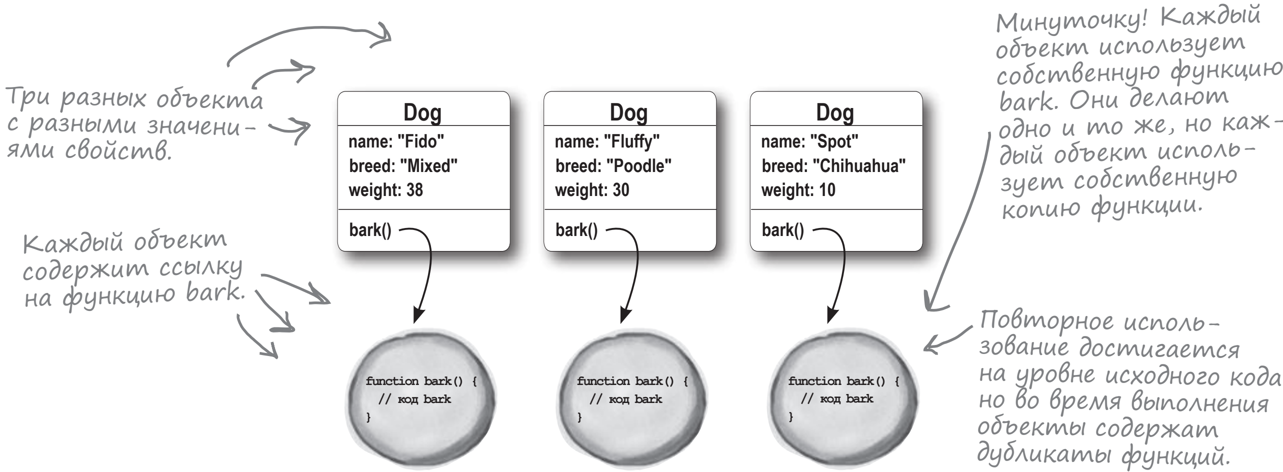
Каждый объект содержит метод bark.

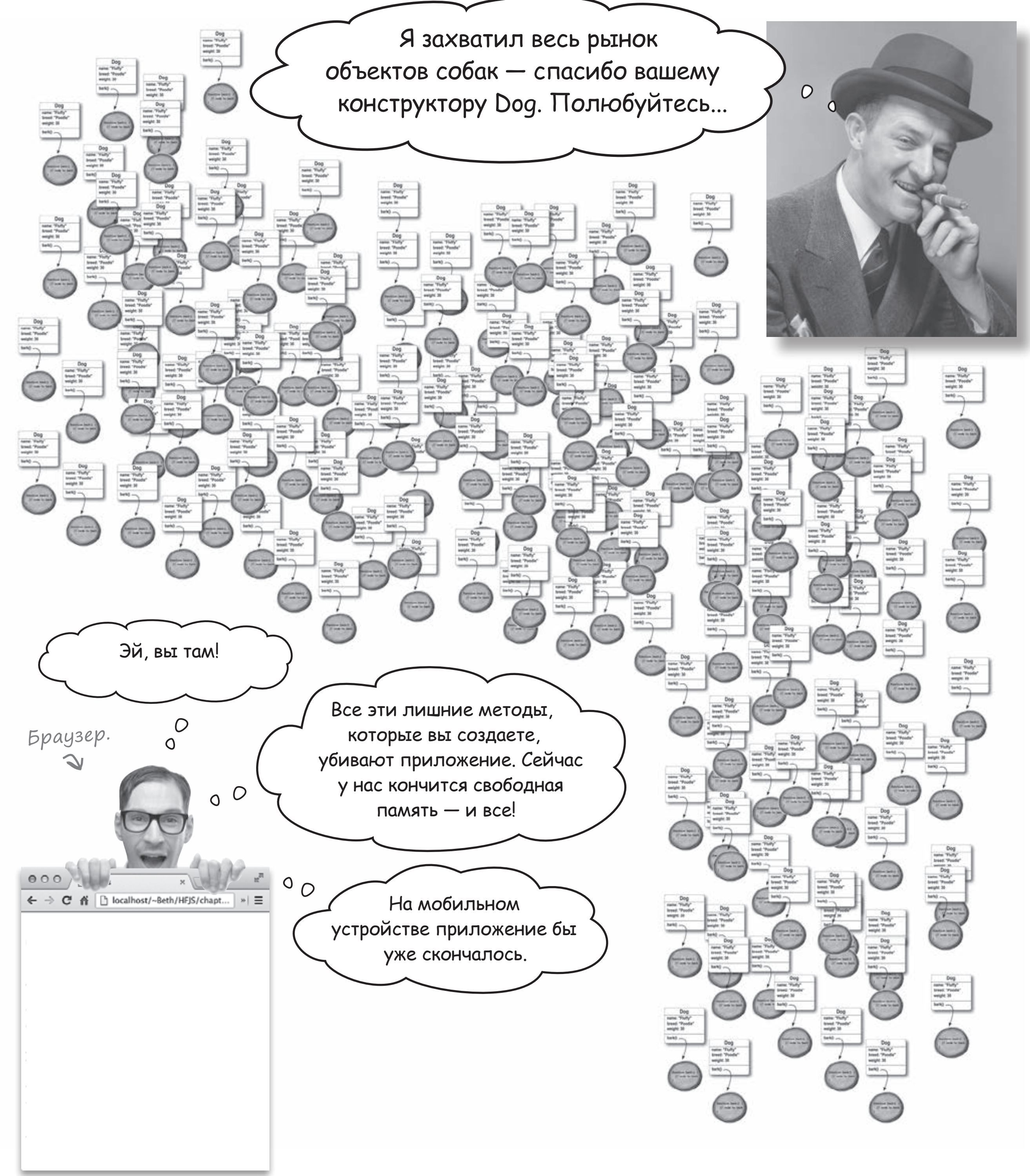
И что еще лучше — код совместно используется всеми объектами, созданными конструктором Dog.

Итак, использование конструктора дает нам аккуратные, похожие объекты, которые мы можем настраивать по своему усмотрению, а также использовать определенные в них методы (в данном случае такой метод всего один — `bark`). Кроме того, все объекты получают один и тот же код от конструктора, что избавит нас от многих хлопот, если в будущем ситуация вдруг изменится. Все это, конечно, хорошо, но давайте посмотрим, что происходит при выполнении следующего фрагмента:

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```

Фрагмент создает три объекта, представляющих собак. Воспользуемся нашей новой диаграммой объектов и посмотрим, что из этого получается:




Я захватил весь рынок объектов собак — спасибо вашему конструктору Dog. Полюбуйтесь...
Эй, вы там!
Браузер.
Все эти лишние методы, которые вы создаете, убивают приложение. Сейчас у нас кончится свободная память — и все!
На мобильном устройстве приложение бы уже скончалось.
localhost/~Beth/HFJS/chapt...

Лично я считаю, что каждая собака должна иметь свой метод bark. Без обид!

Дублирование методов действительно создает проблемы?

Вообще-то да. Как правило, мы не хотим, чтобы при каждом создании экземпляра конструктором создавался новый набор методов. Дублирование ухудшает быстродействие приложения и расходует ресурсы, которые могут быть ограничены, особенно на мобильных устройствах. Как вы вскоре увидите, существуют более гибкие и эффективные способы создания объектов JavaScript.

Давайте отступим на шаг и вспомним одну из главных причин для использования конструкторов: мы стремимся к *повторному использованию поведения*. Например, мы создаем множество объектов собак и хотим, чтобы все они использовали один и тот же метод `bark`. Конструктор позволил нам добиться этой цели на уровне исходного кода: весь код метода bark хранится в одном месте (в конструкторе Dog), поэтому при каждом создании экземпляра используется один и тот же код `bark`. Однако на стадии выполнения такое решение выглядит уже не столь привлекательным, потому что каждый экземпляр наделяется собственной копией метода `bark`.

Как правило, под «поведением» объекта понимается набор поддерживаемых им методов.

Проблема возникает из-за того, что мы не в полной мере используем преимущества объектной модели JavaScript, основанной на *прототипах*. В этой модели создаются объекты, которые являются расширениями других объектов, то есть объектов-прототипов.

Что касается демонстрации прототипов, хм... Если бы у нас был *прототип собаки*, на базе которого можно было бы работать дальше...

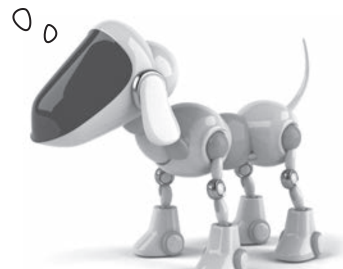
Что такое «прототип»?

Объекты JavaScript способны наследовать свойства и поведение от других объектов. А если говорить конкретнее, в JavaScript используется модель *наследования через прототипы*; соответственно, объект, от которого наследуется поведение, называется *прототипом*. Вся суть этой схемы заключается в наследовании и повторном использовании существующих свойств (включая методы), с расширением свойств во вновь созданном объекте. Впрочем, все это абстрактные рассуждения, так что давайте рассмотрим пример.

Объект, созданный наследованием от другого объекта, получает доступ ко всем его методам и свойствам.

Начнем с прототипа для объекта собаки. Вот как он может выглядеть:

Я — прототип собаки.
Я содержу свойства,
необходимые каждой собаке,
и меня можно использовать как
прототип для создания любой
другой собаки.



Прототип для создания собак: объект со свойствами и методами, которые могут понадобиться каждой собаке.

Прототип

species: "Canine"

bark()
run()
wag()

Содержит свойства, необходимые каждой собаке.

Поведение, которое должное поддерживаться всеми создаваемыми собаками.

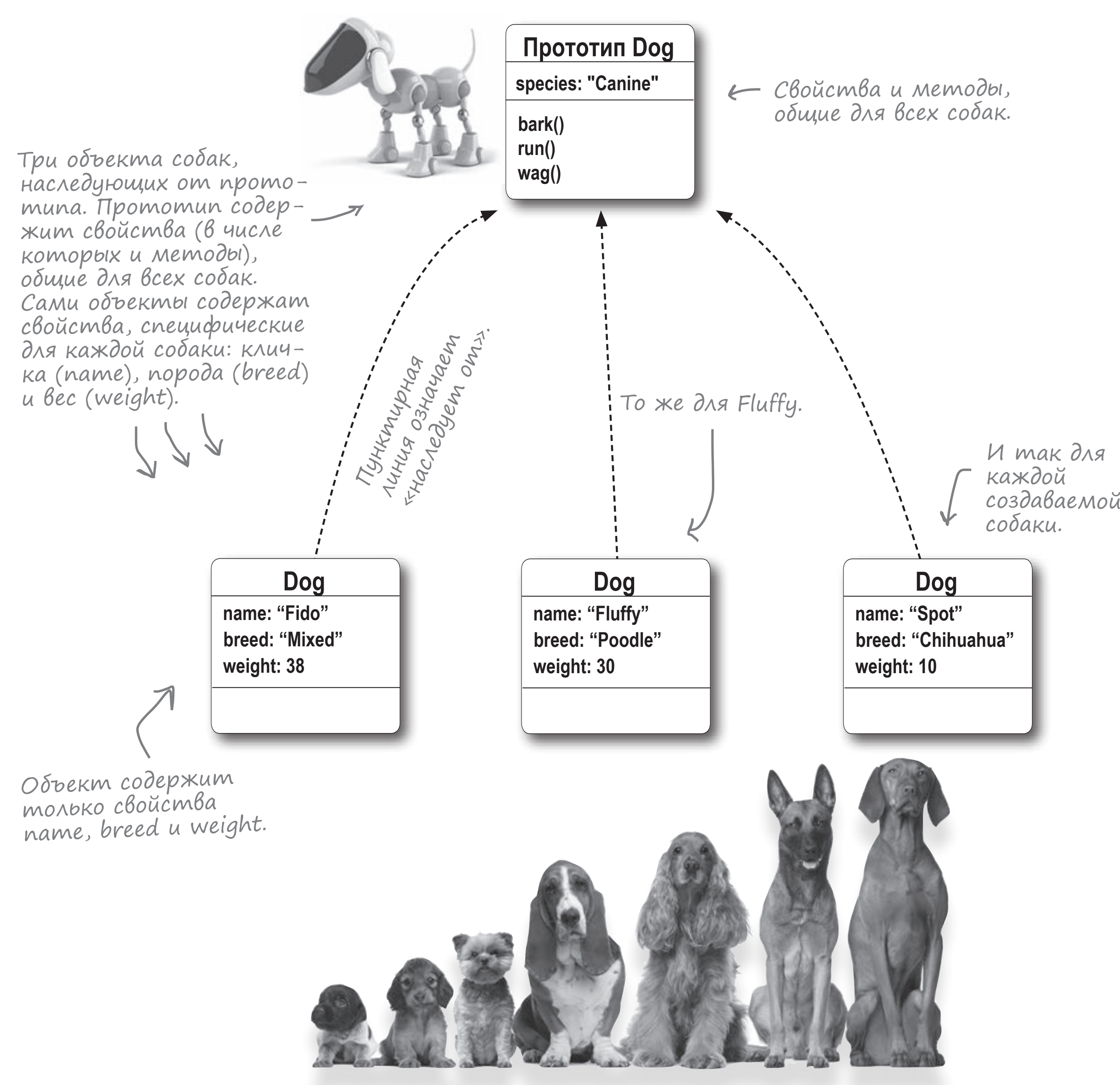
Прототип не включает свойства клички, породы или веса — эти свойства уникальны для каждой собаки и будут предоставляться реальными собаками, наследующими от прототипа.

Имея прототип собаки, мы можем создавать объекты собак, наследующие свойства прототипа. Объект собаки также может дополнять набор свойств прототипа другими свойствами и поведением, присущими конкретным собакам. Например, мы можем добавить в объект каждой собаки свойства для клички, породы и веса.

Если какой-либо собаке нужно будет лаять, бегать или махать хвостом, она может воспользоваться поведением, унаследованным от прототипа. Давайте создадим несколько объектов, чтобы вы лучше поняли, как работает эта схема.

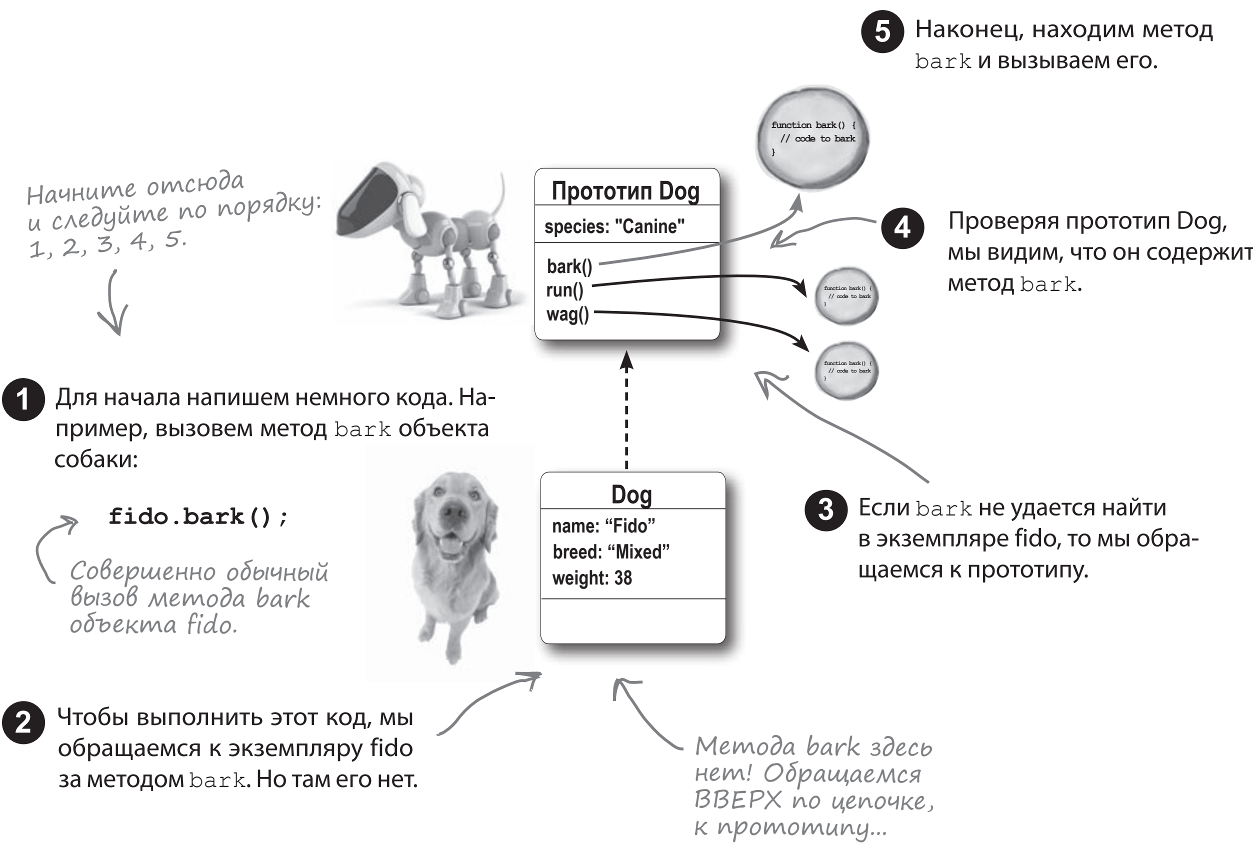
Наследование через прототип

Начнем с построения диаграмм для объектов с кличками Fido, Fluffy и Spot, а затем организуем их наследование от нового прототипа. Наследование будет обозначаться пунктирными линиями, направленными от экземпляров к прототипу. И помните: в прототип включаются методы и свойства, общие для *всех* собак, потому что они будут наследоваться *всеми* собаками. Все свойства, относящиеся к конкретной собаке, например кличка, находятся в экземплярах, потому что они имеют разные значения для разных собак.

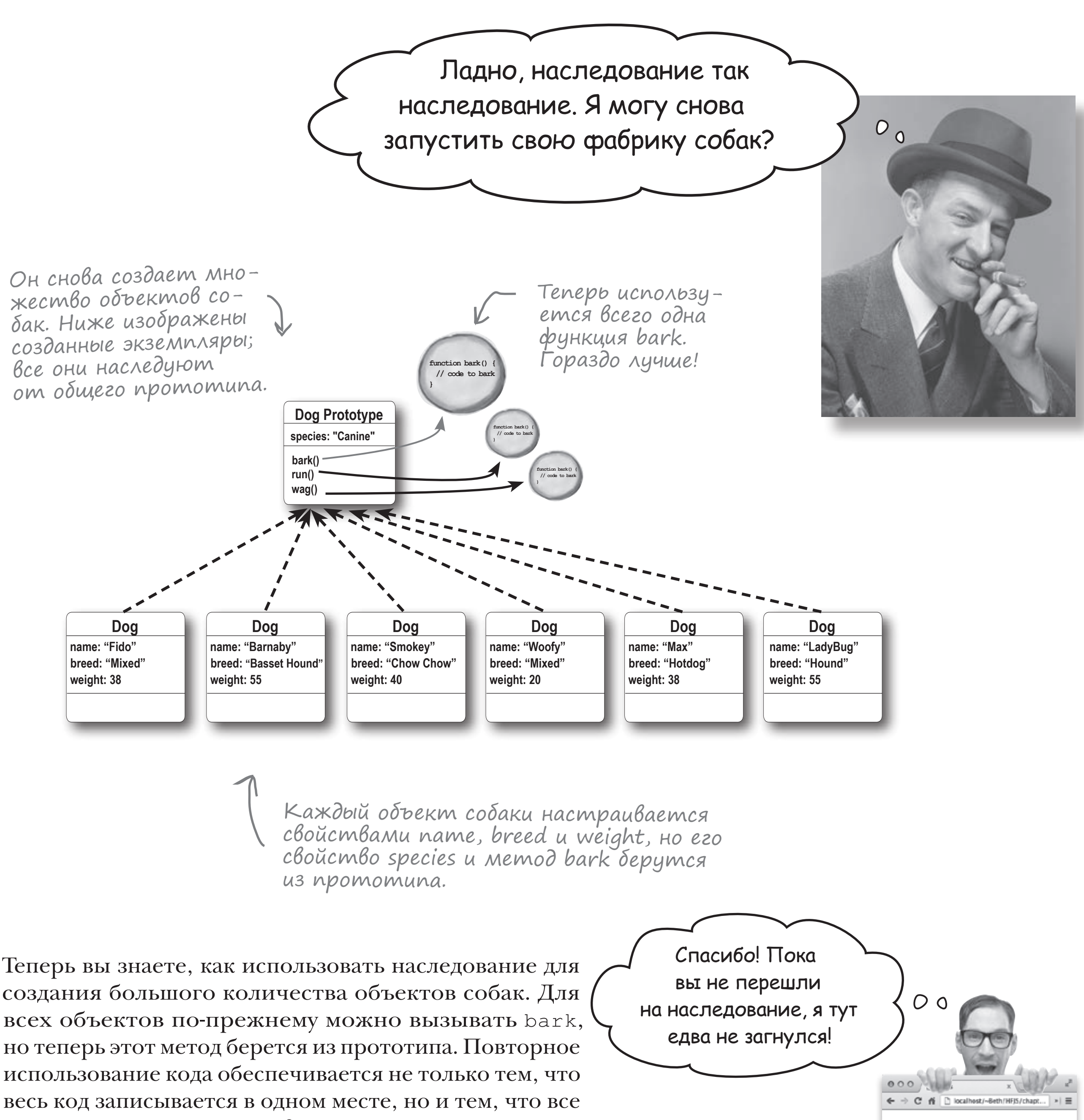


Как работает наследование

Как выполнить метод `bark`, если он определяется не в отдельных экземплярах, а в прототипе? Здесь в игру вступает наследование. Если метод вызывается для экземпляра, в котором указанный метод отсутствует, проверяется прототип метода. Вот как это делается.



Свойства работают примерно так же. Если мы пишем новый код, в котором используется значение `fido.name`, это значение берется из объекта `fido`. Но если нас интересует значение `fido.species`, сначала проверяется объект `fido`; в нем такого свойства нет, поэтому проверяется прототип (где это свойство и обнаруживается).



Теперь вы знаете, как использовать наследование для создания большого количества объектов собак. Для всех объектов по-прежнему можно вызывать `bark`, но теперь этот метод берется из прототипа. Повторное использование кода обеспечивается не только тем, что весь код записывается в одном месте, но и тем, что все экземпляры используют *один и тот же* метод `bark` во время выполнения, благодаря чему удается избежать непроизводительных затрат памяти.

С помощью прототипов можно быстро строить объекты, которые эффективно используют код и могут расширяться новым поведением и свойствами.

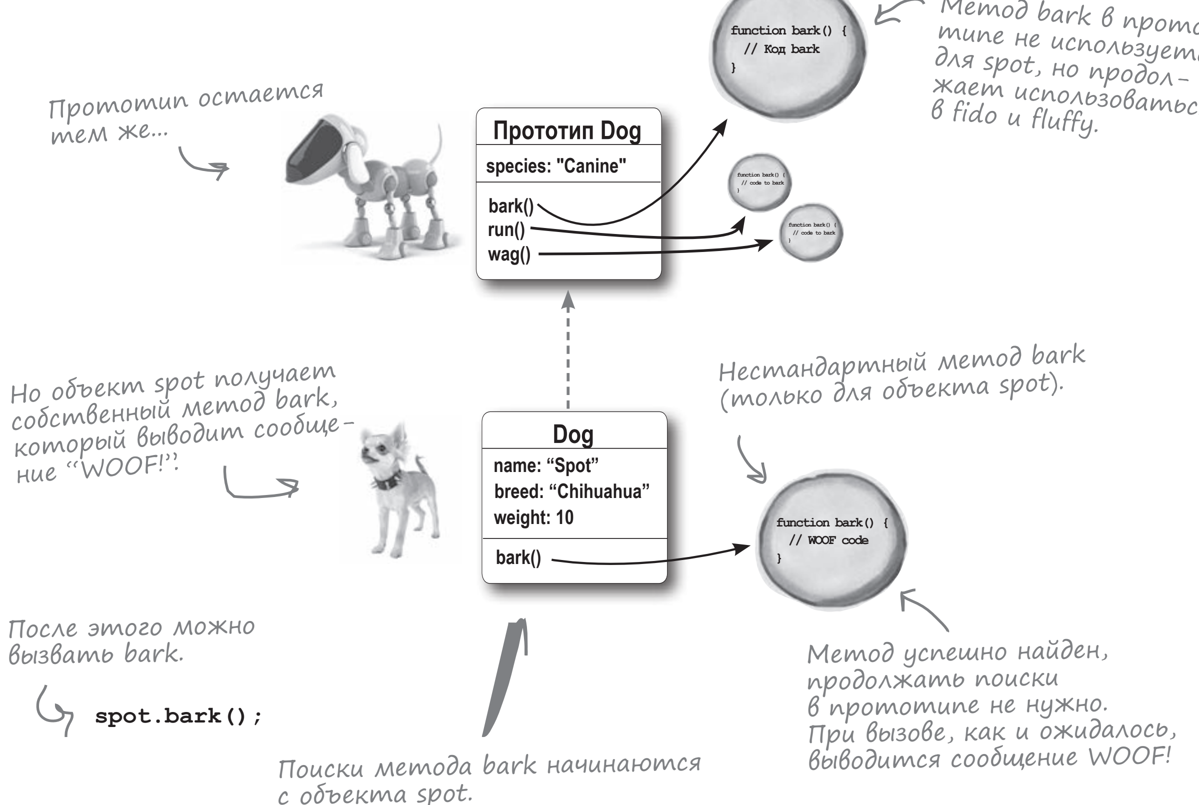
Переопределение прототипа

Мне не нравится такой метод bark. Я хочу лаять ГРОМЧЕ! Как насчет «WOOF!» большими буквами?!

Даже если ваш объект что-то наследует от прототипа, это не значит, что вам придется непременно пользоваться «наследством». Свойства и методы всегда можно *переопределить* в объекте. Переопределение работает, потому что JavaScript всегда обращается за свойством к экземпляру (то есть конкретному объекту собаки) *до того*, как продолжить поиски в прототипе. Следовательно, если вы хотите использовать для `spot` нестандартный метод `bark`, вам достаточно включить свою реализацию в объект `spot`. Когда JavaScript начнет искать метод `bark` для вызова, реализация будет найдена в объекте `spot`, и не будет возиться с поисками в прототипе.

Давайте посмотрим, что происходит при переопределении метода `bark` в объекте `spot`.

Спот

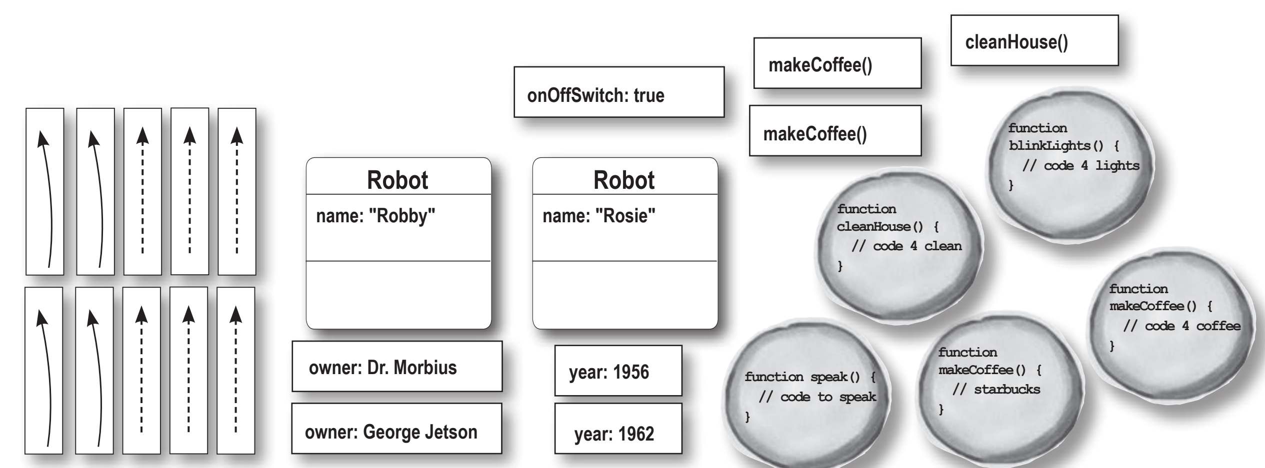


Развлечения с магнитами

На холодильнике была выложена диаграмма объектов, но кто-то все перепутал. Сможете ли вы расставить магниты по местам? На диаграмме будут присутствовать два экземпляра, созданных на базе прототипа. Первый — Робби (Robby) — создан в 1956 году, принадлежит доктору Морбиусу (Dr. Morbius), оснащен выключателем и умеет бегать в Starbucks за кофе. Другой робот — Рози (Rosie) — создан в 1962 году, прибирается в доме Джорджа Джетсона (George Jetson). Удачи! Кстати, внизу могут остаться лишние магниты...

Прототип, от которого наследуют ваши роботы.

Постройте здесь диаграмму объектов.



Как получить прототип

Вероятно, после всех разговоров о прототипах вы предпочтете увидеть пример, в котором используется код вместо диаграмм. Итак, как же создать или получить прототип? Вообще-то он у вас все время был, просто вы об этом не знали.

А вот как обратиться к нему в коде:

```
Dog.prototype
```

Присмотревшись к конструктору Dog, вы обнаружите в нем свойство prototype, в котором хранится ссылка на фактический прототип.

Если теперь использовать значение свойства `prototype`...

Ну и что, не обращайте внимания... Мы пошутили, конечно, вы правы. Мы пытались обойти острые углы (и будем делать это в будущем). В двух словах: в JavaScript функции являются объектами. Более того, в JavaScript практически все базируется на объектах — даже массивы, если вы еще не поняли..

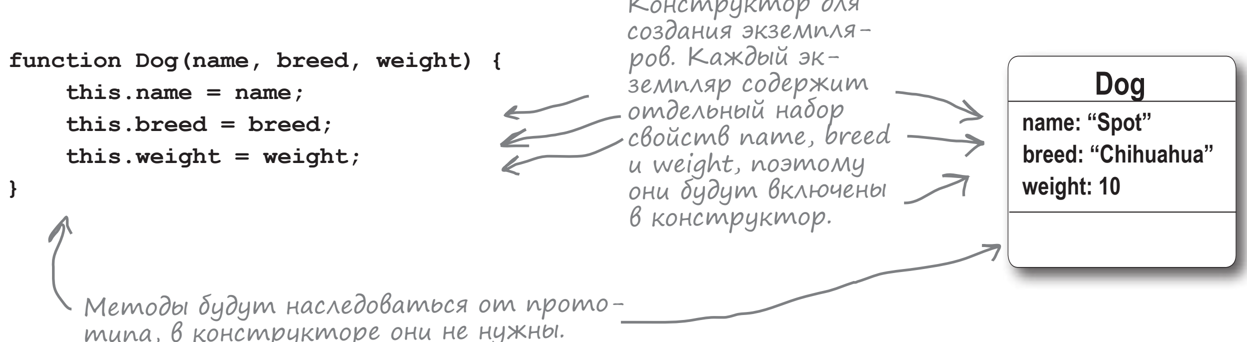
Но пока не будем отвлекаться. Просто знайте, что функции (помимо всего, что вы о них уже знаете) могут обладать свойствами, и в данном случае конструктор всегда содержит свойство prototype. Мы расскажем о функциях и других «замаскированных объектах» позднее в этой книге, честное слово.

Как создать прототип

Как мы уже говорили, объект-прототип задается в свойстве prototype конструктора Dog. Но какие еще свойства и методы включает прототип? Пока вы не зададите их самостоятельно — практически никаких. Другими словами, вам придется добавить свойства и методы в прототип самостоятельно. Обычно это делается до того, как мы начнем использовать конструктор.

Итак, давайте создадим прототип для объектов, представляющих собак. Сначала нам понадобится конструктор — чтобы понять, как он должен выглядеть, посмотрите на диаграмму объектов:

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
```



Конструктор у нас имеется, переходим к настройке прототипа. Мы хотим, чтобы прототип содержал свойства `species` и `bark`, а также методы `run` и `wag`. Вот как это делается:

```
Dog.prototype.species = "Canine";

Dog.prototype.bark = function() {
    if (this.weight > 25) {
        console.log(this.name + " says Woof!");
    } else {
        console.log(this.name + " says Yip!");
    }
};

Dog.prototype.run = function() {
    console.log("Run!");
};

Dog.prototype.wag = function() {
    console.log("Wag!");
};
```

Строка “Canine” задается свойству species прототипа.

Для каждого метода соответствующая функция назначается свойствам bark, run и wag прототипа.

Для любознательных

И не забудьте о сцеплении:

```
Dog.prototype.species
```

Начинаем с объекта Dog и получаем его свойство prototype, которое содержит ссылку на объект со свойством species.

Тест-драйв прототипа

Сохраните этот код в файл (dog.html) и загрузите его в браузере для тестирования. Мы приводим весь код с изменениями, внесенными на предыдущей странице, и добавляем тестовый код. Убедитесь в том, что ваши объекты ведут себя так, как положено.

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}

Dog.prototype.species = "Canine";

Dog.prototype.bark = function() {
    if (this.weight > 25) {
        console.log(this.name + " says Woof!");
    } else {
        console.log(this.name + " says Yip!");
    }
};

Dog.prototype.run = function() {
    console.log("Run!");
};

Dog.prototype.wag = function() {
    console.log("Wag!");
};

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);

fido.bark();
fido.run();
fido.wag();

fluffy.bark();
fluffy.run();
fluffy.wag();

spot.bark();
spot.run();
spot.wag();
```

Конструктор Dog.

А здесь в прототип добавляются свойства и методы.

Мы добавляем в прототип одно свойство и три метода.

Затем объекты создаются, как обычно...

...после чего мы вызываем методы каждого объекта, как обычно. Методы наследуются объектами от прототипа.

Один момент, но ведь для объекта spot нужна особая реализация bark?!

Все методы работают, хорошо.

Консоль JavaScript

```
Fido says Woof!
Run!
Wag!
Fluffy says Woof!
Run!
Wag!
Spot says Yip!
Run!
Wag!
```

Переопределение унаследованного метода

Не беспокойтесь, мы не забыли о Споте. Чтобы предоставить ему нестандартный метод bark, необходимо переопределить прототип. Внесите следующие изменения в код:

```
...
```

Далее следует остальной код. Мы опускаем его для экономии бумаги, битов в электронной версии, или чего-нибудь еще...

```
var spot = new Dog("Spot", "Chihuahua", 10);

spot.bark = function() {
    console.log(this.name + " says WOOF!");
};

// Вызовы для fido и fluffy выглядят так же

spot.bark();
spot.run();
spot.wag();
```

В коде вносится единственное изменение: объект spot получает собственную реализацию метода bark.

Вызов метода bark для объекта spot изменять вообще не придется.

Тест-драйв метода bark

Добавьте приведенный выше код и протестируйте его...

```
JavaScript console
Fido says Woof!
Run!
Wag!
Fluffy says Woof!
Run!
Wag!
Spot says WOOF!
Run!
Wag!
```

Желание Спота выполнено!

Упражнение

Помните нашу диаграмму объектов с роботами Robby и Rosie? Сейчас мы реализуем ее в программном коде. Мы уже написали за вас конструктор Robot, а также добавили тестовый код. Вам остается настроить прототип и реализовать двух роботов. Не забудьте проверить, что у вас получилось.

Базовый конструктор Robot. Ваша задача — настроить его прототип.

Прототип настраивается здесь.

Запишите здесь код для создания роботов. Не забудьте добавить в экземпляры все нестандартные свойства, которые в них должны присутствовать.

Используйте этот код для тестирования своих экземпляров — убедитесь в том, что они работают правильно и наследуют от прототипа.

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}
Robot.prototype.maker =

Robot.prototype.speak =

Robot.prototype.makeCoffee =

Robot.prototype.blinkLights =

var robby =
var rosie =

robby.onOffSwitch =
robby.makeCoffee =


rosie.cleanHouse =


console.log(robby.name + " was made by " + robby.maker +
            " in " + robby.year + " and is owned by " + robby.owner);
robby.makeCoffee();
robby.blinkLights();

console.log(rosie.name + " was made by " + rosie.maker +
            " in " + rosie.year + " and is owned by " + rosie.owner);
rosie.cleanHouse();
```

Не понимаю, почему this.name в методе bark все еще работает, ведь метод bark содержится в прототипе, а не в исходном объекте.

Хороший вопрос. Пока мы не использовали прототипы, все было просто — мы знали, что ссылка `this` указывает на объект, метод которого был вызван. Но при вызове метода `bark` в прототипе создается впечатление, что ссылка `this` должна указывать на объект прототипа. Тем не менее это не так.

Когда вы вызываете метод объекта, ссылке `this` присваивается объект, метод которого был вызван. Если метод в этом объекте отсутствует, но обнаруживается в прототипе, это не влияет на значение `this`. Ссылка `this` всегда ссылается на исходный объект — то есть объект, метод которого был вызван, даже если метод находится в прототипе. Итак, если метод `bark` находится в прототипе, то вызывается этот метод, а `this` ссылается на исходный объект.

Обучаем собак новым трюкам

Пришло время научить наших собак новым трюкам. Да, все верно — именно «собак», а не «собаку». Теперь, когда у нас имеется прототип, при добавлении в него каких-либо методов (даже после того, как мы уже создали объекты собак), все собаки, наследующие от прототипа, немедленно и автоматически получают новое поведение.

Предположим, мы хотим научить всех собак выполнять команду «сидеть». Для этого в прототип добавляется метод sit.

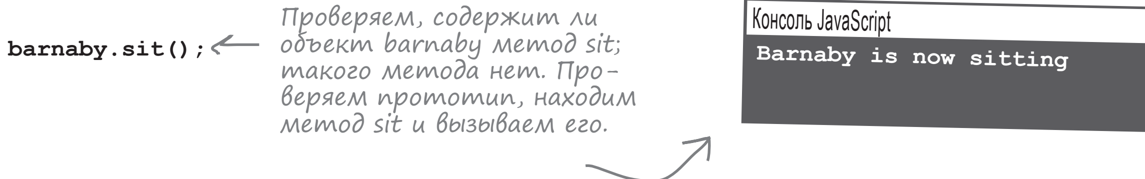
```
var barnaby = new Dog("Barnaby", "Basset Hound", 55);

Dog.prototype.sit = function() {
    console.log(this.name + " is now sitting");
}
```

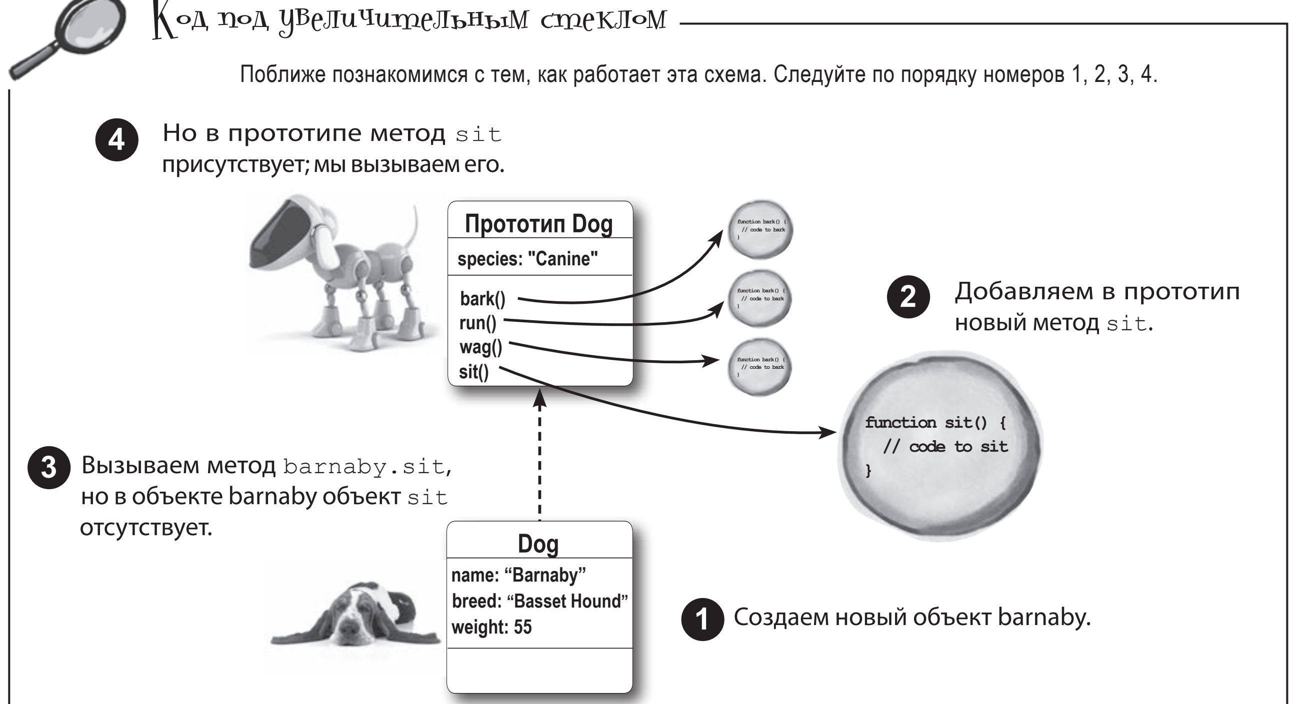
Создаем другой объект для тестирования.

И добавляем метод sit.

Опробуем новый метод на объекте barnaby:

```
barnaby.sit();
```

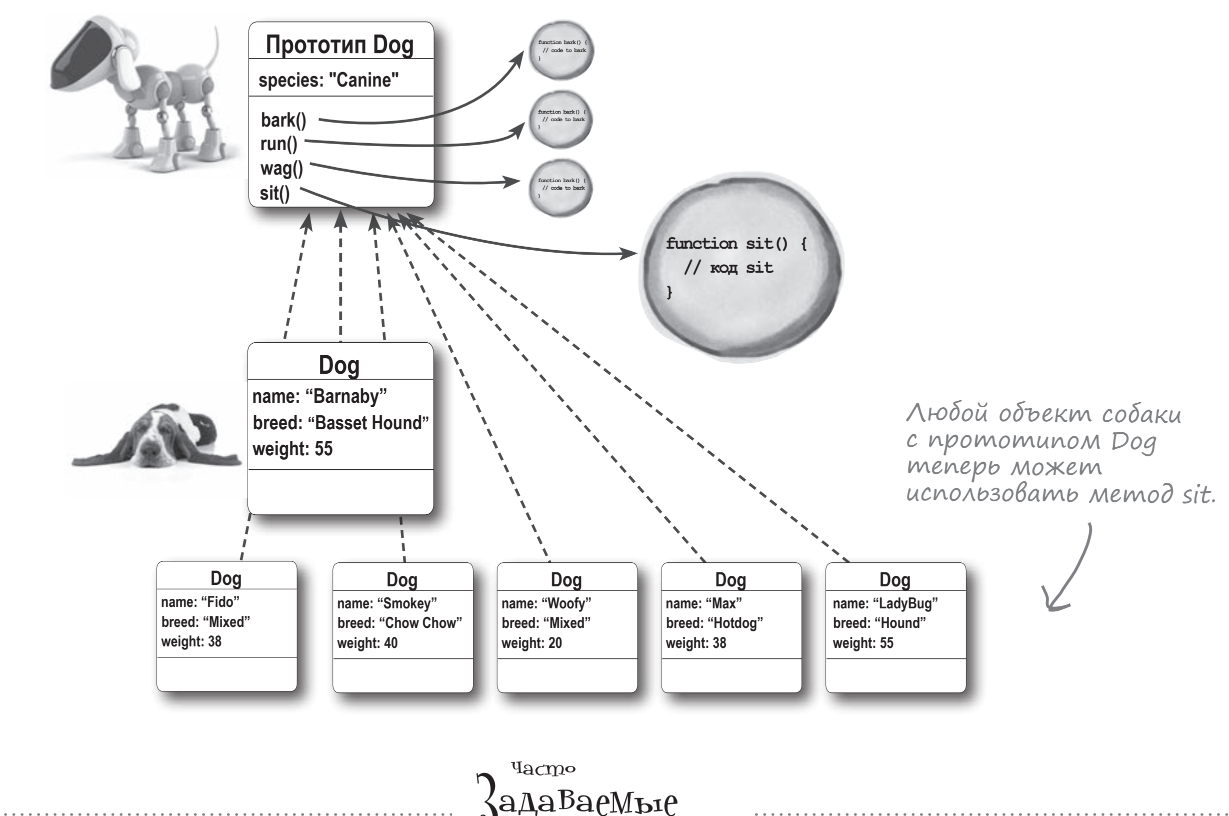




О динамических прототипах

Барнаби теперь выполняет команду «сидеть». Но оказывается, эту команду теперь выполняют *все* наши собаки, потому что после добавления метода в прототип он может использоваться всеми объектами, наследующими от этого прототипа:

Конечно, работает и для свойств.



Любой объект собаки с прототипом Dog теперь может использовать метод sit.

Часто задаваемые вопросы

В: Итак, когда я добавляю в прототип новый метод или свойство, они немедленно проявляются во всех экземплярах объектов, наследующих от этого прототипа?

О: Если под «проявляются» вы имеете в виду, что экземпляры немедленно наследуют этот метод или свойство, тогда все верно. Таким образом, вы можете расширять или изменять поведение всех экземпляров во время выполнения, просто изменяя их прототип.

В: Я понимаю, как добавление нового свойства в прототип делает свойство доступным для объектов, наследующих от прототипа. Но если я *изменяю* существующее свойство в прототипе, отразятся ли эти изменения на объектах? Скажем, если я меняю значение свойства species с “Canine” на “Feline”, означает ли это, что у всех существующих объектов это свойство тоже будет содержать “Feline”?

О: Да. Если вы изменяете свойство в прототипе, такое изменение отражается на всех объектах, наследующих от прототипа, если только объект не переопределил это свойство.

Наши роботы используются в компьютерной игре, код которой приведен ниже. В этой игре при достижении игроком уровня 42 у робота включается новая способность: лазерный луч. Допишите приведенный ниже код, чтобы на уровне 42 лазерный луч включался у обоих роботов, Робби и Рози. Сверьтесь с ответами в конце главы, прежде чем двигаться дальше.

```
function Game() {
    this.level = 0;
}

Game.prototype.play = function() {
    // Действия игрока
    this.level++;
    console.log("Welcome to level " + this.level);
    this.unlock();
}

Game.prototype.unlock = function() {






}

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var game = new Game();
var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

while (game.level < 42) {
     game.play();
}

robby.deployLaser();
rosie.deployLaser();
```

Консоль JavaScript

```
Welcome to level 1
Welcome to level 2
Welcome to level 3
 ...
Welcome to level 41
Welcome to level 42
Rosie is blasting you with
laser beams.
```

Пример вывода. Завершив работу над кодом, запустите его и посмотрите, какой робот победит!

Более интересная реализация метода sit

Давайте сделаем метод `sit` чуть более интересным: в исходном состоянии собака стоит. Если при получении команды «сидеть» собака стоит, она садится. В противном случае выводится сообщение о том, что собака уже сидит. Для этого нам понадобится дополнительное свойство `sitting`, в котором будет храниться информация о том, сидит собака или нет. Код будет выглядеть так:

Начинаем со свойства sitting в прототипе.

Задаем sitting в прототипе значение false — изначально все собаки стоят.

```
Dog.prototype.sitting = false;

Dog.prototype.sit = function() {
    if (this.sitting) {
        console.log(this.name + " is already sitting");
    } else {
        this.sitting = true;
        console.log(this.name + " is now sitting");
    }
};
```

Затем в методе sit проверяем, сидит собака или нет. При проверке this.sitting сначала проверяется значение в прототипе.

Если собака сидит, выводим соответствующее сообщение.

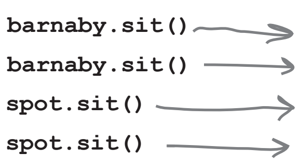
Если собака стоит, мы задаем this.sitting значение true. При этом происходит переопределение прототипа, а значение задается в экземпляре.

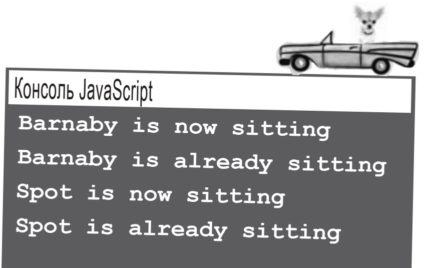
Обратите внимание: экземпляр теперь имеет собственное локальное свойство sitting, равное true.

В этом коде прежде всего стоит обратить внимание на то, что в начале своего существования экземпляр наследует значение по умолчанию для `sitting`. Но при вызове метода `sit` экземпляр добавляет собственное значение `sitting`, в результате чего новое свойство создается в экземпляре. Оно переопределяет унаследованное свойство `sitting` из прототипа. Таким образом, мы можем определить значение по умолчанию для всех объектов, а затем, если понадобится, создать его специализированную версию в каждом конкретном объекте.

Тест-драйв нового метода sit

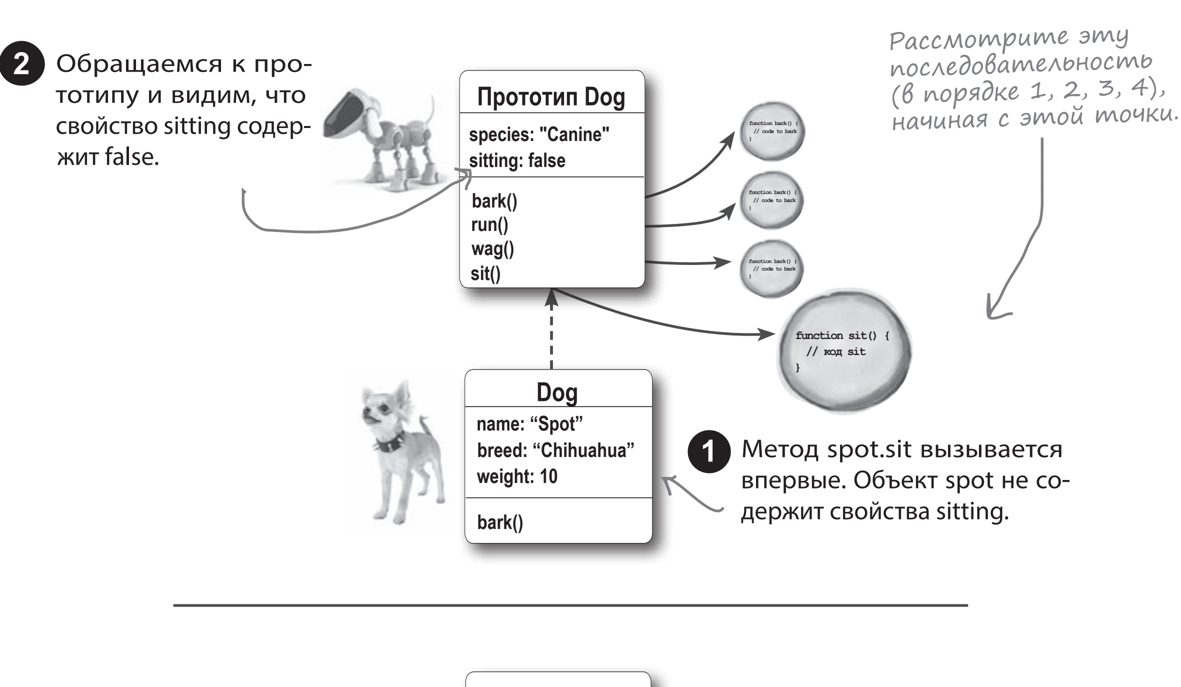
Проверим, как работает код. Внесите изменения, добавьте новое свойство и реализацию `sit`. Теперь при тестировании становится видно, что сначала «садится» объект `barnaby`, а затем объект `spot`, причем для каждой собаки проверка того, сидит она или нет, осуществляется отдельно:

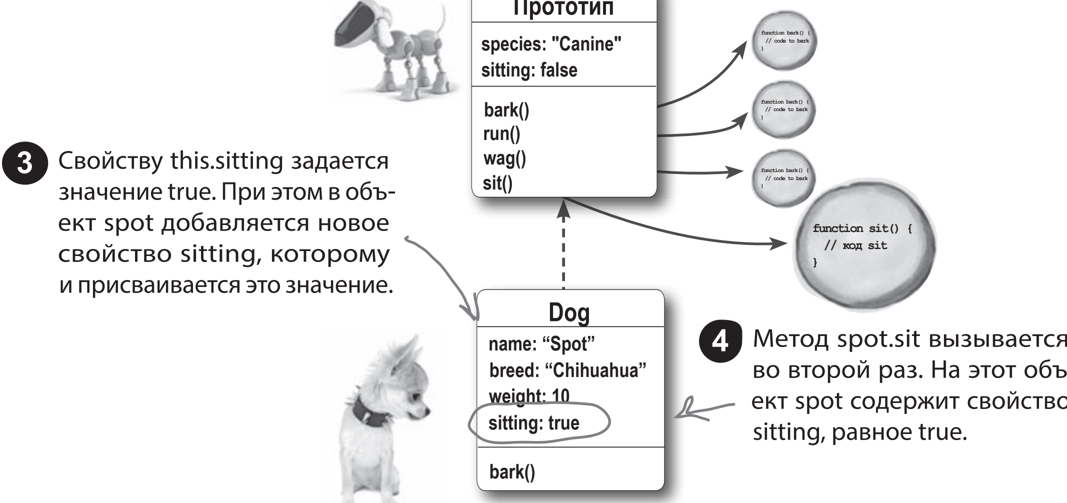




Еще раз: как работает свойство sitting

Давайте убедимся в том, что вы все поняли — поторопившись с изучением этой реализации, можно упустить очень важные подробности. Итак, при первом обращении к свойству `sitting` его значение берется из прототипа. Но затем, когда `sitting` присваивается значение true, это происходит уже не в прототипе, а в экземпляре. И после того как свойство будет добавлено в экземпляр объекта, при всех последующих обращениях к `sitting` значение будет браться из экземпляра, потому что оно переопределяет значение из прототипа. Давайте рассмотрим происходящее еще один раз:





Раз уж мы заговорили о свойствах, можно ли в коде определить, откуда взято используемое значение свойства — из экземпляра или прототипа?

Да, можно. Воспользуйтесь методом `hasOwnProperty`, который имеется у каждого объекта. Метод `hasOwnProperty` возвращает true, если свойство определяется в экземпляре объекта. Если возвращается false, но вы можете обратиться к этому свойству, значит, свойство определено в прототипе.

Рассмотрим пример использования `hasOwnProperty` с объектами `fido` и `spot`. Во-первых, мы знаем, что свойство species реализовано только в прототипе Dog, и ни `spot`, ни `fido` это свойство не переопределяли. Итак, при вызове метода `hasOwnProperty` с передачей имени свойства "species" (в виде строки) в обоих случаях возвращается false:

```
spot.hasOwnProperty("species");
fido.hasOwnProperty("species");
```

В обоих случаях возвращается false, потому что свойство species определяется в прототипе, а не в экземплярах spot и fido.

Попробуем проделать то же самое для свойства `sitting`. Мы знаем, что свойство `sitting` определяется в прототипе и инициализируется значением false. Итак, мы присваиваем `spot.sitting` значение true, что приводит к переопределению свойства `sitting` в прототипе, с определением sitting в экземпляре `spot`. Затем мы запрашиваем у `spot` и `fido`, определяют ли они собственное свойство `sitting`:

Когда мы в первый раз проверяем, содержит ли spot собственное свойство sitting, возвращается значение false.

```
spot.hasOwnProperty("sitting");
spot.sitting = true;
spot.hasOwnProperty("sitting");
fido.hasOwnProperty("sitting");
```

Затем мы задаем spot.sitting значение true, добавляя это свойство в экземпляр spot.

Этот вызов hasOwnProperty возвращает true, потому что spot теперь содержит собственное свойство sitting.

А этот вызов hasOwnProperty возвращает false, потому что экземпляр fido не содержит свойства sitting. Это означает, что свойство sitting, используемое fido, определяется только в прототипе и наследуется fido.

Мы встроили в наших роботов Робби и Рози новую функцию: теперь они могут сообщать о возникающих ошибках при помощи метода reportError. Проанализируйте приведенный ниже код, обращая особое внимание на то, откуда этот метод получает информацию об ошибках и откуда берется эта информация: из прототипа или экземпляра.

Запишите внизу результат выполнения кода:

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.errorMessage = "All systems go.";
Robot.prototype.reportError = function() {
    console.log(this.name + " says " + this.errorMessage);
};
Robot.prototype.spillWater = function() {
    this.errorMessage = "I appear to have a short circuit!";
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

rosie.reportError();
robby.reportError();
robby.spillWater();
rosie.reportError();
robby.reportError();

console.log(robby.hasOwnProperty("errorMessage")); ____________
console.log(rosie.hasOwnProperty("errorMessage")); ____________
```

Содержит ли Робби собственное свойство errorMessage?

А Рози?

Чемпион породы

Наши труды в этой главе окупились. Директор Клуба любителей собак Объектвиля увидел вашу работу над объектами собак и немедленно понял, что нашел правильных людей для моделирования собачьей выставки. От вас потребуется совсем немного: обновить конструктор Dog для создания объектов выставочных собак. Ведь, как известно, выставочные собаки отличаются от обычных – они не носятся сломя голову, а демонстрируют экстерьер. Они не выпрашивают сахар, а выражают желание полакомиться.

А если говорить конкретнее, нужно следующее:

Прекрасная работа с конструктором Dog! Нам хотелось бы привлечь вас к работе над моделированием собачьей выставки. На выставках к собакам предъявляются особые требования, поэтому нам понадобятся дополнительные методы (см. ниже).

Спасибо!

Клуб любителей собак Объектвиля

stack() — команда «стоять».

gait() — разные варианты движения. Метод получает строковый аргумент: "walk", "trot", "pace" или "gallop".

bait() — дать собаке лакомство.

groom() — купание и уход.

С чего начать проектирование объектов

Как мы собираемся спроектировать эту систему? Разумеется, нужно использовать существующий код — в конце концов, собаководы к нам из-за этого и обратились. Но как? Есть несколько возможных подходов:

Если мы добавим новые методы в существующий конструктор Dog, то все собаки смогут выполнять эти методы. Но изменять то, что работает, не хочется.

Можно добавить новые методы собак в экземпляры выставочных собак, но тогда мы возвращаемся к проблемам, описанным в начале главы.

А если мы напишем конструктор выставочной собаки с нуля, нам придется заново реализовать все базовые методы: bark, run, sit...

Джо: Несколько прототипов? Что это значит?

Джуди: Считайте это чем-то вроде собственной реализации наследования.

Джо: Какого наследования? Если бы я унаследовал что-нибудь серьезное, то не работал бы здесь! Шучу...

Джуди: Ты же наследуешь признаки не только от родителей, верно? В тебе есть что-то от бабушки с дедушкой, от прабабушек и прадедушек и т. д.

Джо: Да, я понимаю.

Джуди: Так вот, в JavaScript можно создать цепочку прототипов, от которой наследуют признаки ваши объекты.

Фрэнк: Хорошо бы увидеть пример.

Джуди: Допустим, имеется прототип птицы, который умеет делать то, что умеют все птицы, — скажем, летать.

Фрэнк: Легко, как с нашим прототипом собак.

Джуди: Теперь допустим, вы хотите реализовать семейство уток — кряквы, нырки...

Фрэнк: ...И не забудь древесных черноклювых уток.

Джуди: Спасибо за помощь, Фрэнк.

Фрэнк: Не за что. Я о них как раз прочитал в книге *Паттерны проектирования.*

Джуди: Хорошо, но утки — просто особая разновидность птиц. Они умеют плавать, а нам не хотелось бы включать это поведение в прототип птицы. Но на JavaScript мы можем создать прототип утки, наследующий от прототипа.

Джо: Я правильно понял? У нас есть конструктор Duck, который ссылается на прототип утки. Но этот прототип сам содержит ссылку на прототип птицы?

Фрэнк: Стоп, не гони лошадей... В смысле уток.

Джуди: Посмотри на это так, Фрэнк: допустим, ты создаешь утку и вызываешь ее метод `fly`. Что произойдет, если ты заглянешь в объект и не найдешь такого метода? Заглядываешь в прототип, от которого наследует утка, — это птица. Там и находится метод `fly`.

Джо: А если вызвать метод `swim`, мы обращаемся к экземпляру утки — ничего. Переходим к прототипу утки — и он там обнаруживается.

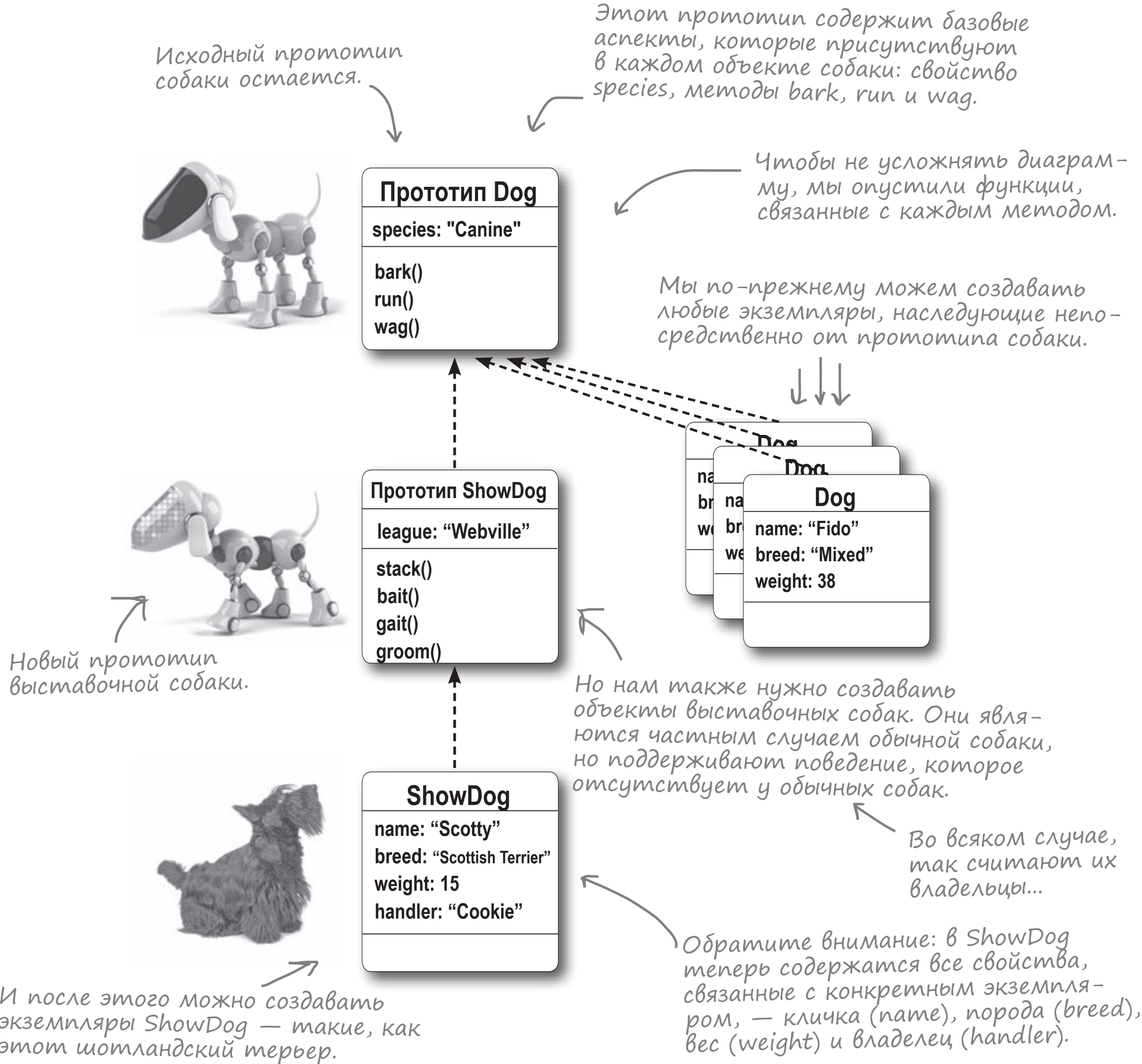
Джуди: Верно... Так что мы не просто используем поведение прототипа утки, а при необходимости доходим по цепочке до прототипа птицы.

Джо: Да, идеально подходит для расширения нашего прототипа собаки в прототип выставочной собаки. Посмотрим, что из этого выйдет.

Создание цепочки прототипов

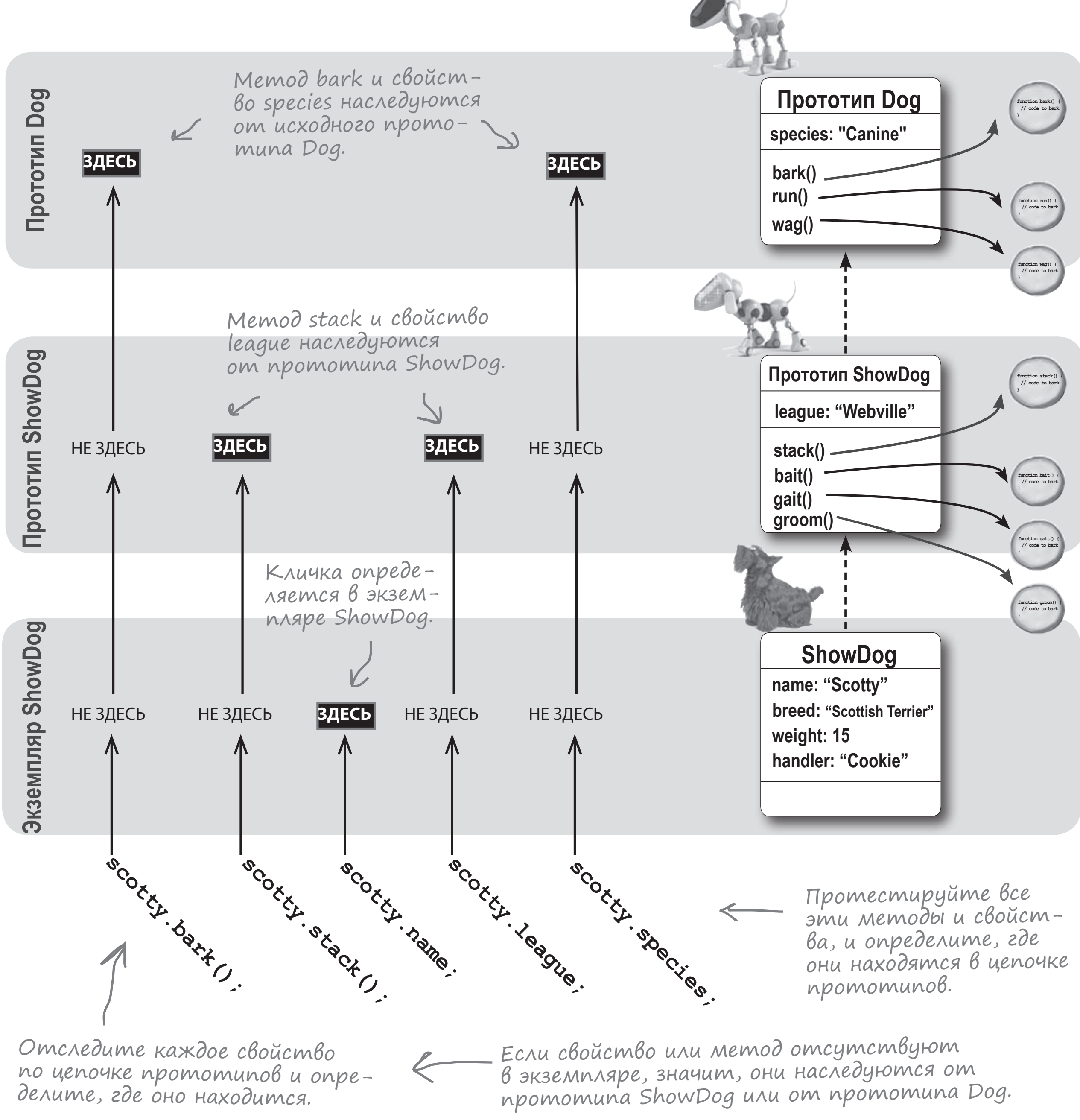
Взглянем на ситуацию с точки зрения *цепочки прототипов*. Вместо того чтобы создавать экземпляр, наследующий свойства от одного прототипа, мы можем создать цепочку из одного или нескольких прототипов, от которых наследуют ваши экземпляры. В целом это вполне логичное развитие подхода, который мы использовали ранее.

Допустим, мы хотим создать прототип выставочной собаки, а методы `bark`, `run` и `wag` этот прототип должен получать из исходного прототипа собаки. Создадим эту цепочку и посмотрим, как работает эта схема:



Как работает наследование в цепочке прототипов

Мы создали цепочку прототипов для выставочных собак; посмотрим, как наследование работает в этом контексте. Просмотрите список свойств и методов в нижней части страницы и свяжите их по цепочке прототипов с объектом, в котором они определяются.



Развлечения с магнитами

Мы выложили на холодильнике очередную диаграмму объектов, и снова кто-то все перепутал. Снова! Сможете ли вы расставить магниты по местам? Нам понадобится новая линейка космических роботов, наследующих свойства от обычных роботов. Космические роботы переопределяют метод speak обычных роботов, а также расширяют функциональность роботов новым свойством homePlanet. Удачи! (Внизу могут остаться лишние магниты!)

Прототип роботов.

Постройте здесь диаграмму объектов.

Прототип Robot
maker: "ObjectsRUs"
speak()
makeCoffee()
blinkLights()

```
function blinklights() {
  // code 4 lights
}
```

Robot
name: "Robby"
year: 1956
owner: "Dr. Morbius"

Robot
name: "Rosie"
year: 1962
owner: "George Jetson"

Прототип космических роботов.

Прототип SpaceRobot

speak()

year: 1977

pilot()

year: 2009

homePlanet: "Earth"

homePlanet: "Tatooine"

```
function speak() {
  // code to speak
  // in space
}
```

```
function
pilot() {
  // code to pilot
  // spaceship
}
```

SpaceRobot
name: "C3PO"
year: 1977
owner: "L. Skywalker"

SpaceRobot
name: "Simon"
year: 2009
owner: "Carla Diana"

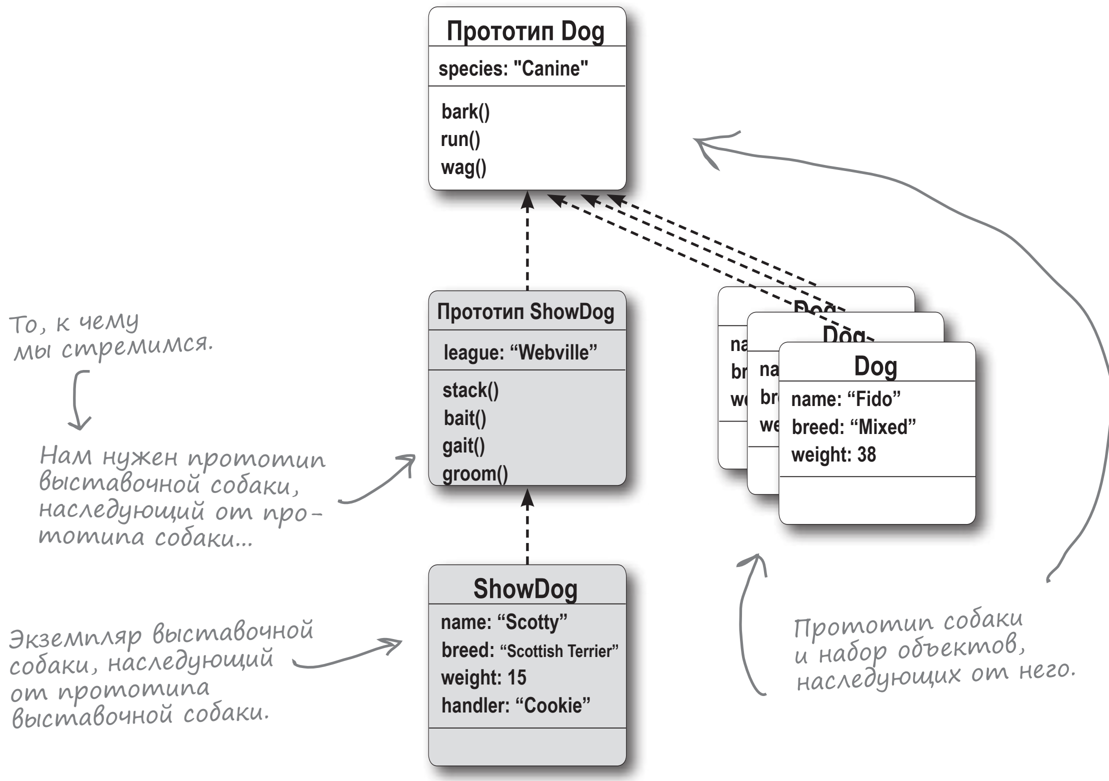
Создание прототипа выставочной собаки

При создании прототипа собаки нам ничего специально делать не приходилось — у нас уже был пустой объект, предоставляемый свойством `prototype` конструктора Dog. Мы взяли его и добавили свойства и методы, которые должны были наследоваться экземплярами собак.

Но с прототипом выставочных собак придется дополнительно потрудиться, потому что нам нужен прототип, наследующий от другого прототипа (прототипа собаки). Для этого мы должны создать объект, наследующий от прототипа собаки, а затем явно создать необходимые связи.

Прямо сейчас у нас имеется прототип собаки и набор экземпляров, наследующих от него. А нужен нам прототип выставочной собаки (наследующий от прототипа собаки) и набор экземпляров, наследующих от прототипа выставочной собаки.

Настройка выполняется за несколько шагов, которые мы сейчас последовательно рассмотрим.



Для начала нам понадобится объект, наследующий от прототипа собаки

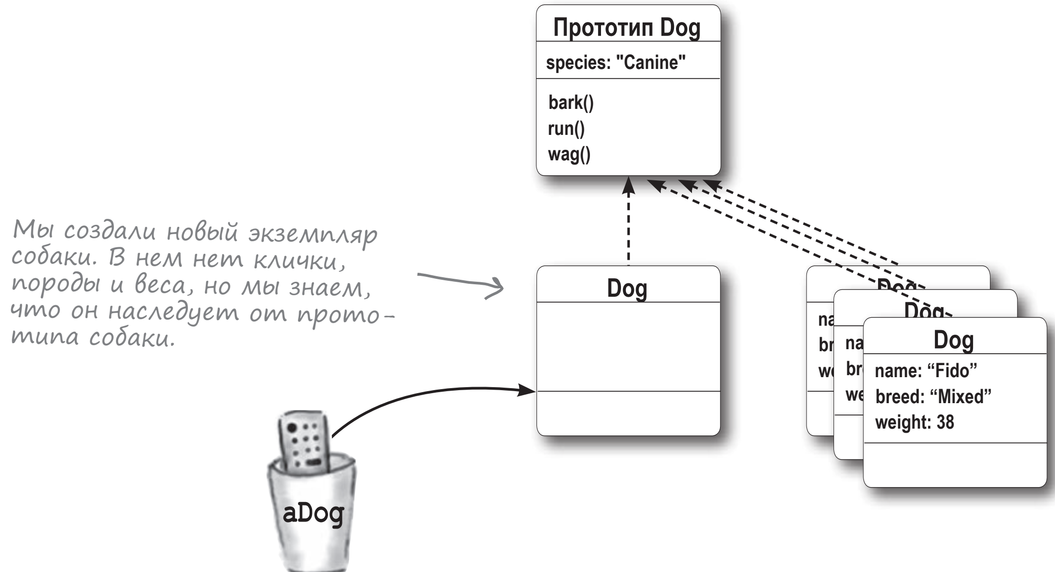
Мы установили, что прототип выставочной собаки — объект, наследующий от прототипа собаки. А как лучше всего создать объект, наследующий от прототипа собаки? Вообще-то мы уже неоднократно делали это при создании экземпляров собак. Помните? Вот так:

Чтобы создать объект, наследующий от прототипа собаки, мы используем оператор new с конструктором Dog.

```
var aDog = new Dog();
```

О том, что случилось с аргументами конструктора, мы сейчас расскажем...

Этот код создает объект, наследующий от прототипа собаки. Мы знаем это, потому что точно так же ранее создавались все экземпляры собак — просто на этот раз конструктору не передаются аргументы. Дело в том, что сейчас подробности инициализации для нас не важны; нужен только объект собаки, наследующий от прототипа собаки.



Теперь нам понадобится прототип выставочной собаки. Как и экземпляр собаки, это всего лишь объект, наследующий от прототипа собаки. Давайте посмотрим, как использовать пустой экземпляр собаки для создания нужного нам прототипа выставочной собаки.

Затем экземпляр собаки преобразуется в прототип выставочной собаки

Итак, у нас есть экземпляр собаки, но как превратить его в объект прототипа выставочной собаки? Для этого экземпляр собаки назначается свойству `prototype` конструктора ShowDog. Хотя постойте, у нас еще нет конструктора ShowDog... так давайте напишем его:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
```

Конструктор получает все необходимое для создания собаки (name, breed, weight) и выставочной собаки (handler).

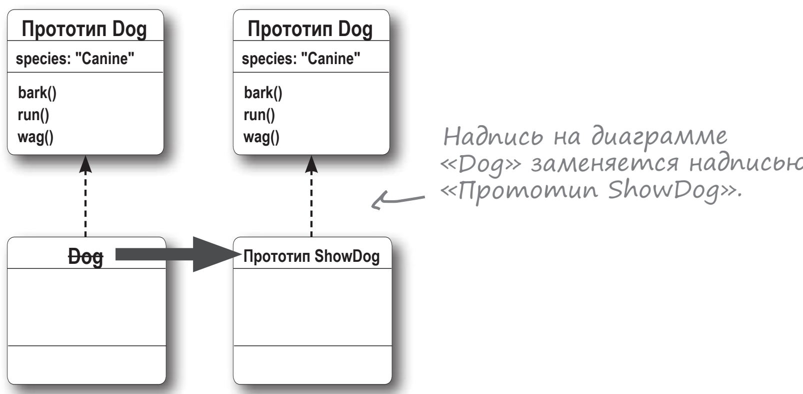
Теперь, когда у нас имеется конструктор, мы можем задать его свойству `prototype` новый экземпляр собаки:

```
ShowDog.prototype = new Dog();
```

Мы могли бы воспользоваться экземпляром собаки, созданным на предыдущей странице, но вместо этого можно пропустить присваивание и напрямую назначить новый объект свойству prototype.

Итак, подумаем, что же здесь произошло: у нас имеется конструктор ShowDog, который может использоваться для создания экземпляров выставочных собак, и прототип выставочной собаки, который представляет собой экземпляр собаки.

Чтобы диаграмма объектов более точно отражала роли этих объектов, мы заменим надпись «Dog» на «Прототип ShowDog». Но помните, что прототип выставочной собаки *все равно остается экземпляром собаки*.



Итак, у нас имеется конструктор ShowDog, и мы подготовили объект прототипа выставочной собаки; теперь нужно вернуться назад и заполнить некоторые подробности. Сейчас мы более подробно рассмотрим конструктор, а также добавим в прототип некоторые свойства и методы, чтобы выставочные собаки обладали дополнительным поведением.

Пришло время заполнить прототип

Мы создали прототип выставочной собаки. На данный момент он представляет собой простой экземпляр собаки. А сейчас нужно дополнить его свойствами и поведением, с которыми он будет больше похож на прототип выставочной собаки.

Некоторые свойства и методы, специфические для выставочных собак:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}

ShowDog.prototype = new Dog();

Showdog.prototype.league = "Webville";

ShowDog.prototype.stack = function() {
      console.log("Stack");
};

ShowDog.prototype.bait = function() {
      console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
      console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
      console.log("Groom");
};
```

Конструктор ShowDog имеет много общего с конструктором Dog. Выставочной собаке нужна кличка (name), порода (breed), вес (weight), а также дополнительное свойство для владельца собаки (handler). Эти свойства будут определяться в экземпляре выставочной собаки.

Лига, к которой относятся все наши выставочные собаки. Добавляем соответствующее свойство в прототип.

Методы, которые должны поддерживаться всеми выставочными собаками. Пока не будем усложнять их.

Добавляем все эти свойства в прототип выставочной собаки, чтобы они наследовались всеми объектами выставочных собак.

Здесь мы берем экземпляр собаки, который становится прототипом выставочной собаки, и добавляем новые свойства и методы.

С этими дополнениями прототип выставочной собаки действительно начинает выглядеть так, как требуется. Давайте еще раз обновим нашу диаграмму объектов — и наверное, можно переходить к основательному тестированию. Хочется надеяться, что Клубу любителей собак понравится результат нашей работы.

Прототип Dog
- species: "Canine"
- bark()
- run()
- wag()

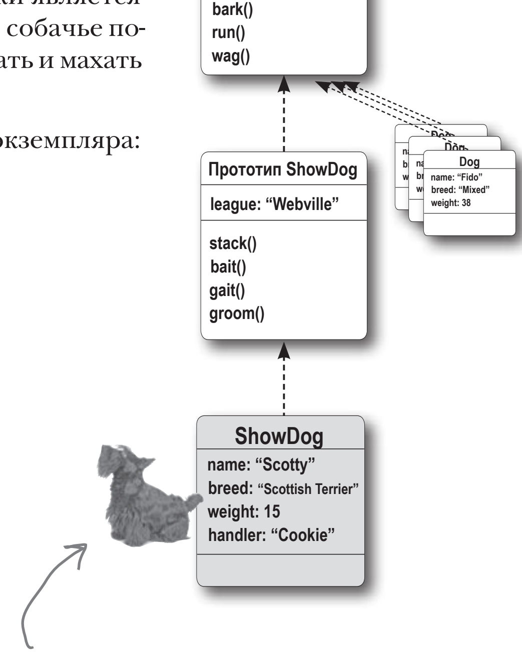
Прототип ShowDog
- league: "Webville"
- stack()
- bait()
- gait()
- groom()

Мы говорим, что прототип выставочной собаки «расширяет» прототип собаки. Он наследует свойства из прототипа собаки и дополняет их новыми свойствами.

Создание экземпляра выставочной собаки

Осталось сделать последний шаг: создать экземпляр ShowDog. Этот экземпляр будет наследовать свойства и методы выставочной собаки от прототипа выставочной собаки, а поскольку прототип выставочной собаки является экземпляром Dog, выставочная собака унаследует все основное собачье поведение и свойства от прототипа, поэтому она сможет лаять, бегать и махать хвостом вместе с остальными собаками.

Ниже приведен весь код реализации вместе с кодом создания экземпляра:




```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}

ShowDog.prototype = new Dog();

Showdog.prototype.league = "Webville";

ShowDog.prototype.stack = function() {
       console.log("Stack");
};

ShowDog.prototype.bait = function() {
       console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
       console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
       console.log("Groom");
};

var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```

А это наш экземпляр выставочной собаки. Он наследует от прототипа выставочной собаки, который в свою очередь наследует от прототипа собаки... Собственно, как мы и планировали. Вернувшись к с. 592, вы увидите, что цепочка прототипов завершена.

Новая выставочная собака scotty.

Тест-драйв для выставочных собак

Возьмите весь код с предыдущей страницы и добавьте его в тестовый код, приведенный внизу, чтобы хорошенько протестировать объект scotty. А заодно добавьте несколько собственных собак и протестируйте их:

```
scotty.stack();
scotty.bark();
console.log(scotty.league);
console.log(scotty.species);
```

Вот что получилось у нас.

Консоль JavaScript

```
Stack
Scotty says Yip!
Webville
Canine
```

Упражнение

Ваш ход. Добавьте в линейку выпускаемых роботов разновидность космических роботов SpaceRobot. Конечно, эти роботы должны уметь делать все, что делают обычные роботы, а также обладать дополнительным поведением. Мы уже начали писать код, допишите и протестируйте его. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

```
function SpaceRobot(name, year, owner, homePlanet) {



}

SpaceRobot.prototype = new ____________;

 ___________________.speak = function() {
    alert(this.name + " says Sir, If I may venture an opinion...");
};

 ___________________.pilot = function() {
    alert(this.name + " says Thrusters? Are they important?");
};

var c3po = new SpaceRobot("C3PO", 1977, "Luke Skywalker", "Tatooine");
c3po.speak();
c3po.pilot();
console.log(c3po.name + " was made by " + c3po.maker);

var simon = new SpaceRobot("Simon", 2009, "Carla Diana", "Earth");
simon.makeCoffee();
simon.blinkLights();
simon.speak();
```

Упражнение

Давайте поближе рассмотрим всех этих собак, которых мы создаем. Мы уже тестировали объект fido прежде, и знаем, что это действительно собака. Но давайте проверим, является ли он также и выставочной собакой (чего быть не должно). И как насчет scotty? Понятно, что он должен быть выставочной собакой, но будет ли он также обычной собакой? Не уверены. А заодно проверим конструкторы объектов fido и scotty...

```
var fido = new Dog("Fido", "Mixed", 38);
if (fido instanceof Dog) {
    console.log("Fido is a Dog");
}
if (fido instanceof ShowDog) {
    console.log("Fido is a ShowDog");
}

var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
if (scotty instanceof Dog) {
    console.log("Scotty is a Dog");
}
if (scotty instanceof ShowDog) {
    console.log("Scotty is a ShowDog");
}
console.log("Fido constructor is " + fido.constructor);
console.log("Scotty constructor is " + scotty.constructor);
```

Выполните этот код и запишите внизу полученный результат.

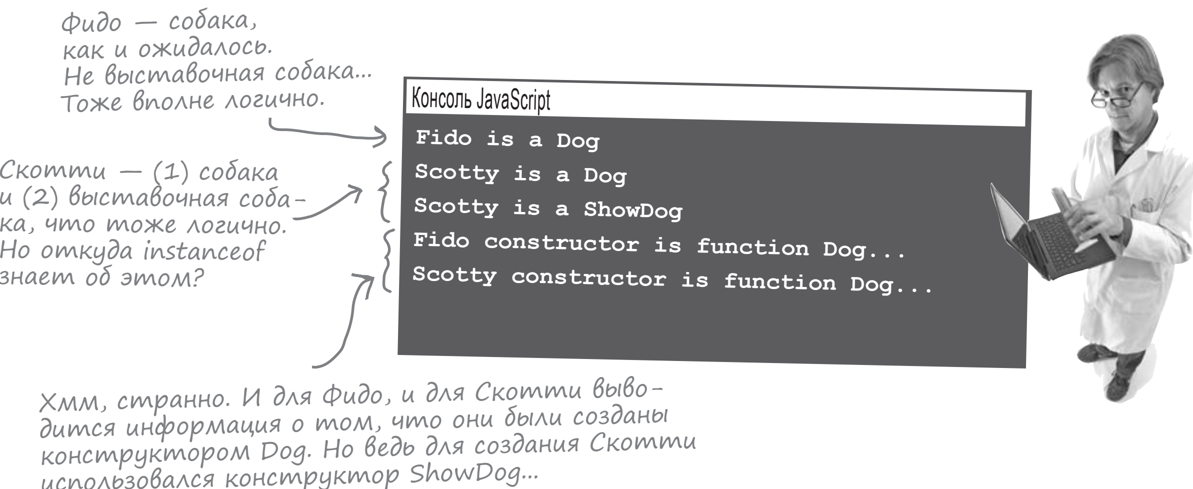
Консоль JavaScript

Здесь записывается ваш результат:

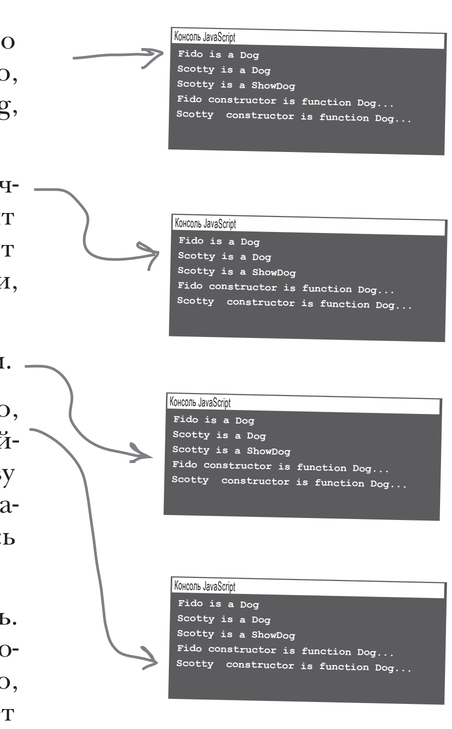
Наш результат приведен на следующей странице.

Анализ результатов

Результат, который мы получили при тестовом запуске:



Давайте немного обдумаем результаты. Во-первых, Фидо очевидно является обычной собакой, но не выставочной — как, собственно, и предполагалось; ведь объект fido создавался конструктором Dog, который к выставочным собакам не имеет никакого отношения.



Скотти является собакой *и* выставочной собакой. И это тоже логично, но как так вышло? Дело в том, что **instanceof** не только смотрит на разновидность самого объекта, но и учитывает все объекты, от которых он наследует. Раз выставочная собака наследует от собаки, Скотти также является собакой.

Фидо имеет конструктор Dog — да, понятно, мы так его и создавали.

Скотти тоже имеет конструктор Dog. И вот это уже непонятно, потому что он создавался конструктором ShowDog. Подумайте, откуда взялся этот конструктор: мы обращаемся к свойству `scotty.constructor`, которое в программе не задавалось. Следовательно, оно должно наследоваться от прототипа (раз не задавалось явно для выставочной собаки).

Откровенно говоря, это недоработка, которую нужно устранить. Видите ли, если не позаботиться о задании свойства constructor прототипа выставочной собаки, никто за нас это не сделает. Конечно, и без этого все работает нормально, но незаданное свойство может вызвать недоразумения, если вы или кто-то другой обратится к `scotty.constructor`, ожидая получить конструктор SnowDog.

Не беспокойтесь, сейчас все исправим.

Наводим чистоту

Готовый код уже можно отправлять в Клуб любителей собак, но сначала нужно исправить пару последних мелких недостатков.

Первый мы уже видели: у экземпляров ShowDog неправильно задается свойство constructor — они наследуют конструктор Dog. Чтобы не было неясностей: весь код нормально работает и в этом варианте, но назначение правильного конструктора в объектах может пригодиться, когда другой разработчик возьмется за ваш код, начнет изучать объект выставочной собаки — и сильно удивится.

Чтобы исправить свойство constructor, нужно позаботиться о том, чтобы он правильно задавался в прототипе ShowDog. В этом случае при конструировании объект выставочной собаки унаследует правильное свойство constructor. Вот как это делается:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
ShowDog.prototype = new Dog();
ShowDog.prototype.constructor = ShowDog;
```

Мы берем прототип выставочной собаки и явно назначаем его свойству constructor конструктор ShowDog.

Помните, что это скорее вопрос стиля; без этого изменения код тоже работает.

Все, что нужно сделать. При последующей проверке Скотти содержит правильное значение свойства constructor, как и все остальные выставочные собаки.

А для прототипа собаки это делать не нужно, потому что у него свойство constructor правильно задается по умолчанию.

Упражнение

Повторно запустите тесты из предыдущего упражнения и убедитесь в том, что выставочная собака Скотти теперь имеет правильный конструктор.

То, что получилось у нас. Обратите внимание: для Скотти теперь выводится конструктор ShowDog.

Консоль JavaScript

```
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function ShowDog...
```

Еще немного усилий

Осталось еще одно место, в котором нужно навести порядок: код конструктора ShowDog. Еще раз посмотрите на него:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
```

Если вдруг вы не заметили — этот код скопирован из конструктора Dog.

Каждый раз, когда вы видите дублирование кода, у вас внутри должен звучать сигнал тревоги. В данном случае конструктор Dog уже умеет выполнять эту часть работы, так почему бы не позволить конструктору выполнить ее? Кроме того, в нашем примере используется простой код, но нередко конструкторы содержат сложный код вычисления исходных значений свойств, и нам не хотелось бы дублировать код каждый раз, когда мы создаем новый конструктор, наследующий от другого прототипа. Давайте исправим этот недостаток. Сначала мы перепишем код, а потом подробно разберем его:

Идея устранения дублирующегося кода даже обозначается специальным сокращением: DRY (Don't Repeat Yourself, «Не повторяйтесь!»). Его знают все крутые программисты.

```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}
```

Этот фрагмент использует конструктор Dog для обработки свойств name, breed и weight.

Но свойство handler должно обрабатываться в этом коде, потому что конструктор Dog о нем ничего не знает.

Как видите, избыточный код в конструкторе ShowDog заменяется вызовом метода `Dog.call`. Вот как работает это решение: `call` — встроенный метод, который может использоваться с любой функцией (не забывайте, что Dog — это функция). `Dog.call` вызывает функцию Dog и передает ей объект, который должен использоваться как `this`, вместе со всеми аргументами функции Dog. Рассмотрим по частям:

В этом коде мы вызываем конструктор Dog, но приказываем ему использовать наш экземпляр ShowDog как текущий объект this. Соответственно, функция Dog задает свойства name, breed и weight объекта ShowDog.

```
Dog.call(this, name, breed, weight);
```

Dog — вызываемая функция.

Используется как this в теле функции Dog.

Остальные аргументы просто передаются Dog, как обычно.

call — вызываемый метод. Он обеспечивает вызов функции Dog. Мы используем метод call вместо прямого вызова Dog, чтобы управлять значением this.

Вызов Dog.call шаг за шагом

Понять, зачем использовать `Dog.call` для вызова Dog, поначалу непросто, поэтому мы разберем этот вызов еще раз, начиная с измененной версии кода.

```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}
```

Мы будем использовать код конструктора Dog для назначения свойств name, breed и weight.

Но Dog ничего не знает о свойстве handler; это свойство должно назначаться в ShowDog.

Сначала мы вызываем ShowDog с оператором `new`. Напомним, что оператор `new` создает новый пустой объект и назначает его переменной `this` в теле ShowDog.

```
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```

Затем выполняется тело конструктора ShowDog. Выполнение начинается с вызова Dog, для которого используется метод `call`. Он вызывает Dog, передает ему `this`, а также значения `name`, `breed` и `weight` как аргументы.


```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);

            function Dog(name, breed, weight) {
                this.name = name;
                this.breed = breed;
                this.weight = weight;
            }

    this.handler = handler;
}
```

Тело Dog выполняется как обычно, если не считать того, что this содержит объект ShowDog, а не Dog.

Объект this, созданный оператором new для ShowDog, используется в качестве this в теле Dog.

После того как выполнение функции Dog завершится (она ничего не возвращает, потому что не была вызвана оператором `new`), завершается код ShowDog, задающий значение параметра `handler` свойству `this.handler`. Поскольку для вызова ShowDog использовался оператор `new`, возвращается экземпляр ShowDog с инициализированными свойствами name, breed, weight и handler.

Последний тест-драйв

Великолепная работа: мы создали отличную систему, которая наверняка понравится заказчикам из Клуба любителей собак. Проведем последнее тестирование, чтобы собаки могли проявить все свои способности.

```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}
ShowDog.prototype = new Dog();
ShowDog.prototype.constructor = ShowDog;
ShowDog.prototype.league = "Webville";
ShowDog.prototype.stack = function() {
    console.log("Stack");
};

ShowDog.prototype.bait = function() {
      console.log("Bait");
};
ShowDog.prototype.gait = function(kind) {
    console.log(kind + "ing");
};
ShowDog.prototype.groom = function() {
    console.log("Groom");
};
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
var beatrice = new ShowDog("Beatrice", "Pomeranian", 5, "Hamilton");
fido.bark();
fluffy.bark();
spot.bark();
scotty.bark();
beatrice.bark();
scotty.gait("Walk");
beatrice.groom();
```

Мы собрали воедино весь код ShowDog. Добавьте в файл с кодом Dog для тестирования.

Внизу добавлен тестовый код.

Создаем несколько объектов — как обычных, так и выставочных собак.

Убедимся в том, что все объекты работают так, как положено.

Консоль JavaScript

```
Fido says Woof!
Fluffy says Woof!
Spot says Yip!
Scotty says Yip!
Beatrice says Yip!
Walking
Groom
```

Часто задаваемые вопросы

В: Создавая экземпляр собаки, который использовался для прототипа выставочной собаки, мы вызывали конструктор Dog без аргументов. Почему?

О: Потому что от этого экземпляра нам нужно только одно: сам факт наследования от прототипа собаки. Этот экземпляр не обозначает никакую конкретную собаку (Фидо, Спота и т. д.); это обобщенный экземпляр собаки, наследующий от прототипа собаки.

Кроме того, все собаки, наследующие от прототипа выставочной собаки, определяют собственные свойства name, breed и weight. Таким образом, даже если экземпляр собаки содержал значения этих свойств, мы их никогда не увидим, потому что экземпляры выставочной собаки их всегда переопределяют.

В: Что происходит со свойствами в объекте собаки, который мы используем как прототип выставочной собаки?

О: Значения им не присваиваются, поэтому свойства содержат undefined.

В: Если мы не назначим свойству prototype экземпляра ShowDog экземпляр собаки, что произойдет?

О: Выставочные собаки будут работать нормально, но не будут наследовать поведение из прототипа собаки. Это означает, что они не будут поддерживать методы bark, run или wag, а их свойство species не будет содержать строку “Canine”. Попробуйте сами. Закомментируйте строку кода, в которой мы задаем ShowDog.prototype экземпляр new Dog(), и попробуйте вызвать bark для объекта scotty. Что произойдет?

В: Могу ли я создать объектный литерал и использовать его как прототип?

О: Да. Вы можете использовать любой объект как прототип ShowDog. Конечно, в этом случае ваши выставочные собаки ничего не будут наследовать от прототипа собаки. Вместо этого они наследуют свойства и методы, помещенные в объектный литерал.

В: Я случайно разместил строку, в которой ShowDog.prototype присваивался экземпляр Dog, ниже, где создавался объект scotty, и мой код не работал. Почему?

О: Потому что при создании scotty (экземпляра ShowDog) он получает прототип, который был назначен ShowDog.prototype на момент его создания. Таким образом, если объект собаки не был назначен prototype до создания scotty, объект scotty будет использовать другой объект в качестве прототипа (объект, полученный конструктором ShowDog по умолчанию). И этот объект не содержит никакие из свойств прототипа Dog. Прототип выставочной собаки следует назначать сразу же после создания конструктора, но до добавления чего-либо в прототип или создания любых экземпляров ShowDog.

В: Если я изменю свойство в прототипе собаки (скажем, заменю значение species с “Canine” на “Feline”), отразится ли это изменение на созданных мной выставочных собаках?

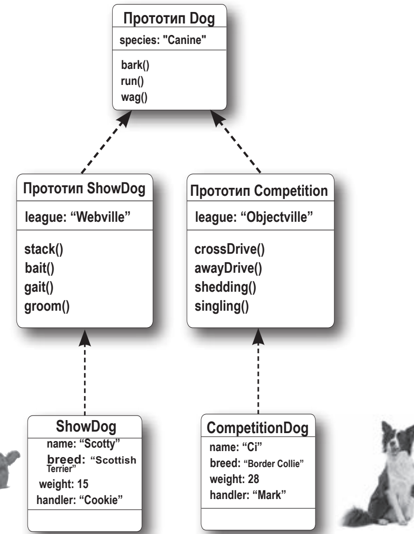
О: Да, все изменения в прототипе влияют на все экземпляры, наследующие от этого прототипа в цепочке, независимо от количества промежуточных звеньев.

В: Ограничена ли длина цепочки прототипов?

О: Теоретически — нет, но на практике — возможно. Чем длиннее цепочка прототипов, тем больше работы потребует разрешение метода или свойства. Впрочем, системы времени выполнения часто хорошо справляются с оптимизацией таких операций. В общем случае вам не понадобятся системы, требующие многоуровневого наследования. А если понадобятся — лучше пересмотрите структуру своих объектов.

В: А если у меня появится еще одна категория собак — скажем, племенные собаки? Смогу ли я создать прототип племенной собаки, который наследует от того же прототипа собаки, что и прототип выставочной собаки?

О: Да, сможете. Вам придется создать отдельный экземпляр собаки, который будет использоваться как прототип племенной собаки, но после этого остается лишь сделать то же, что мы делали при создании прототипа выставочной собаки.

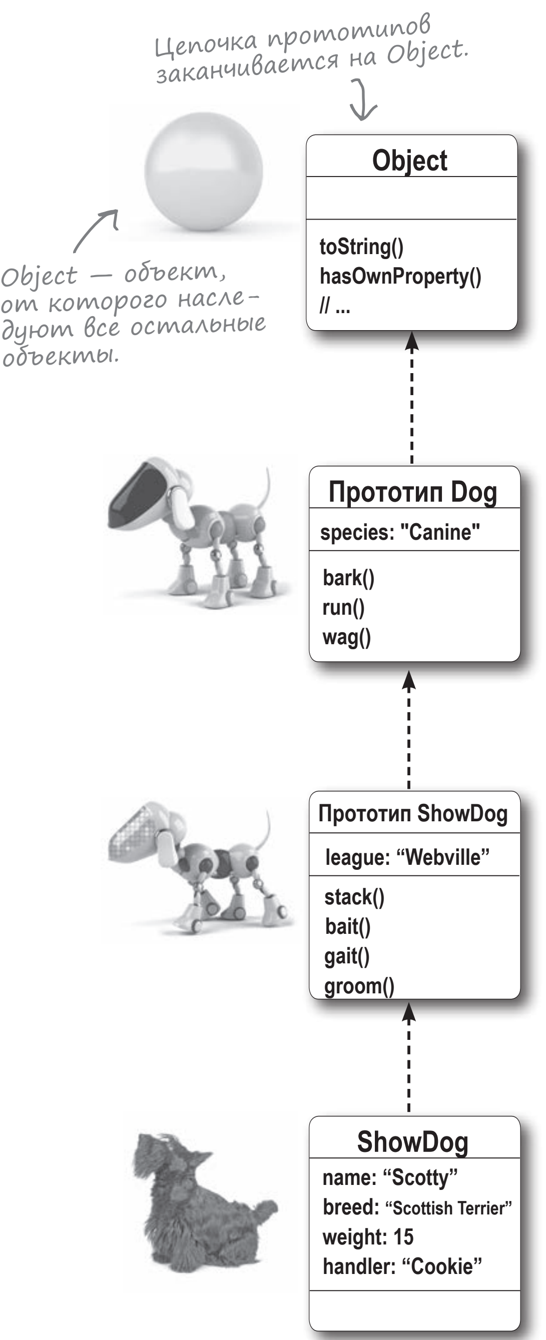


Цепочка не заканчивается на собаках

Вы уже видели пару цепочек прототипов — у нас был исходный прототип, от которого наследовали объекты собак, а также специализированные экземпляры выставочных собак, которые наследовали сначала от прототипа выставочной собаки, а затем от собаки.

Но в обоих ситуациях завершается ли цепочка на прототипе собаки? Нет, потому что у собаки есть свой собственный прототип Object.

Любая создаваемая цепочка прототипов будет завершаться на Object, ведь для создаваемого экземпляра по умолчанию используется прототип Object.



Что такое Object?

Object можно рассматривать как своего рода «объект-прародитель» — от него изначально наследуют все объекты. А еще Object реализует несколько ключевых методов, которые являются основной частью системы объектов JavaScript. Многие из них не будут использоваться в вашей повседневной работе, но другие встречаются достаточно часто.

Один из таких методов уже встречался нам в этой главе: метод `hasOwnProperty` наследуется всеми объектами (еще раз — потому что все объекты в конечном итоге наследуют от Object). Если вы еще не забыли, мы использовали метод `hasOwnProperty` в этой главе для определения того, находится ли свойство в экземпляре объекта или в одном из его прототипов.

Также от Object наследуется метод `toString`, который обычно переопределяется в экземплярах. Этот метод возвращает представление объекта в формате String. Вскоре вы увидите, как переопределить этот метод для получения более точного описания наших объектов.

Object как прототип

Итак, сознавали вы это или нет, но у каждого объекта, который мы когда-либо создавали, был прототип — и это был Object. Вы можете задать в качестве прототипа объекта другой вид объекта, как мы сделали с прототипом выставочных собак, но в конечном счете все цепочки прототипов ведут к Object.

Использование наследования... с переопределением встроенного поведения

При наследовании от встроенных объектов вы можете переопределять методы, содержащиеся в этих объектах. Типичный пример — метод `toString` прототипа Object. Все объекты наследуют от Object, и поэтому могут использовать метод `toString` для получения простого строкового представления любого объекта, например `console.log`, для вывода представления объекта на консоль:

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var toy = new Robot("Toy", 2013, "Avary");

console.log(toy.toString());
```

Консоль JavaScript

```
[Object object]
```

Метод toString, наследуемый от Object, не очень хорошо справляется со своими обязанностями.

Как видите, метод `toString` не лучшим образом преобразует игрушечного робота в строку. Однако мы можем переопределить метод `toString` и написать реализацию, которая создает строку специально для объектов Robot:

```
function Robot(name, year, owner) {
    // Тот же код
}

Robot.prototype.toString = function() {
    return this.name + " Robot belonging to " + this.owner;
};

var toy = new Robot("Toy", 2013, "Avary");


console.log(toy.toString());
```

Консоль JavaScript

```
Toy Robot belonging to Avary
```

Так гораздо лучше! Здесь используется наша нестандартная реализация toString.

Обратите внимание: метод `toString` может активизироваться, даже если вы не вызываете его напрямую. Например, если вы используете оператор + для конкатенации строки с объектом, JavaScript использует метод `toString` для преобразования объекта в строку, прежде чем объединять его с другой строкой.

```
console.log("Robot is: " + toy);
```

Объект toy преобразуется в строку методом toString перед конкатенацией. Если объект toy переопределяет toString, то он использует этот метод.

Игрушечный? Да этот робот работает на базе Arduino, и им даже можно управлять из JavaScript!

ОПАСНАЯ ЗОНА

Начиная переопределять свойства и методы, легко увлечься. Особенно важно действовать осторожно при переопределении свойств и методов встроенных объектов — вы можете случайно изменить поведение другого кода, зависящего от этих свойств для выполнения определенных операций.

Итак, если вы собираетесь переопределять свойства в Object, прочитайте сначала правила техники безопасности. Иначе в вашем коде могут появиться крайне коварные и трудноуловимые ошибки.

НЕ УВЕРЕН — НЕ ПЕРЕОПРЕДЕЛЯЙ!

Свойства Object, которые переопределять не рекомендуется:

constructor — Свойство constructor указывает на функцию-конструктор, связанную с прототипом.

hasOwnProperty — Вы уже знаете, что делает метод hasOwnProperty.

isPrototypeOf — С помощью метода isPrototypeOf можно узнать, является ли объект прототипом другого объекта.

propertyIsEnumerable — Метод propertyIsEnumerable проверяет, можно ли получить доступ к свойству при переборе всех свойств объекта.

МОЖНО ПЕРЕОПРЕДЕЛЯТЬ

Итак, теперь вы разбираетесь в прототипах и умеете правильно выполнять переопределение. Некоторые свойства Object, которые могут пригодиться в вашем коде:

toString

toLocaleString — Метод toLocaleString, как и toString, преобразует объект в строку. Этот метод переопределяется для построения локализованной строки (страны/языка) с описанием объекта.

valueOf — valueOf — еще один метод, предназначенный для переопределения. По умолчанию он просто возвращает объект, для которого был вызван, но вы можете переопределить его, чтобы метод возвращал любое другое значение.

Применяем наследование с пользой... расширяя встроенный объект

Вы уже знаете, что добавление методов в прототип позволяет включить новую функциональность во все экземпляры этого прототипа. Это относится не только к вашим собственным, но и к встроенным объектам.

Помните: обычно мы рассматриваем строки как примитивные типы, но они также имеют объектную форму. JavaScript автоматически преобразует строку в объект, когда потребуется.

Для примера возьмем объект String — мы уже использовали в коде методы String (`substring`). Но что если вы захотите добавить собственный метод, чтобы он мог использоваться любым экземпляром String? Прием расширения объектов через прототип можно использовать и для String.

Допустим, мы хотим расширить прототип String методом `cliche`, который возвращает true, если строка содержит банальные выражения из заранее известного списка. Вот как это можно сделать:

```
String.prototype.cliche = function() {
    var cliche = ["lock and load","touch base", "open the kimono"];

    for (var i = 0; i < cliche.length; i++) {
        var index = this.indexOf(cliche[i]);
        if (index >= 0) {
            return true;
        }
    }
    return false;
};
```

В прототип String добавляется метод cliche.

Определяем клише, которые нужно найти в тексте.

А затем используем функцию indexOf объекта String, чтобы узнать, содержит ли строка искомые клише. Если поиск окажется успешным, немедленно возвращается значение true.

Обратите внимание: this — строка, для которой вызывается метод cliche.

Напишем код для тестирования метода:

Для тестирования создадим несколько предложений, пара из которых содержит клише.

```
var sentences = ["I'll send my car around to pick you up.",
                 "Let's touch base in the morning and see where we are",
                 "We don't want to open the kimono, we just want to inform them."];

for (var i = 0; i < sentences.length; i++) {
    var phrase = sentences[i];
    if (phrase.cliche()) {
        console.log("CLICHE ALERT: " + phrase);
    }
}
```

Каждый элемент sentences представляет собой строку, для которой можно вызвать метод cliche.

Для создания строки не нужно использовать конструктор String и оператор new. JavaScript автоматически преобразует каждую строку в объект String, когда мы вызываем метод cliche.

Если возвращается true, значит, в строке найдены клише.

Тест-драйв программы поиска клише

Сохраните код в файле HTML, откройте браузер и загрузите страницу. Откройте консоль; на ней должны появиться следующие сообщения:

Прекрасно работает. Осталось убедить корпоративную Америку внедрить этот код!

Консоль JavaScript

```
CLICHE ALERT: Let's touch base in the morning
and see where we are
CLICHE ALERT: We don't want to open the kimono,
we just want to inform them.
```

Будьте осторожны!

Будьте осторожны при расширении встроенных объектов (таких, как String) вашими собственными методами.

Следите за тем, чтобы имя, выбранное вами для метода, не конфликтовало с именем существующего метода объекта. А при подключении стороннего кода изучите все нестандартные расширения, которые в нем могут использоваться (и снова следите за конфликтами имен). Наконец, некоторые встроенные объекты просто не рассчитаны на расширение (как, например, Array). В общем, основательно изучите обстановку, прежде чем браться за добавление методов во встроенные объекты.

Упражнение

Ваша очередь. Напишите метод palindrome, который возвращает true, если строка читается одинаково в обоих направлениях (будем считать, что строка содержит всего одно слово — не будем отвлекаться на палиндромы из нескольких слов). Добавьте метод в String.prototype и протестируйте результат. Сверьтесь с ответом в конце главы.

Теория Великого объединения ~~Всего~~ JavaScript

Поздравляем, вы взялись за изучение нового языка программирования (а может, вашего первого языка) и успешно справились с этой задачей. Раз вы дочитали до этой страницы, следовательно, вы знаете о JavaScript больше, чем практически все остальные.

Мы исходим из того, что примерно 5,9 миллиардов человек вообще не знают JavaScript, так что всех остальных можно рассматривать как ошибку округления. Из этого следует, что вы знаете JavaScript практически лучше всех остальных.

А если говорить серьезно, то добравшись до конца книги, вы существенно продвинулись на пути к тому, чтобы стать экспертом JavaScript. Остается накопить опыт проектирования и программирования веб-приложений (да и любых приложений JavaScript).

Объекты для лучшей жизни

При изучении такой сложной темы, как JavaScript, порой бывает трудно увидеть «лес за деревьями». Но когда большая часть понятна, становится проще сделать шаг назад и полюбоваться лесом.

Изучая JavaScript, вы последовательно знакомитесь с основными понятиями: изучаете примитивы (которые могут использоваться как объекты), массивы (которые временами действуют как объекты), функции (которые имеют свойства и методы, как объекты), конструкторы (которые выглядят как полуфункции, полуобъекты) и... сами объекты. Все это выглядит достаточно сложно.

После всего, что вы узнали, отступите назад, рассслабьтесь, неспешно вдохните и помедитируйте: «Существуют только объекты».

Существуют только объекты...
Существуют только объекты...
Существуют только объекты...

Конечно, существуют примитивы — булевские значения, числа и строки, но мы уже знаем, что они могут в любой момент интерпретироваться как объекты. Существуют встроенные типы — такие, как Date, Math и RegEx, но и они являются обычными объектами. Даже массивы являются объектами и, как вы знаете, выглядят иначе они только потому, что JavaScript предоставляет «синтаксические удобства», которые упрощают создание и работу с объектами. И конечно, существуют и сами объекты, с простотой объектных литералов и мощью объектных систем на базе прототипов.

Но как насчет функций? Это тоже объекты? Давайте посмотрим:

```
function meditate() {
    console.log("Everything is an object...");
}
alert(meditate instanceof Object);
```

Итак, это правда: функции — это тоже объекты. Но к этому моменту это уже не должно вас особенно удивлять. В конце концов, мы можем присваивать функции переменным (как объекты), передавать их в аргументах (как объекты), возвращать из функций (как объекты). Мы даже видели, что они обладают свойствами:

```
Dog.constructor
```

Напоминаем: это функция.

А это свойство.

Это правда! Функции тоже являются объектами.

Ничто не мешает вам добавить собственные свойства к функции, если это будет удобно. И кстати говоря, чтобы окончательно прояснить картину: вы задумывались, что метод — всего лишь свойство объекта, которому назначается анонимное функциональное выражение?

Собираем все вместе

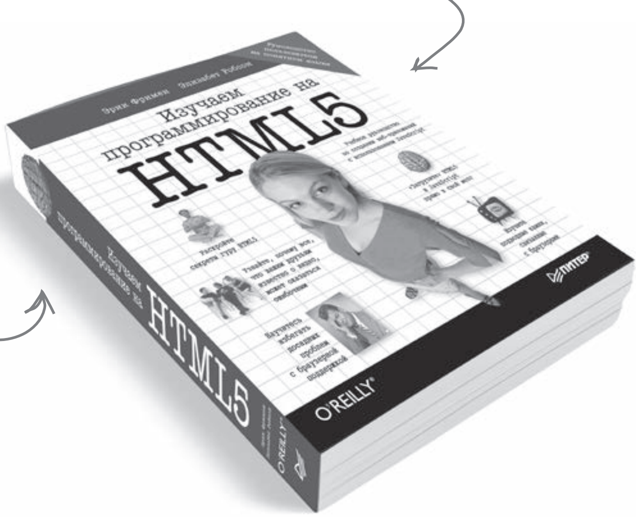
Значительная часть мощи и гибкости JavaScript происходит от взаимодействий между функциями и объектами, а также возможности использования их как первоклассных значений. Если задуматься, все эффективные концепции программирования, которые мы изучали, — конструкторы, замыкания, создание объектов с поведением, которое можно использовать заново и расширять, параметризация использования функций и т. д. — базировались на понимании нетривиальных возможностей объектов и функций.

А сейчас вы знаете достаточно для того, чтобы пойти еще дальше...

На сайте http://wickedlysmart.com/javascript можно найти дополнительные материалы к книге — и сделать следующий шаг, если вы на него решитесь...

Что дальше?

Мы разобрались с азами и можно двигаться дальше. Вы готовы применить свой опыт для работы с браузером и программными интерфейсами. В этом вам поможет книга *Изучаем программирование на HTML5* , в которой вы узнаете, как добавить в ваши приложения поддержку геопозиционирования, графического вывода, локального хранения данных, веб-потоков и т. д.



Эта область стремительно развивается, так что перед тем, как браться за книгу «Изучаем программирование на HTML5», загляните на сайт http://wickedlysmart.com/javascript за последними рекомендациями, обновлениями и исправлениями к этой книге.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Система объектов JavaScript использует **наследование через прототипы**.
- Экземпляр, создаваемый конструктором, содержит собственные свойства и копию методов из конструктора.
- Свойства, добавляемые в прототип конструктора, **наследуются** всеми экземплярами, созданными этим конструктором.
- Размещение свойств в прототипе может сократить дублирование кода объектов на стадии выполнения.
- Чтобы **переопределить** свойства из прототипа, просто добавьте их в экземпляр.
- Конструктор включает **прототип** по умолчанию, к которому можно обратиться через свойство prototype конструктора.
- Вы можете назначить собственный объект свойству prototype конструктора.
- Если вы используете собственный объект как прототип, не забудьте назначить правильный конструктор свойству constructor для предотвращения недоразумений.
- Если вы добавите свойства в прототип после создания экземпляров, наследующих от него, то все экземпляры немедленно унаследуют новые свойства.
- Чтобы узнать, определено ли свойство в экземпляре, используйте метод **hasOwnProperty**.
- Метод **call** может использоваться для вызова функции с указанием объекта, который должен использоваться в качестве **this** в теле функции.
- **Object** — объект, от которого в конечном итоге наследуют все прототипы и экземпляры.
- Object содержит свойства и методы, которые наследуются всеми объектами (как, например, toString и hasOwnProperty).
- Вы можете переопределять или добавлять свойства во встроенные объекты (такие, как Object или String), но будьте осторожны — такие изменения могут приводить к непредвиденным последствиям.
- В JavaScript объектом является практически все — функции, массивы, многие встроенные объекты и все объекты, которые вы создаете самостоятельно.

Развлечения с магнитами. Решение

На холодильнике была выложена диаграмма объектов, но кто-то все перепутал. Сможете ли вы расставить магниты по своим местам? На диаграмме будут присутствовать два экземпляра, созданных на базе прототипа. Первый — Робби (Robby) — создан в 1956 году, принадлежит доктору Морбиусу (Dr. Morbius), оснащен выключателем и умеет бегать в Starbucks за кофе. Другой робот — Рози (Rosie) — создан в 1962 году, прибирается в доме Джорджа Джетсона (George Jetson). Удачи! Кстати, внизу могут остаться лишние магниты...

Наше решение выглядит так:

Прототип, от которого могут наследовать ваши роботы.

Прототип Robot
maker: "ObjectsRUs"
speak()
makeCoffee()
blinkLights()

```
function speak() {
  // code to speak
}
```

```
function
makeCoffee() {
  // code 4 coffee
}
```

```
function
blinkLights() {
  // code 4 lights
}
```

Robot
name: "Robby"
year: 1956
owner: Dr. Morbius
onOffSwitch: true
makeCoffee()

```
function
makeCoffee() {
  // starbucks
}
```

Robot
name: "Rosie"
year: 1962
owner: George Jetson
cleanHouse()

```
function
cleanHouse() {
  // code 4 clean
}
```

Помните нашу диаграмму объектов с роботами Robby и Rosie? Сейчас мы реализуем ее в программном коде. Мы уже написали за вас конструктор Robot, а также добавили тестовый код. Вам остается настроить прототип и реализовать двух роботов. Не забудьте проверить, что у вас получилось. Ниже приведено наше решение.

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}
Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.speak = function() {
    alert("Warning warning!!");
};
Robot.prototype.makeCoffee = function() {
    alert("Making coffee");
};
Robot.prototype.blinkLights = function() {
    alert("Blink blink!");
};
var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");
robby.onOffSwitch = true;
robby.makeCoffee = function() {
    alert("Fetching a coffee from Starbucks.");
};
rosie.cleanHouse = function() {
    alert("Cleaning! Spic and Span soon...");
};
console.log(robby.name + " was made by " + robby.maker +
            " in " + robby.year + " and is owned by " + robby.owner);
robby.makeCoffee();
robby.blinkLights();
console.log(rosie.name + " was made by " + rosie.maker +
            " in " + rosie.year + " and is owned by " + rosie.owner);
rosie.cleanHouse();
```

Базовый конструктор Robot.

Здесь мы назначаем прототип со свойством maker...

...и три метода, общие для всех роботов.

Здесь создаются наши роботы, Робби и Рози.

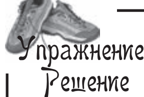
Здесь мы добавляем в Робби новое свойство, а также метод для приготовления кофе (из Starbucks).

А Рози получает новый метод для уборки в доме (и почему этим должны заниматься роботы женского пола?)

Результат (а также некоторые сообщения, которые здесь не приводятся).

Консоль JavaScript

```
Robby was made by ObjectsRUs in 1956
and is owned by Dr. Morbius
Rosie was made by ObjectsRUs in 1962
and is owned by George Jetson
```



Наши роботы используются в компьютерной игре, код которой приведен ниже. В этой игре при достижении игроком уровня 42 у робота включается новая способность: лазерный луч. Допишите приведенный ниже код, чтобы на уровне 42 лазерный луч включался у обоих роботов, Робби и Рози. Ниже приведено наше решение.

```
function Game() {
    this.level = 0;
}

Game.prototype.play = function() {
    // Действия игрока
    this.level++;
    console.log("Welcome to level " + this.level);
    this.unlock();
}

Game.prototype.unlock = function() {
    if (this.level === 42) {
        Robot.prototype.deployLaser = function () {
            console.log(this.name +  " is blasting you with laser beams.");
        }
    }
}

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var game = new Game();
var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

while (game.level < 42) {
     game.play();
}

robby.deployLaser();
rosie.deployLaser();
```

Мы вызываем unlock при каждой игре, но новая способность не активизируется до тех пор, пока значение level не достигнет 42.

Секрет этой игры: при достижении уровня 42 в прототип добавляется новый метод. Это означает, что все роботы наследуют способность применения лазеров!

Консоль JavaScript

```
Welcome to level 1
Welcome to level 2
Welcome to level 3
 ...
Welcome to level 41
Welcome to level 42
Rosie is blasting you with
laser beams.
```

Пример вывода. Завершив работу над кодом, запустите его и посмотрите, какой робот победит!

Мы встроили в наших роботов Робби и Рози новую функцию: теперь они могут сообщать о возникающих ошибках при помощи метода reportError. Проанализируйте приведенный ниже код, обращая особое внимание на то, откуда этот метод получает информацию об ошибках и откуда берется эта информация: из прототипа или экземпляра.

Ниже приведено наше решение.

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.errorMessage = "All systems go.";
Robot.prototype.reportError = function() {
    console.log(this.name + " says " + this.errorMessage);
};
Robot.prototype.spillWater = function() {
    this.errorMessage = "I appear to have a short circuit!";
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

rosie.reportError();
robby.reportError();
robby.spillWater();
rosie.reportError();
robby.reportError();

console.log(robby.hasOwnProperty("errorMessage"));   true
console.log(rosie.hasOwnProperty("errorMessage"));   false
```

Метод reportError только использует значение errorMessage, поэтому он не переопределяет свойство.

Метод spillWater присваивает новое значение this.errorMessage, что приводит к переопределению свойства в прототипе для любого робота, вызывающего этот метод.

Мы вызываем метод spillWater для объекта robby. Так Робби получает собственное свойство errorMessage, которое переопределяет свойство в прототипе.

Но метод spillWater никогда не вызывается для Рози, поэтому она наследует свойство от прототипа.

Развлечения с магнитами. Решение

Мы выложили на холодильнике очередную диаграмму объектов, и снова кто-то все перепутал. Снова! Сможете ли вы расставить магниты по местам? Нам понадобится новая линейка космических роботов, наследующих свойства от обычных роботов. Космические роботы переопределяют метод speak обычных роботов, а также расширяют функциональность роботов новым свойством homePlanet. Ниже приведено наше решение.

Упражнение
Решение

Ваша очередь. Напишите метод palindrome, который возвращает true, если строка читается одинаково в обоих направлениях (будем считать, что строка содержит всего одно слово — не будем отвлекаться на палиндромы из нескольких слов). Добавьте метод в String.prototype и протестируйте результат. Ниже приведено наше решение.

```
function SpaceRobot(name, year, owner, homePlanet) {
    this.name =  name;
    this.year = year;
    this.owner = owner;
    this.homePlanet = homePlanet;
}

SpaceRobot.prototype = new Robot();

SpaceRobot.prototype.speak = function() {
    alert(this.name + " says Sir, If I may venture an opinion...");
};

SpaceRobot.prototype.pilot = function() {
    alert(this.name + " says Thrusters? Are they important?");
};

var c3po = new SpaceRobot("C3PO", 1977, "Luke Skywalker", "Tatooine");
c3po.speak();
c3po.pilot();
console.log(c3po.name + " was made by " + c3po.maker);

var simon = new SpaceRobot("Simon", 2009, "Carla Diana", "Earth");
simon.makeCoffee();
simon.blinkLights();
simon.speak();
```

Конструктор SpaceRobot похож на конструктор Robot, если не считать того, что у нас появляется дополнительное свойство homePlanet для экземпляров SpaceRobot.

Прототип SpaceRobot должен наследовать от прототипа Robot, поэтому мы назначаем экземпляр Robot свойству prototype конструктора SpaceRobot.

Эти два метода добавлены в прототип.

Наш результат (некоторые сообщения опущены).

Консоль JavaScript

```
C3PO was made by ObjectsRUs
```

Ваша очередь. Напишите метод palindrome, который возвращает true, если строка читается одинаково в обоих направлениях. Добавьте метод в String.prototype и протестируйте результат. Ниже приведено наше решение (только для палиндромов из одного слова).

```
String.prototype.palindrome = function() {
    var len = this.length-1;
    for (var i = 0; i <= len; i++) {
        if (this.charAt(i) !== this.charAt(len-i)) {
            return false;
        }
        if (i === (len-i)) {
            return true;
        }
    }
    return true;
};

var phrases = ["eve", "kayak", "mom", "wow", "Not a palindrome"];

for (var i = 0; i < phrases.length; i++) {
    var phrase = phrases[i];
    if (phrase.palindrome()) {
        console.log("'" + phrase + "' is a palindrome");
    } else {
        console.log("'" + phrase + "' is NOT a palindrome");
    }
}
```

Сначала получаем длину строки.

Мы проходим по каждому символу в строке и проверяем совпадение символов в позициях i и len-i.

Если они не равны, мы возвращаем false, потому что строка не является палиндромом.

Если i доходит до середины строки или цикл выполняется до конца, мы возвращаем true, потому что строка является палиндромом.

Несколько слов для тестирования.

Просто перебираем каждое слово в массиве и вызываем для него метод palindrome. Если метод возвращает true, значит, слово является палиндромом.

Продвинутое решение

```
String.prototype.palindrome = function() {
    var r = this.split("").reverse().join("");
    return (r === this.valueOf());
}
```

Мы разбиваем строку на буквы; каждая буква представляется элементом массива. Затем элементы массива переставляются в обратном порядке, а буквы объединяются в строку. Если исходная строка равна новой, значит, она — палиндром. Обратите внимание на valueOf — это объект, а не строковый примитив (как r), без valueOf строка сравнивается с объектом, и равенства не будет даже в случае палиндрома.

Поздравляем!

Вы добрались до конца.

И что же дальше? Столько интересного!
Теперь, когда вы получили достаточно серьезные познания в JavaScript, пора становиться мастером.
Присоединяйтесь к нам на сайте http://wickedlysmart.com/hfjs, и мы продолжим наше путешествие.

Э. Фримен, Э. Робсон

Изучаем программирование на JavaScript

Перевел с английского Е. Матвеев

Заведующий редакцией	*П. Щеголев*
Ведущий редактор	*Ю. Сергиенко*
Художник	*С. Маликова*
Корректор	*С. Беляева*
Верстка	*Н. Лукьянова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

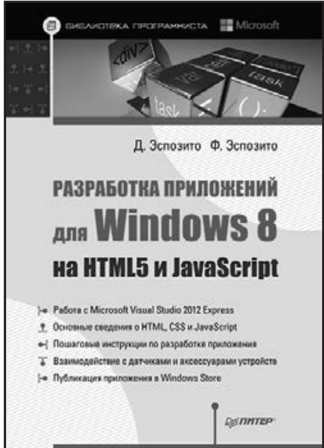
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 08.12.14. Формат 84×108/16. Усл. п. л. 67,200. Тираж 2000. Заказ 0000.

Отпечатано в соответствии с предоставленными материалами в ООО «ИПК Парето-Принт».
170546, Тверская область, Промышленная зона Боровлево-1, комплекс № 3А, www.pareto-print.

Разработка приложений для Windows 8 на HTML5 и JavaScript

Д. Эспозито, Ф. Эспозито



ISBN 978-5-496-00794-8

Объем: 384 с.

С помощью этой книги вы быстро освоите разработку приложений для Windows 8 с использованием таких технологий, как HTML5 и JavaScript. Написанное известным экспертом Дино Эспозито в соавторстве со своим сыном, это практическое пособие содержит все необходимое для того, чтобы помочь читателю спроектировать, создать и опубликовать свое приложение для Windows 8. Издание состоит из трех частей. В первой части рассматриваются вопросы использования Microsoft Visual Studio 2012 Express, а также даются краткие сведения об HTML, CSS и JavaScript. Во второй части книги рассматриваются основы программирования для Windows 8 с предоставлением пошаговых упражнений, помогающих освоить пользовательский интерфейс Windows 8, графику, видео, хранилища данных, интернет-вызовы. В третьей части основное внимание уделяется современному программированию для Windows 8 с упором на работу с датчиками и аксессуарами устройств (такими, как принтеры, GPS, веб-камеры и т. д.), взаимодействию с системой и публикации готового приложения в Windows Store.