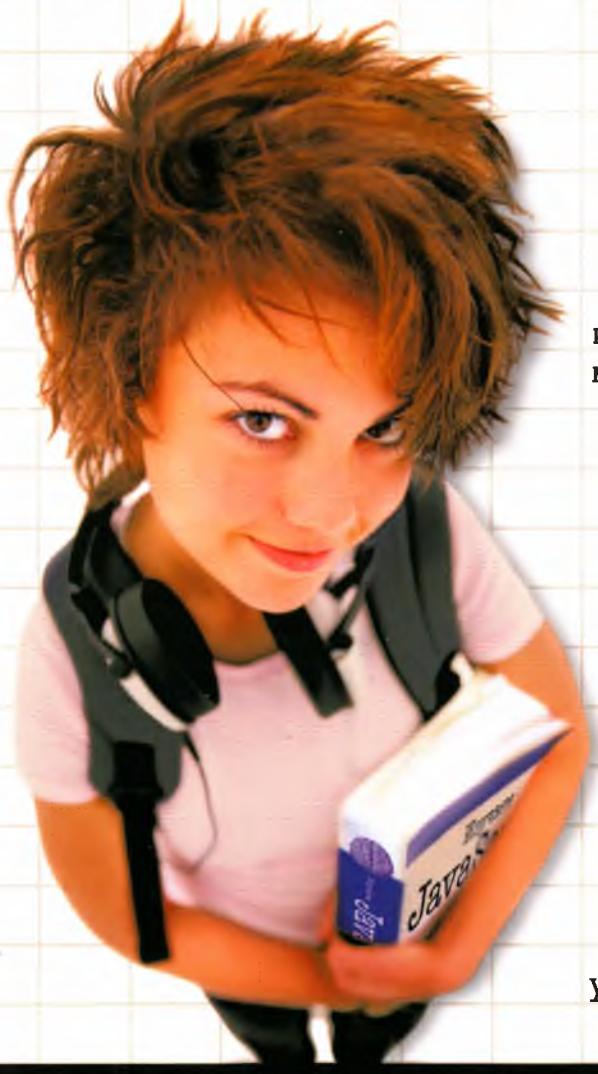
Майкл Моррисон

Изучаем JavaScript

Улучшай качество взаимодействия пользователя с веб-страницей



Освой концепцию и синтаксис JavaScript максимально эффективно

Научись оптимизировать JavaScript-код

Управляй HTML-кодом с помощью DOM

Избавься от страха перед обработчиком событий

Проверяй свои знания с помощью сотен упражнений и примеров

O'REILLY®

ПИТЕР®

Head First JavaScript

Michael Morrison

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Изучаем JavaScript

Майкл Моррисон

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

Майкл Моррисон

Изучаем JavaScript

Перевела с английского И. Рузмайкина

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Научный редактор	*С. Бойко*
Литературный редактор	*Е. Пасечник*
Художественный редактор	*Л. Адуевская*
Корректоры	*В. Листова, И. Тимофеева*
Верстка	*Л. Харитонов*

ББК 32.988.02-018.1 УДК 004.43

Моррисон М.

М80 Изучаем JavaScript. — СПб.: Питер, 2012. — 608 с.: ил.

ISBN 978-5-459-00322-2

Вы готовы сделать шаг вперед в своей практике веб-программирования 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 java-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.

ISBN 978-0596527747 англ.

ISBN 978-5-459-00322-2

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.
Подписано в печать 27.09.11. Формат 84x100/16. Усл. п. л. 63,840. Тираж 2500. Заказ 26462.
Отпечатано по технологии CtP в ОАО «Первая Образцовая типография», обособленное подразделение «Печатный двор».
197110, Санкт-Петербург, Чкаловский пр., 15.

Посвящается ребятам из Netscape, которые еще в прошлом веке мечтали, чтобы Интернет стал чем-то большим, чем гигантской книгой с массой гиперссылок.

Хотя, конечно, это они намечтали ужасный тег `<blink>`... продемонстрировав, что в мечтах не следует заходить слишком далеко!

Автор книги Head First JavaScript

Первым компьютером **Майкла Моррисона** был TI-99/4A, укомплектованный эргономичной клавиатурой, черно-белым «монитором», роль которого играл телевизор, и набором кассет со стереосистемой. С того времени он сменил множество компьютеров, но до сих пор скучает по играм в Parsec на стареньком TI.

В настоящее время интересы Майкла сместились в сторону создания интерактивных веб-приложений и... катания на роликовой доске. К решению технических проблем он подходит с той же беспечной отвагой, как к рискованному спорту. Создав несколько видеоигр, изобретя пару игрушек, написав дюжину компьютерных книг и основав множество компьютерных курсов, Майкл наконец пришел к идее написать книгу, посвященную JavaScript.

Впрочем, по-настоящему подготовиться к написанию книг серии Head First невозможно. Нужно просто принять красную пилюлю и провалиться в Матрицу, которая называется Head First. Получив такой опыт, Майкл уже никогда не будет смотреть на процесс обучения по-старому. Чему он крайне рад. Сейчас он с женой сидит на берегу своего пруда с золотыми рыбками, отражающего чудеса интерактивного Интернета.

Краткое содержание

Содержание

Введение

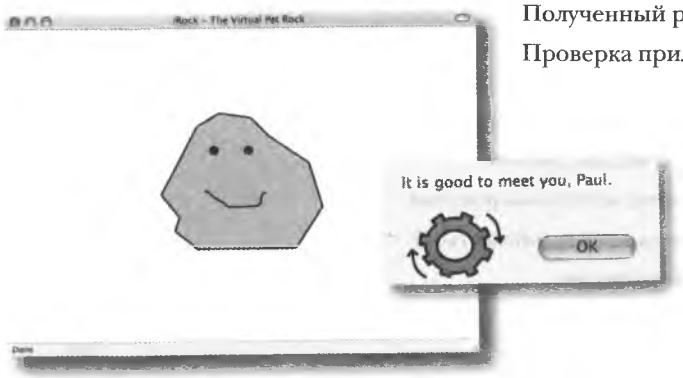
Ваш мозг думает о JavaScript. Вы сидите за книгой и пытаетесь что-нибудь *выучить*, но ваш *мозг* продолжает считать, что вся эта писанина не важна. Ваш мозг говорит: «Выгляни в окно! На свете есть более важные вещи. Например, серфинг или голодный тигр, когда ты попался на его пути». Как *заставить* ваш мозг думать, что ваша жизнь действительно зависит от знания JavaScript?

1

интерактивная сеть

Реакции виртуального мир

Устали представлять Интернет набором пассивных страниц? Кто из нас не держал в руках книг. Их читаешь, в них находишь информацию. Но они не **интерактивны**. Как и интернет-страницы без JavaScript. Без сомнения, отправить данные формы и проделать другие трюки можно и при помощи кода HTML и CSS, но реальная **интерактивность** требует **более умного подхода** и большей работы... зато и результат впечатляет ***намного больше***.



Краткое содержание

Содержание

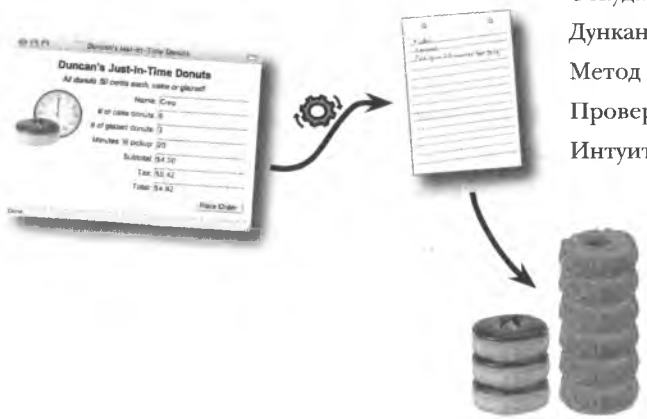
Введение

Ваш мозг думает о JavaScript. Вы сидите за книгой и пытаетесь что-нибудь *выучить*, но ваш *мозг* продолжает считать, что вся эта писанина не важна. Ваш мозг говорит: «Выгляни в окно! На свете есть более важные вещи. Например, серфинг или голодный тигр, когда ты попался на его пути». Как *заставить* ваш мозг думать, что ваша жизнь действительно зависит от знания JavaScript?

2 Хранение данных

Все на своем месте

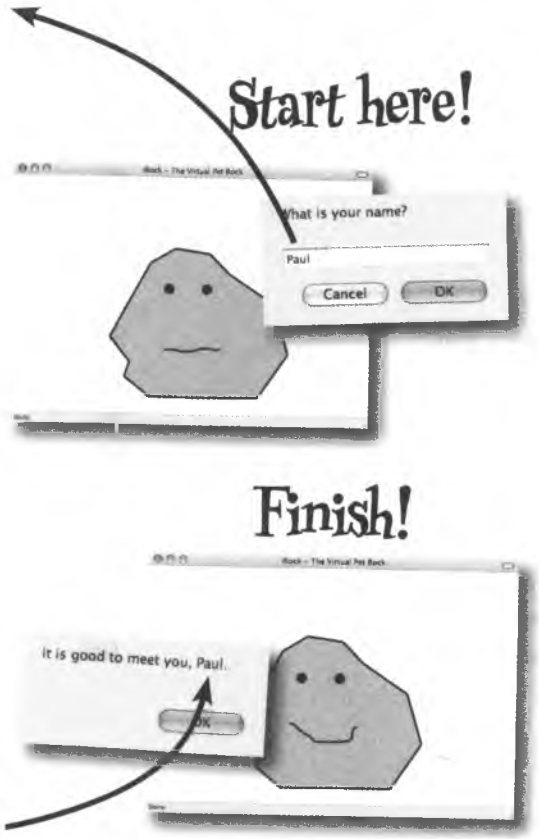
В реальном мире люди часто не придают значения местам для хранения своего имущества. В JavaScript такое поведение невозможно. Ведь там не существует роскоши в виде огромных шкафов и гаражей на три машины. В JavaScript **все имеет свое место**, и ваша задача в этом убедиться. Мы поговорим о **данных** — как их ***представить***, как ***хранить их*** и как их ***найти*** после сохранения. Вы научитесь превращать захламленые комнаты с данными в аккуратные помещения с ящиками, каждый из которых имеет пометку.



3 Исследование клиента

Знакомство с браузером

Иногда JavaScript хочет знать, что происходит в окружающем мире. Ваши сценарии могут существовать в качестве кода на веб-страницах, но по большей части они живут в мире, создаваемом браузером или клиентом. **Умным сценариям** часто необходимо знать больше о мире, в котором они живут, в этом случае они могут **общаться с браузером**, чтобы узнать про него как можно больше. Независимо от того, что требуется узнать: размер экрана или нажата ли кнопка в браузере, они постоянно поддерживают отношения с браузером.



4 Принятие решений

Если на дороге развилка...

Жизнь неотделима от принятия решений. Стоять или идти, пойти на сделку с негодяем или пойти в суд... Результата невозможно добиться без выбора. То же самое происходит в JavaScript — **вам приходится выбирать между различными вариантами сценария**. ***Приходится то и дело принимать решения***. Стоит ли поверить данным, введенным пользователем, и отправить его охотиться на львов? Или же проверить еще раз, может быть, он всего лишь пытался заказать билет до Львова? Выбор за вами!

5 Циклы

Рискуя повториться

Говорят, что повторение — мать учения. Заниматься новыми и интересными делами здорово, но наши дни, как правило, состоят из рутины. Доведенное до автоматизма мытье рук, нервный тик, щелчок на кнопке Reply To All при получении любого дурацкого сообщения! Кажется, повторение не самая лучшая вещь в этом мире. А вот в мире JavaScript без него никак. Вы удивитесь, как часто бывают востребованы одни и те же фрагменты кода. Здесь вам на помощь приходят циклы. Без них пришлось бы снова и снова набирать один и тот же код.

seat_avail.png

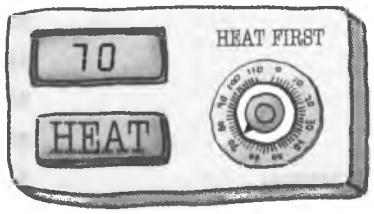
seat_unavail.png

seat_select.png

6 Функции

Многократное использование

Начни JavaScript выступать за экологию, это выступление возглавили бы функции. Ведь именно они увеличивают эффективность кода и позволяют использовать его многократно. Они ориентированы на решение задач и позволяют все систематизировать. Функции дают возможность упростить любой сценарий, ну кроме разве что и так простых. Их значение невозможно оценить, поэтому просто скажем, что именно функции делают сценарии такими экологичными.



7 Формы и проверка данных

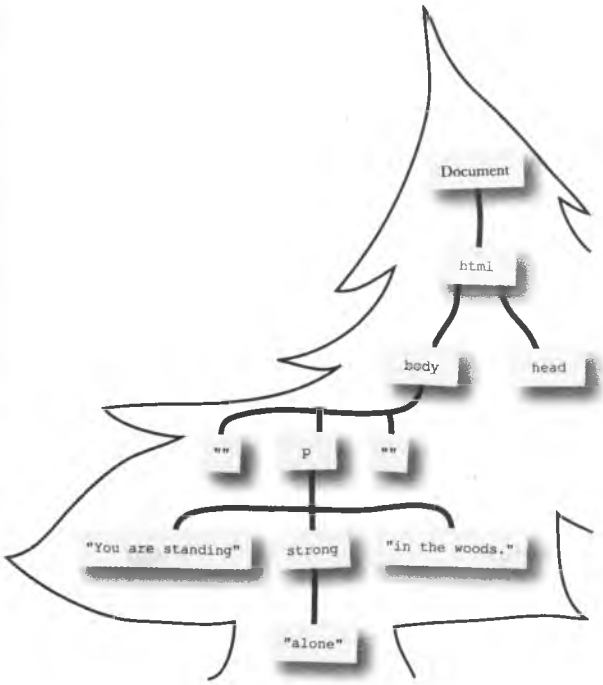
Пусть он все расскажет

Для получения информации от пользователей при помощи JavaScript вам не потребуется быть джентльменом. Но вы должны быть аккуратны. Люди часто делают ошибки, а это означает, что данные, полученные при помощи веб-форм, далеко не всегда ***корректны***. **Проверяя вводимые данные при помощи JavaScript**, вы увеличиваете **надежность веб-приложений** и снимаете дополнительную нагрузку с серверов. Полоса пропускания нам пригодится для восхитительных видеороликов и чудесных фотографий.

Управление страницами

8 Управление HTML с DOM

Управление содержимым веб-страницы при помощи JavaScript напоминает приготовление еды. Конечно, это не настолько грязное занятие... И, увы, вы не сможете съесть результат своих трудов. Тем не менее вы получаете ***полный доступ к HTML***-ингредиентам, из которых состоит веб-страница, и, что еще важнее, вы можете ***менять*** исходный рецепт. Ведь **JavaScript дает возможность управлять HTML-кодом веб-страницы**, что открывает для вас целый ряд интереснейших перспектив, которые реализуются посредством ***набора стандартных объектов*** **DOM**.

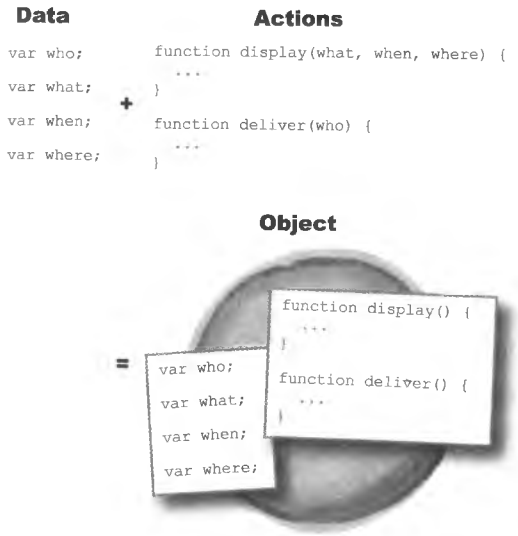


9

Оживляем данные

Объекты как франкенданные

Объекты JavaScript вовсе не так ужасны, как заставил вас думать доктор. Зато они интересны тем, что соединяют друг с другом отдельные части языка JavaScript, делая его более мощным. **Объекты объединяют данные с действиями** в новый **тип**, намного более «живой», чем все, что вы использовали раньше. Вы познакомитесь с **массивами, которые сортируют себя сами**, со строками, которые умеют искать в своем составе указанные последовательности символов, и многими другими замечательными особенностями.

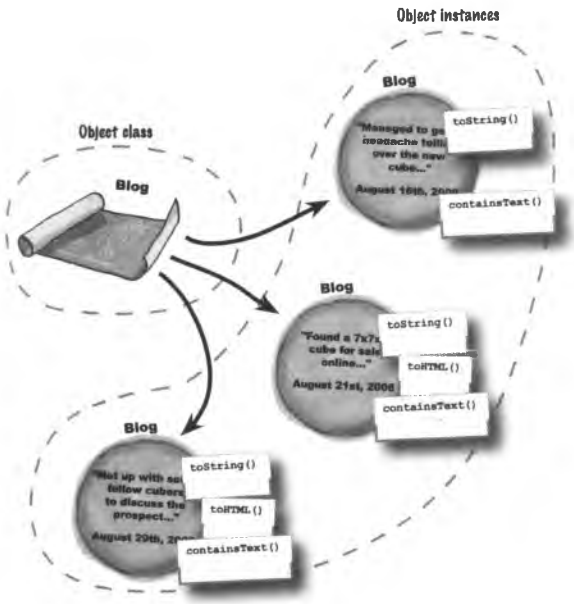


10

Специальные объекты

Работа со специальными объектами

Если бы все было так легко, мы бы, конечно, так и сделали. JavaScript не гарантирует возврат денег, но вы действительно можете делать с ним все, что захотите. Специальные объекты — это эквивалент тройного эспрессо с сахаром и корицей. Вот такая специальная чашка кофе! Точно так же в специальных объектах вы можете смешивать код, добиваясь именно того результата, который вам нужен, и пользуясь преимуществами свойств и методов. И в конце получается объектно-ориентированный код, расширяющий язык JavaScript... только для вас!

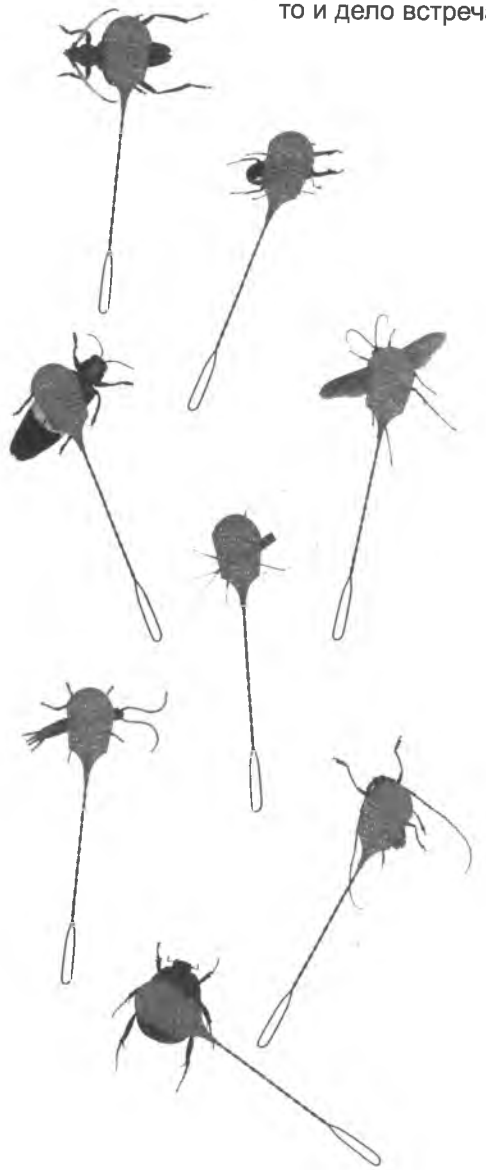


11

Охота на ошибки

Когда сценарий не работает

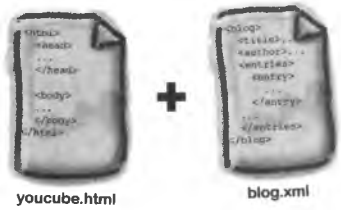
Даже самые лучшие планы в JavaScript иногда не реализуются. И когда это происходит, главное — не паниковать. Лучшие программисты не те, которые никогда не делали ошибок, — на самом деле это просто лгуны. Лучшие — это те, кто может **успешно обнаружить и устранить** ошибку. Отладчики высокой квалификации **нарабатывают хорошую манеру написания кода,** минимизирующую вероятность появления неприятных ошибок. ***Лучше предотвратить, чем потом бороться.*** Тем не менее ошибки то и дело встречаются, и вам нужен арсенал средств борьбы с ними...



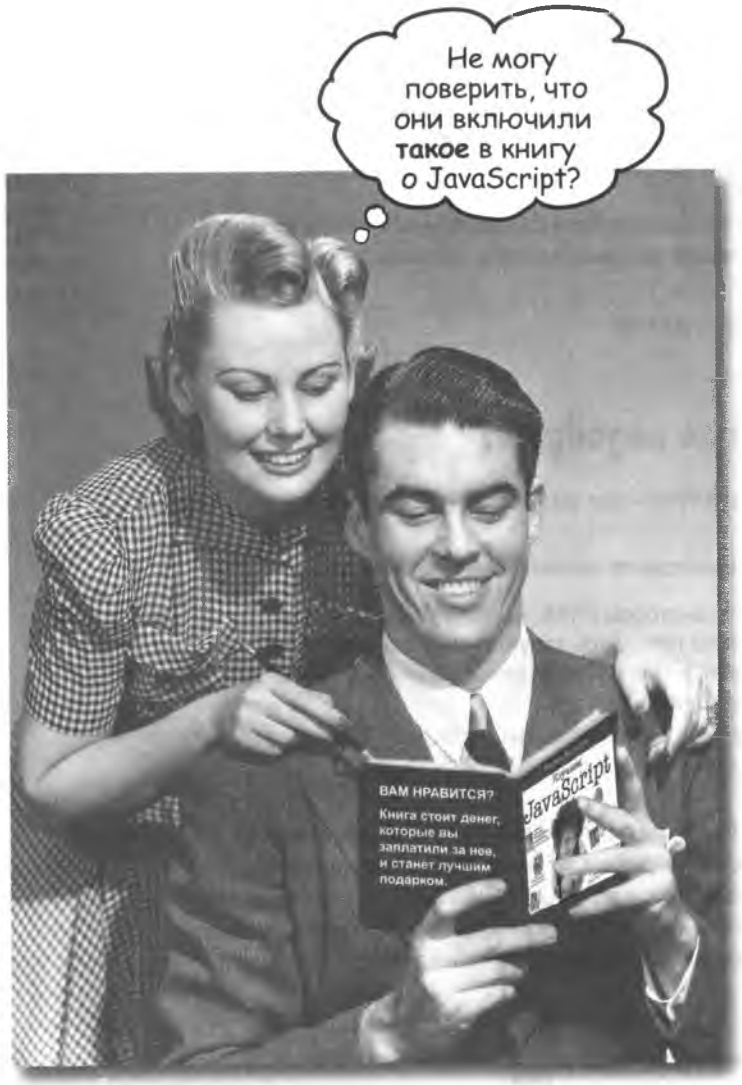
12 Динамические данные

Удобные веб-приложения

Современный Интернет очень отзывчив, страницы умеют реагировать на каждый каприз пользователя. Именно об этом мечтают многие разработчики. JavaScript играет важную роль в осуществлении этой мечты при помощи технологии **Ajax**, позволяющей эффективно менять «чувствительность» страниц. Благодаря Ajax страницы научились **быстро загружаться и динамически сохранять данные, отвечая на действия пользователя в реальном времени** без необходимости перезагрузки браузера.



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...

1. Имеете ли вы доступ к компьютеру с браузером, текстовым редактором и выходом в Интернет?
2. Хотите ли ли вы **научиться создавать веб-страницы, превращающие работу в Интернете в по-настоящему интерактивный опыт?**

   Вы предпочитаете оживленную беседу сухим, скучным академическим лекциям?

С нашей помощью вы научитесь писать на языке JavaScript код, заставляющий страницы делать множество потрясающих вещей, невозможных при использовании только HTML.

...то эта книга для вас.

Кому эта книга не подойдет?

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

1. Вы **никогда не** создавали веб-страниц?

   (Быть знатоком HTML не обязательно, но вы должны понимать, какую роль в появлении страниц играют HTML и CSS и как опубликовать страницу в Интернете.)
2. Считаете себя мастером написания сценариев и ищете **справочник** по JavaScript?
3. Вы **боитесь попробовать что-нибудь новое**? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Считаете, что техническая книга, в которой компоненты Java изображены в виде человечков, серьезной быть не может?

...эта книга не для вас.

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

Мы знаем, о чем Вы думаете

«Разве серьезные книги по программированию на JavaScript такие?»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь научиться?»

И мы знаем, что думает Ваш мозг

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

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

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

Активизируются нейроны. Вспыхивают эмоции. Происходят химические реакции.

И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

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

И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь *важное*. На тигров, например. Или на то, что к огню лучше не прикасаться. Или на то, что вам не следовало соглашаться на просьбу друга посидеть с его домашней анакондой.

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

Эта книга для тех, кто хочет учиться.

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

Основные принципы серии «Head First»

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

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

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

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я 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, сентименты 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 задумываетесь. 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 изучить паттерны проектирования, и по возможности быстрее. Вы хотите *запомнить* прочитанное и *применять* новую информацию на практике. 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 JavaScript так же важна, как и голодный тигр?

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

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

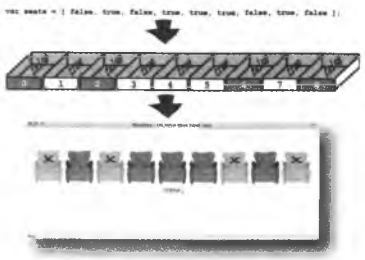
Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно не интересует, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует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

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ:

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

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



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

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

Так как ***проделанное*** запоминается намного лучше *прочитанного*, в книге вы найдете более 80 ***упражнений***. Надеемся, они заставят вас испытать победное чувство «я смог это сделать!».

Мы совместили ***несколько стилей обучения***, потому что одни читатели любят 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 имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

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

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

В историях, примерах, картинках «живут» ***люди***. Ведь вы человек. И ваш мозг больше внимания уделяет *людям*, а не *вещам*.

Вырежьте и прикрепите на холодильник.

Что сможете 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.

1. **Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.**
Не просто *читайте*. Обдумывайте. Натыкаясь на вопрос, не читайте ответ сразу. Представьте, что вам его задал человек. Чем больше вы заставляете мозг думать, тем больше информации вы поймете и запомните.

2. **Выполняйте упражнения, делайте заметки.**
Делать упражнения за вас означает выполнять вашу работу. Упражнения нужно не просто *читать*. **Возьмите карандаш.** Вы быстрее усвоите материал, если будете записывать результаты своих размышлений.

3. **Читайте раздел «Часто задаваемые вопросы».**
Это не просто вкладки с факультативной информацией — это **часть основного материала**! Не пропускайте их.

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

5. **Пейте воду. И побольше воды.**
Мозг лучше всего работает при избытке жидкости. Обезвоживание уменьшает способность к познанию.

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.**
Старайтесь понять, не перегружен ли мозг. Если только что прочитанное сразу забывается, явно пора на отдых. Пытаясь выучить сразу слишком много, вы не ускорите процесс усвоения материала, а, наоборот, замедлите его.

8. **Пусть это станет реальностью!**
Представляйте себя героем историй. Делайте собственные подписи к картинкам. *Лучше* хихикать над плохой шуткой, чем оставаться равнодушным.

9. **Просто работайте!**
Научиться программировать можно только одним способом: **писать код**. Именно этим вам и предстоит заняться. Не пропускайте упражнения — обучение происходит в процессе решения задач, даже таких необычных, как «Приключения нарисованного человечка», поиск мест в кинотеатре для настоящих мачо или заполнение блога YouCube. Не переходите к следующим страницам, не закончив упражнений. И, если вам доведется поработать над реальным проектом, не забывайте использовать приемы, описанные в книге.

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 даем сведения, которые вам действительно требуются.

Если вы по другим источникам изучаете историю развития JavaScript, продолжайте это делать, так как данная книга вам не поможет. Она призвана научить вас решать практические, каждодневные задачи по созданию интерактивных веб-страниц, с которыми пользователям будет приятно работать. Мы перешагиваем через формализм и рассматриваем только те понятия из JavaScript, которые действительно потребуются вам для работы.

Здесь не рассматриваются все нюансы языка JavaScript.

Разумеется, можно было бы описать все операторы, события, объекты и ключевые слова JavaScript, но было решено ограничиться более компактным изданием, которое удобно иметь под рукой. Поэтому основной упор дается на концепции, которые используются программистами в 95 % случаев. Чтобы в результате вы получили способность самостоятельно писать сложные сценарии.

Существует большая библиотека уже готовых фрагментов кода JavaScript, и поэтому крайне важно понимать, когда следует писать собственный вариант функции или метода, а когда можно ограничиться стандартным. Слово «специальный» в этой книге означает, что код должен быть написан вами лично, а не взят из библиотеки JavaScript.

В процессе чтения желательно пользоваться разными браузерами.

Несмотря на тот факт, что все современные браузеры поддерживают JavaScript, имеются небольшие различия в процедуре обработки кода сценариев. Именно поэтому желательно проверять результаты своей работы по крайней мере в двух браузерах. Известно, что лучше всех с обработками ошибок справляется Firefox. Но не стесняйтесь попросить друзей и знакомых протестировать ваши сценарии и в их браузерах также.

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.**

Повторения применяются 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 кода были сделаны по возможности компактными.

Наши читатели не любят просматривать по 200 строк кода, чтобы найти две нужные строки. Большинство примеров книги приводится в минимальном контексте, чтобы та часть, которую вы непосредственно изучаете, была понятной и простой. Не ждите, что весь код будет стопроцентно устойчивым или даже просто завершенным — примеры написаны в учебных целях и не всегда являются полнофункциональными.

Все варианты кода из нашей книги помещены в Интернет, чтобы дать вам возможность скопировать их к себе и исследовать. Скачать их можно по адресу

http://www.headfirstlabs.com/books/hfjs/

Упражнения «Мозговой штурм» не имеют ответов.

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

Технические редакторы

Алекс Ли – студент Хьюстонского университета, специализирующийся на автоматизированных системах управления. Обожает бег, компьютерные игры и изучение новых языков программирования.

Ти Ви Сканнел из города Систерс, штат Орегон, занимается программированием с 1995 года. Разработчик каркаса Ruby on Rails.

Элейн Нельсон занимается разработкой веб-сайтов около 10 лет. Как она говорит своей матери, ученая степень в английском языке много где может пригодиться. Узнать о текущих увлечениях Элейн вы можете на ее сайте – elainenelson.org.

Флетчер Мур является веб-разработчиком и дизайнером в институте Georgia Tech. Увлекается велоспортом, музыкой, садоводством и является фанатом бейсбольной команды Red Sox. Проживает в Атланте с женой Катариной, дочерью Сэйлор и сыном Сэтчелом.

Энтони Ти Холденер III является разработчиком веб-приложений и автором книги Ajax: The Definitive Guide, также вышедшей в издательстве O'Reilly.

Захарий Кессин занимается веб-программированием около 15 лет. Проживает в Израиле с женой и тремя детьми.

Катерина Сент-Джон – доцент кафедры информатики и математики в университете города Нью-Йорк, занимается исследованием в области вычислительной биологии и случайных структур.

Стивен Таллент живет и работает в городе Нэшвилле штата Теннеси, разрабатывая спортивные приложения и воспитывая маленьких детей. Кроме того, он увлекается катанием на роликовой доске и кулинарией. И даже готовится сделать вторую карьеру в качестве повара в буфете.

Благодарности

Моему редактору:

Помните, в начальной школе нам предоставляли возможность переписываться с детьми из других городов, обмениваясь информацией о своей жизни? Именно таким другом по переписке стала для меня **Катрин Нолан** с момента начала этого проекта. Мы общались по телефону, электронной почте, факсу. В процессе этого общения Катрин стала больше чем коллегой по работе над изданием. Она стала моим другом. И часто «деловые» звонки заканчивались переходом от разговоров про JavaScript к обсуждению других наших увлечений. Мы оба получили удовольствие от работы и от ее конечного результата. Спасибо, Катрин. Я помню, что задолжал тебе несколько мартини.

Катрин Нолан, поклонница десятичной системы счисления.

Команда издательства O'Reilly:

Сложно подобрать слова для команды Head First. Но я попробую.

Бретт МакЛафлин с момента моего появления в лагере новобранцев Head First заставил меня сконцентрироваться. Этот парень одинаково серьезно относится как к необходимости анализа сценариев в процессе обучения, так и к игре на гитаре. Я уверен, что даже ко сну он отходит, предварительно задав себе вопрос: «Зачем я это делаю?» Но именно его вклад помог создать такую выдающуюся книгу. Спасибо, Бретт!

Бретт МакЛафлин лидер команды Head First и поклонник блюзов.

Лу Барр, богиня дизайна.

Лу Барр стала еще одним моим другом по переписке. Мне кажется, она спустилась к нам откуда-то с дизайнерского Олимпа. Без нее мы никогда не получили бы столь прекрасно сверстанной книги.

Вряд ли процесс работы над книгой протекал бы столь гладко без **Сандерса Кляйфелда**. Именно он нашел выход из многих сложных ситуаций.

Я не забыл и об остальных членах команды O'Reilly. **Лари Петриски** поверила в меня настолько, чтобы запустить данный проект, **Кетрин МакКаллох** обеспечила поддержку сайта (www.headfirstlabs.com), а **Кейт МакНамара** с удивительной точностью заполнила все пробелы. Спасибо, ребята!

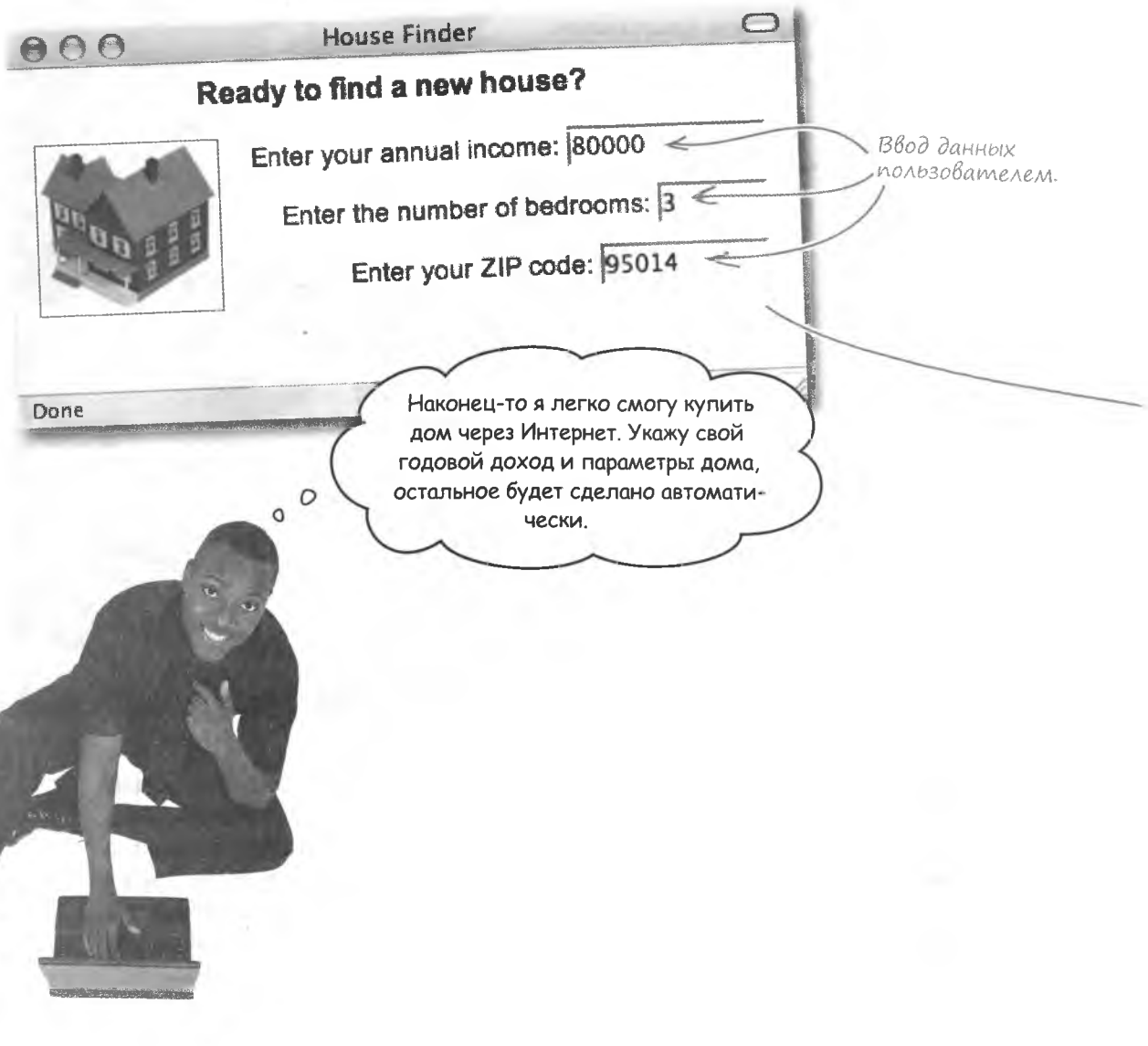
Ну и наконец, самой теплой благодарности заслужили, наверное, **Кетти Сьерра** и **Берт Байтс** за их потрясающее видение всех серий Head First. Работа в этой команде была для меня счастьем...

Реакции виртуального мира

Устали представлять Интернет набором пассивных страниц? Кто из нас не держал в руках книг. Их читаешь, в них находишь информацию. Но они не **интерактивны**. Как и интернет-страницы без JavaScript. Без сомнения, отправить данные формы и проделать другие трюки можно и при помощи кода HTML и CSS, но реальная **интерактивность** требует **более умного подхода** и большей работы... зато и результат впечатляет ***намного больше***.

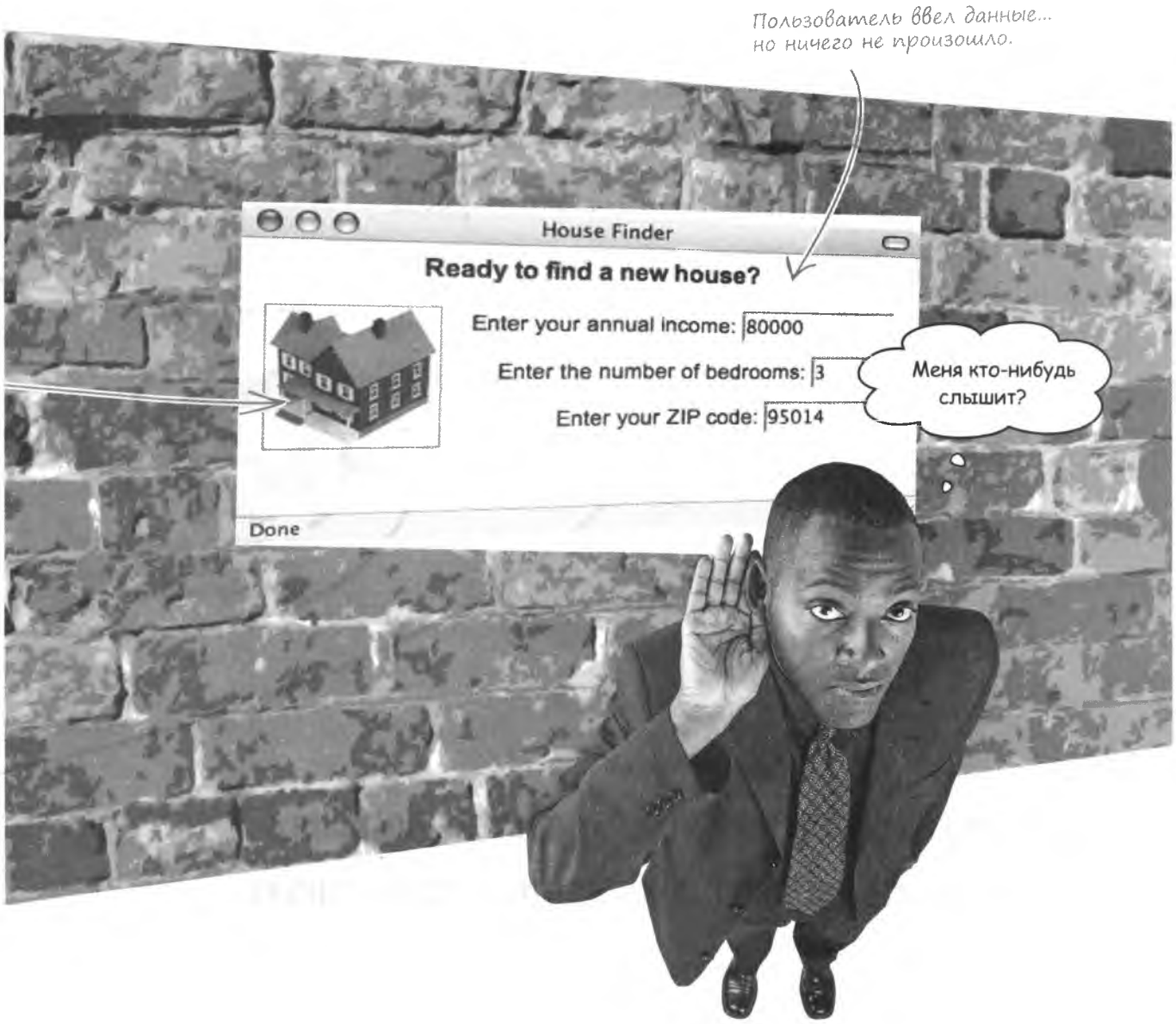
То, что нужно людям

Мы знаем, что Интернет – это виртуальная реальность, но пользуются им вполне реальные люди, с реальными нуждами. Им требуются убойные рецепты мясного рулета, возможность скачать любимую песню или даже купить новый дом. К счастью, когда дело доходит до ваших нужд, сеть ведет себя по-разному!



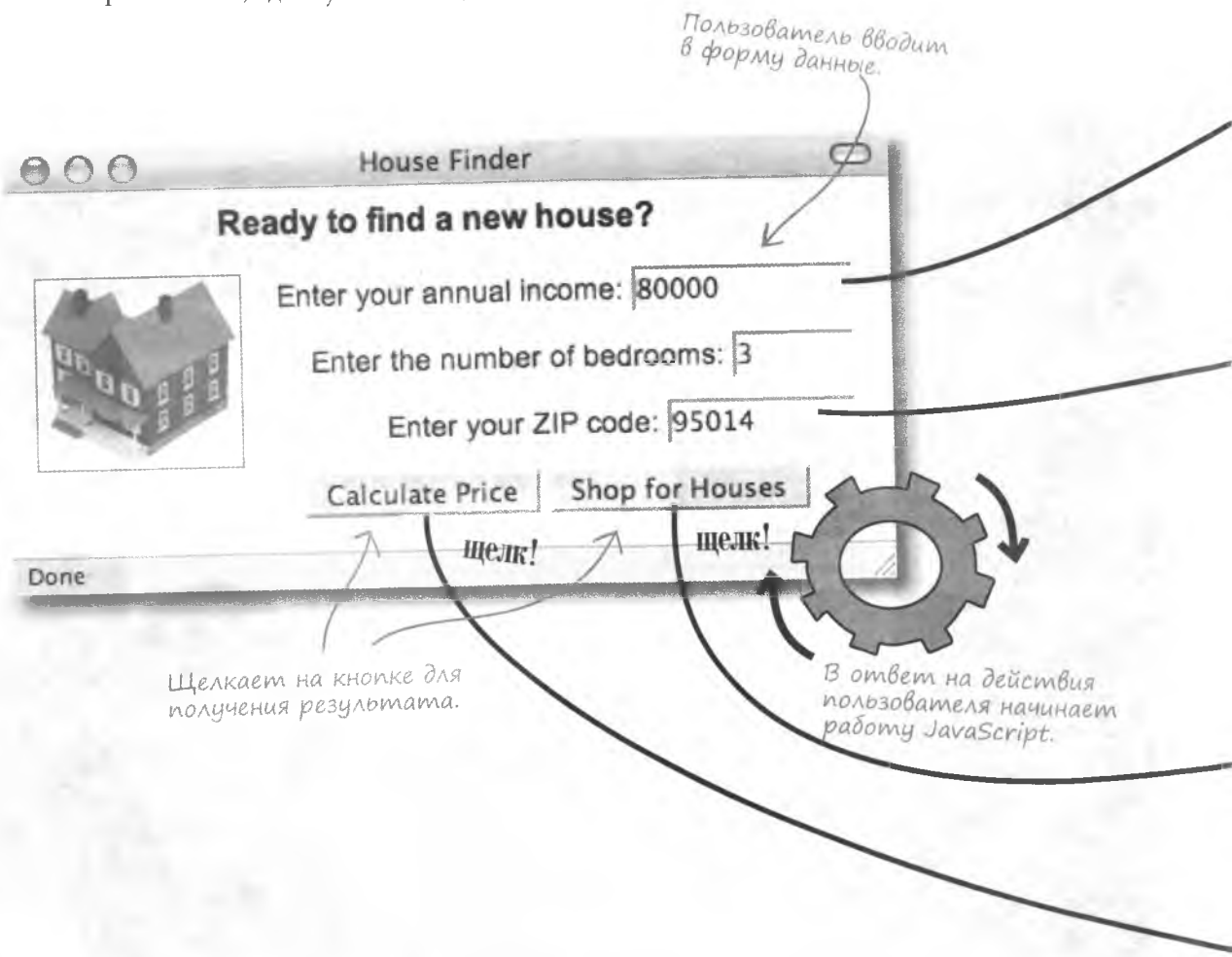
И ничего... как будто говоришь со стенкой

Сеть не всегда реагирует так, как вам хотелось бы. Более того, зачастую она кажется совершенно равнодушной, отдельной от внешнего мира и никак не отвечающей нуждам многочисленных пользователей. Вы ожидаете реакции на введенные вами данные... но ничего не происходит. Не принимайте это близко к сердцу, статический Интернет по-другому просто не умеет.

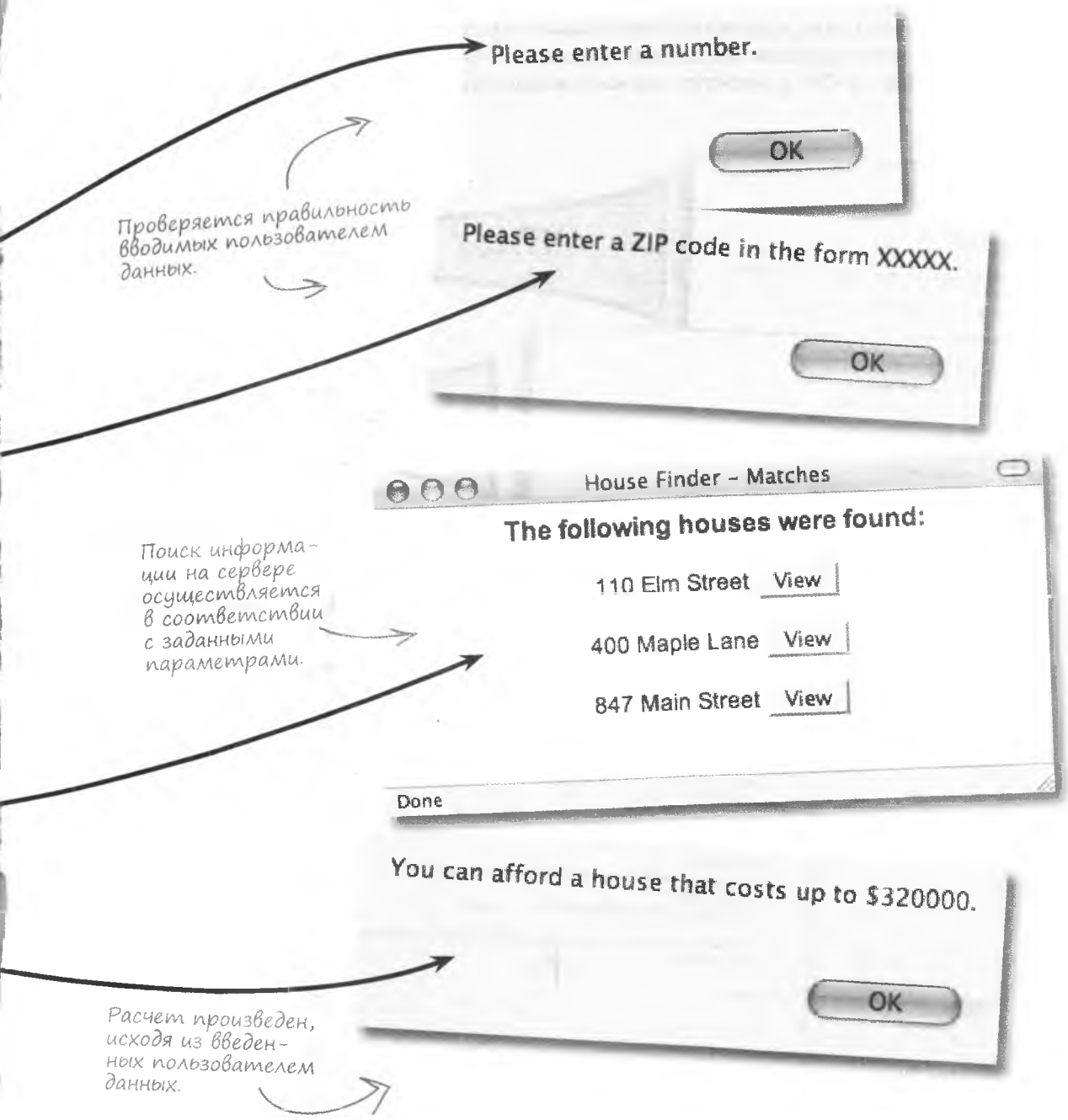


A JavaScript отвечает

Язык JavaScript подобен выключателю, переводящему страницу в интерактивный режим. Он активирует функции, которые прислушиваются к нуждам пользователей, обрабатывают вводимые данные и отвечают на запросы. Возможно, это некоторое преувеличение, но именно JavaScript позволяет превратить веб-страницу в интерактивное приложение, вдохнув в нее жизнь!

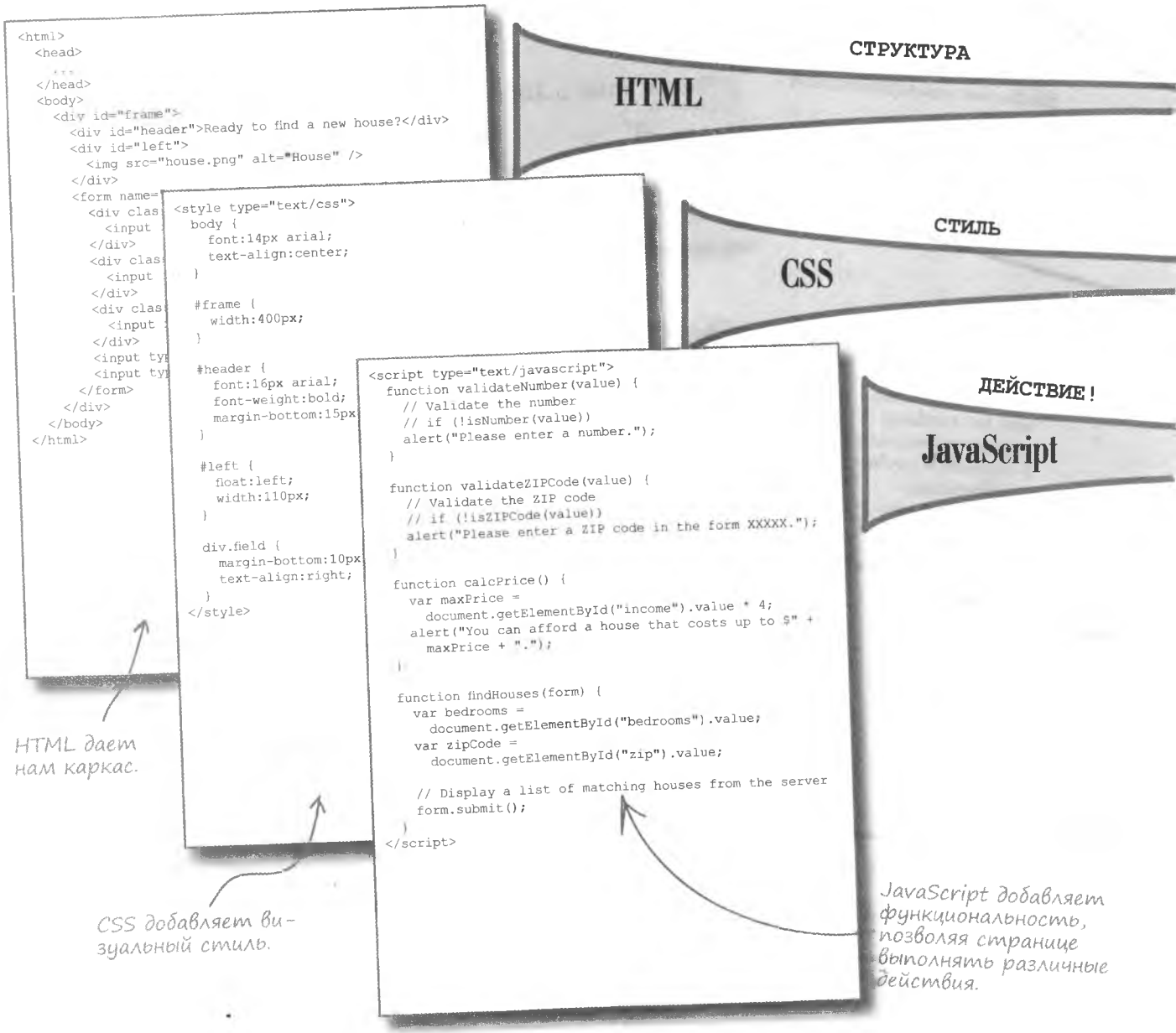


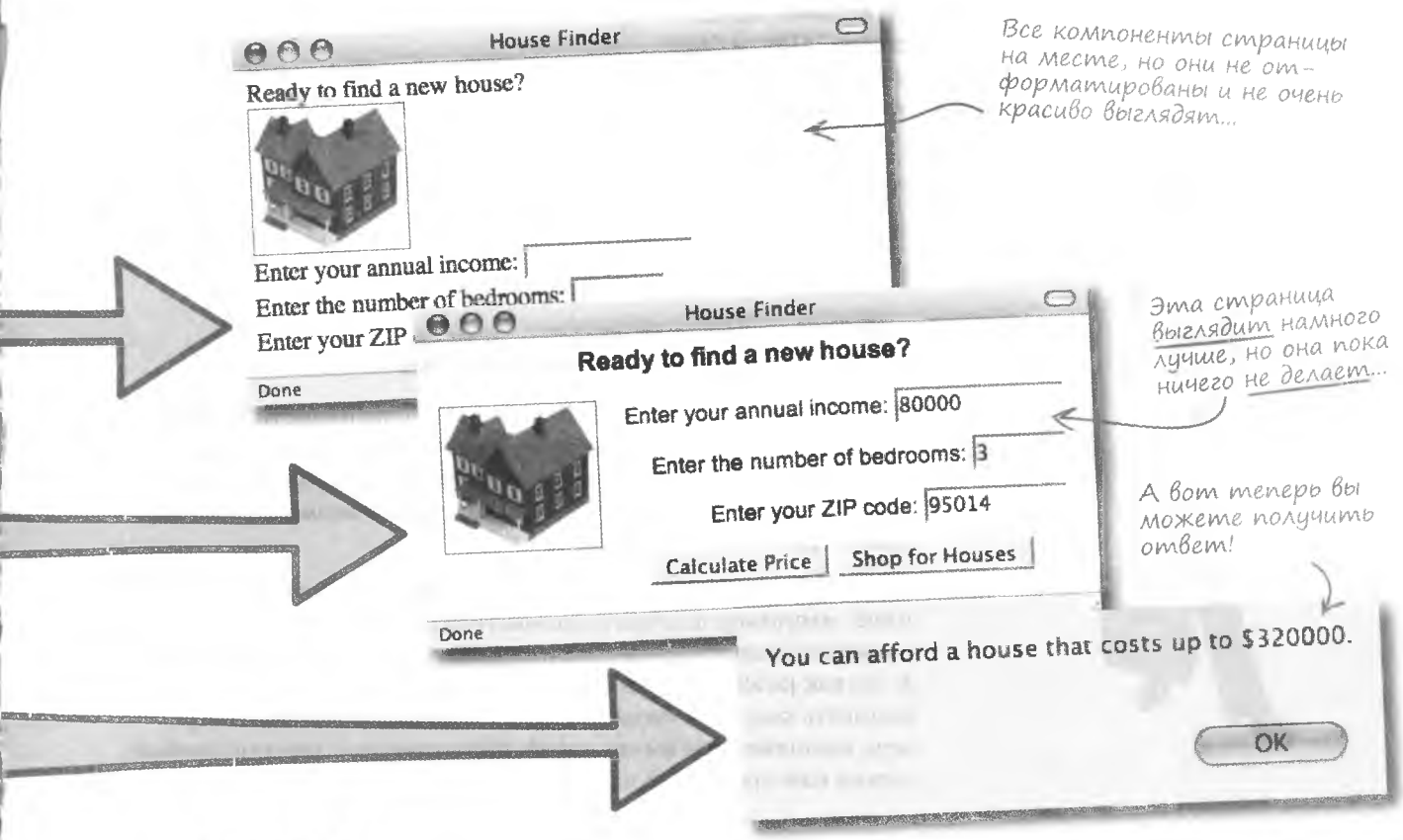
JavaScript оживляет веб-страницы, позволяя им отвечать на ввод пользовательских данных.


Please enter a number.
OK
Проверяется правильность вводимых пользователем данных.
Please enter a ZIP code in the form XXXXX.
OK
House Finder – Matches
The following houses were found:
110 Elm Street View
400 Maple Lane View
847 Main Street View
Done
Поиск информации на сервере осуществляется в соответствии с заданными параметрами.
You can afford a house that costs up to $320000.
OK
Расчет произведен, исходя из введенных пользователем данных.

Свет, камера, взаимодействие!

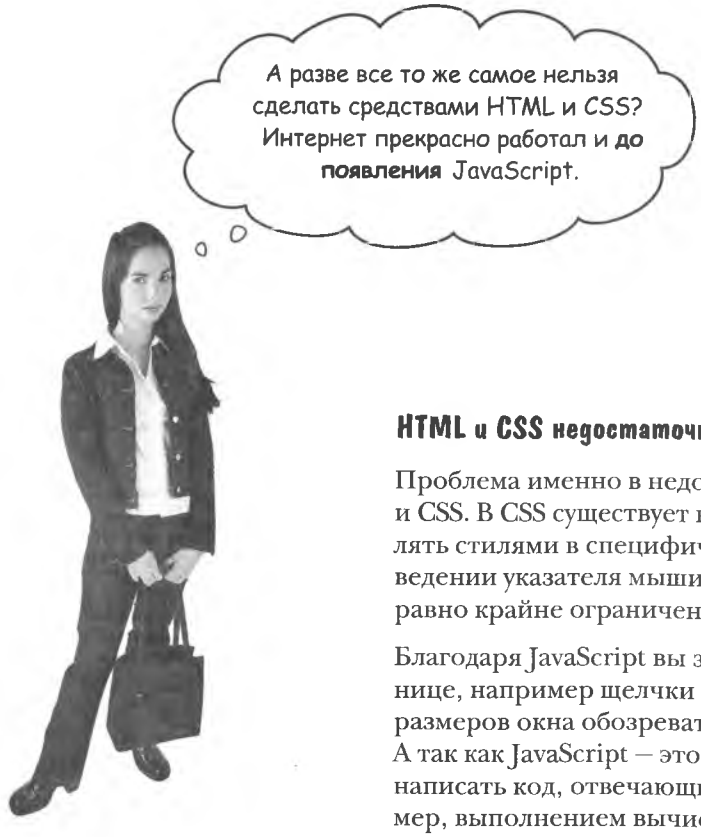
HTML, CSS и JavaScript — это три кита современного конструирования веб-страниц. HTML обеспечивает структуру, CSS добавляет стиль, а JavaScript обеспечивает «сцепление с дорогой». Чтобы пройти путь к интерактивным веб-страницам, вы должны следовать от структуры (HTML) в стиле (CSS) к действию (JavaScript). Как и в CSS, в JavaScript код часто находится непосредственно внутри веб-страницы.





Спасибо, JavaScript! Я вот-вот найду холостяцкую берлогу, о которой давно мечтал.

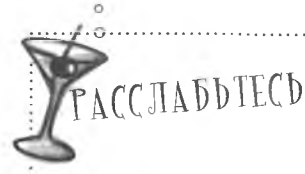
JavaScript начинает работу, как только пользователь просит страницу выполнить какую-либо задачу.



HTML и CSS недостаточно интерактивны

Проблема именно в недостаточной интерактивности HTML и CSS. В CSS существует набор приемов, позволяющих управлять стилями в специфических ситуациях, например при наведении указателя мыши на ссылки, но ваши возможности все равно крайне ограничены.

Благодаря JavaScript вы замечаете все происходящее на странице, например щелчки пользователя на кнопках, изменение размеров окна обозревателя или ввод данных в текстовое поле. А так как JavaScript — это язык написания сценариев, вы можете написать код, отвечающий на действия пользователя, например, выполнением вычислений, динамической заменой изображения или проверкой данных.



Не беспокойтесь о деталях.

JavaScript позволяет очень многое, но вы пока в самом начале знакомства с этим языком. Смею вас заверить, что события, функции и многие другие элементы JavaScript со временем станут для вас понятными. Кроме того, есть вероятность, что вы уже знаете намного больше, чем вам кажется.

HTML + CSS + JavaScript = РЕАЛЬНАЯ интерактивность

Возьми в руку карандаш

Вы уже знаете больше, чем вам кажется. Посмотрите на код для страницы House Finder и напишите, что делает каждый из выделенных фрагментов кода. Не бойтесь строить догадки.

```
<html>
  <head>
    <title>Поиск домов</title>
    <script type="text/javascript">
      function validateNumber(value) {
        // Проверка ввода числа
        // if (!isNumber(value))
        alert("Пожалуйста, введите число.");
      }

      function validateZIPCode(value) {
        // Проверка ввода индекса
        // if (!isZIPCode(value))
        alert("Пожалуйста, введите индекс в формате XXXXX.");
      }

      function calcPrice() {
        var maxPrice = document.getElementById("income").value * 4;
        alert("Вы можете позволить дом стоимостью до $" + maxPrice + ".");
      }

      function findHouses(form) {
        var bedrooms = document.getElementById("bedrooms").value;
        var zipCode = document.getElementById("zip").value;

        // Отображение списка подходящих домов с сервера
        form.submit();
      }
    </script>
  </head>

  <body>
    <div id="frame">
      <div id="header">Готовы к поиску нового дома?</div>
      <div id="left">
        <img src="house.png" alt="House" />
      </div>
      <form name="orderform" action="…" method="POST">
        <div class="field">Укажите ваш годовой доход:
          <input id="income" type="text" size="12"
          onblur="validateNumber(this.value)"/></div>
        <div class="field">Введите число спален:
          <input id="bedrooms" type="text" size="6"
          onblur="validateNumber(this.value)"/></div>
        <div class="field">Введите индекс:
          <input id="zip" type="text" size="10"
          onblur="validateZIPCode(this.value)"/></div>
        <input type="button" value="Вычислить цену"
          onclick="calcPrice();" />
        <input type="button" value="Купить дом"
          onclick="findHouses(this.form);" />
      </form>
    </div>
  </body>
</html>
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Возьми в руку карандаш

Решение

Вот какую функцию выполняют выделенные фрагменты кода. Надеемся, ваши догадки совпали с правильными ответами.

```
<html>
  <head>
    <title>Поиск домов</title>
    <script type="text/javascript">
      function validateNumber(value) {
        // Проверка ввода числа
        // if (!isNumber(value))
        alert("Пожалуйста, введите число.");
      }

      function validateZIPCode(value) {
        // Проверка ввода индекса
        // if (!isZIPCode(value))
        alert("Пожалуйста, введите индекс в формате XXXXX.");
      }

      function calcPrice() {
        var maxPrice = document.getElementById("income").value * 4;
        alert("Вы можете позволить дом стоимостью до $" + maxPrice + ".");
      }

      function findHouses(form) {
        var bedrooms = document.getElementById("bedrooms").value;
        var zipCode = document.getElementById("zip").value;

        // Отображение списка подходящих домов с сервера
        form.submit();
      }
    </script>
  </head>

  <body>
    <div id="frame">
      <div id="header">Готовы к поиску нового дома?</div>
      <div id="left">
        <img src="house.png" alt="House" />
      </div>
      <form name="orderform" action="..." method="POST">
        <div class="field">Укажите ваш годовой доход:
          <input id="income" type="text" size="12"
          onblur="validateNumber(this.value)"/></div>
        <div class="field">Введите число спален:
          <input id="bedrooms" type="text" size="6"
          onblur="validateNumber(this.value)"/></div>
        <div class="field">Введите индекс:
          <input id="zip" type="text" size="10"
          onblur="validateZIPCode(this.value)"/></div>
        <input type="button" value="Вычислить цену"
          onclick="calcPrice();" />
        <input type="button" value="Купить дом"
          onclick="findHouses(this.form);" />
      </form>
    </div>
  </body>
</html>
```

Пользователя просят ввести индекс в формате XXXXX.

Вычисляет максимальную цену дома, умножая доход пользователя на четыре.

Проверяет, было ли введено число в поле income.

Значение в поле ввода ZIP code.

После щелчка на кнопке вычисляет максимальную цену дома.

Тег <script>

Поместим фрагмент на языке JavaScript непосредственно в HTML-код, как было показано на предыдущей странице. Первым делом вы должны дать понять браузеру, что он будет иметь дело с JavaScript... и здесь вам на помощь придет тег <script>.

Этот тег добавляется в произвольное место HTML-кода, но обычно его помещают внутрь заголовка. Вот таким образом:

```
<html>
  <head>
    <title>House Finder</title>

    <script type="text/javascript">

      function validateNumber(value) {
        // Проверка ввода числа
        // if (!isNumber(value))
        alert("Please enter a number.");
      }

    </script>
  </head>

  <body>
    <!-- Остальная часть HTML-кода -->
  </body>
</html>
```

Этот тег говорит о том, что внутри находится некий сценарий...

...в данном случае написанный на языке JavaScript.

Тег script на HTML-странице обычно помещается внутрь тега head.

Фрагмент между открывающим и закрывающим тегами script написан на языке JavaScript... браузер понимает, что это уже не HTML, а язык написания сценариев.

Закрывающий тег script указывает браузеру, что дальше опять начинается обычный HTML-код.

Часто Задаваемые Вопросы

В: То есть все, что я помещу между тегами <script>, будет относиться к JavaScript?

О: Не обязательно... тег <script> говорит браузеру, что начинается сценарий, но он может быть и на другом языке. Язык сценария определяется в части type="text/javascript".

В: То есть я могу использовать и другие языки сценариев?

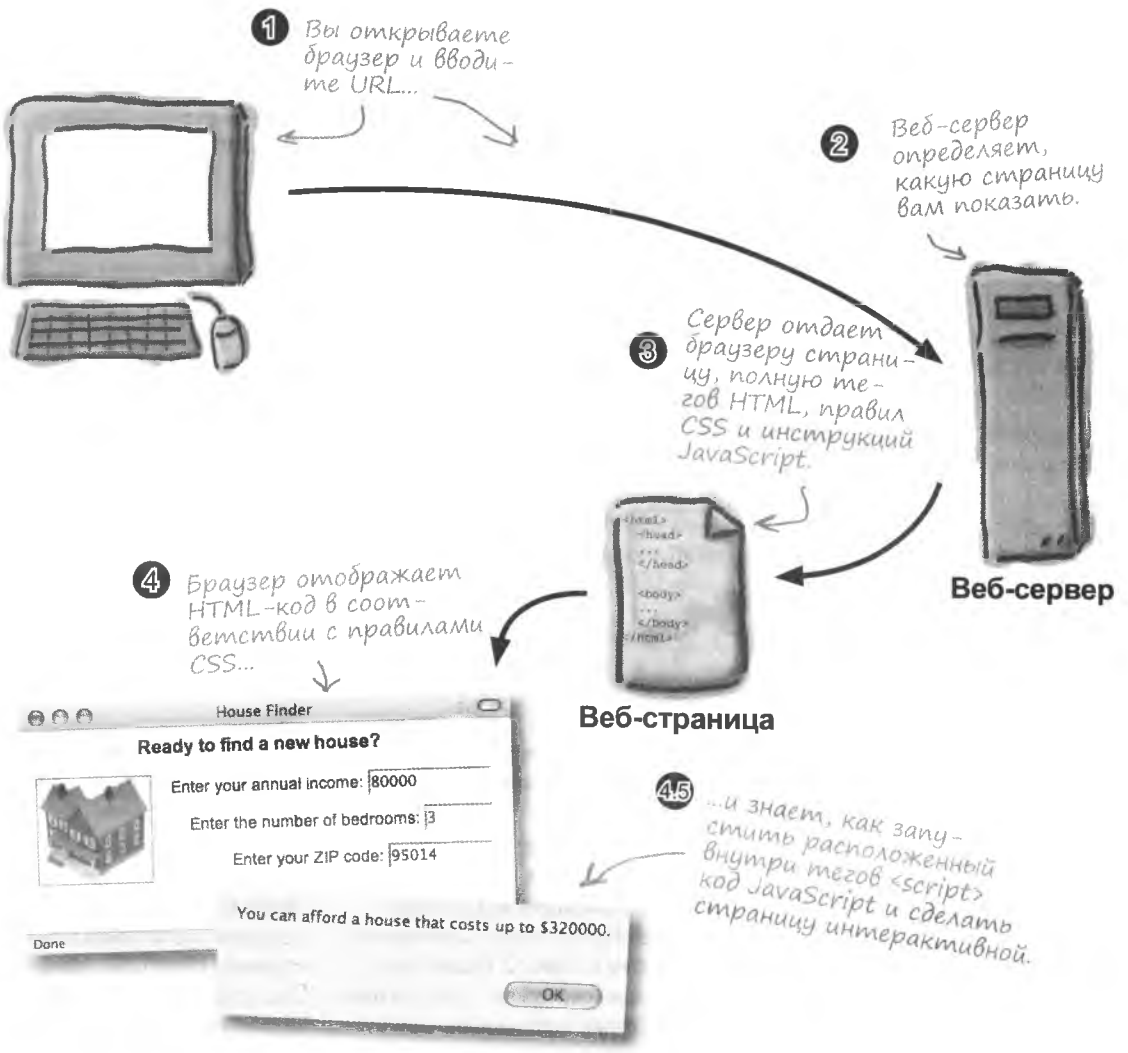
О: Конечно. Microsoft предоставляет вам VBScript (версия языка Visual Basic) и разновидность Ajax, которая называется ASP.NET AJAX. О последней мы поговорим в главе 12. Существуют и другие языки сценариев. Но в этой книге мы всегда будем использовать text/javascript.

В: Должны ли элементы <script> располагаться внутри тегов <head>?

О: Поместить теги <script> можно куда угодно... но их расположение вне заголовка считается дурным тоном. Это все равно что вставить в середину веб-страницы стили CSS... фрагменты на JavaScript лучше выделять в отдельную группу, и для этого прекрасно подходит часть страницы, ограниченная тегами <head>.

Ваш браузер понимает HTML, CSS и JavaScript

Как вы знаете, браузеры умеют отображать HTML-код. На языке CSS мы объясняем браузерам, как именно показывать различные части HTML. Соответственно, язык JavaScript — это всего лишь еще один способ вашего общения с браузером... Но на этот раз вы уже не указываете способ отображения, а подаете команды.



Часто Задаваемые Вопросы

В: Каким образом браузеры запускают код JavaScript?

О: В браузеры встроено специальное программное обеспечение, называемое интерпретатором JavaScript. Именно он запускает написанный на этом языке код. Именно поэтому JavaScript называют *интерпретируемым* языком программирования, в отличие от *транслируемого*. Транслируемые языки, например C++ или C#, необходимо сначала преобразовать в исполняемый файл при помощи компилятора. Программам на JavaScript этого не требуются, так как они интерпретируются непосредственно браузером.

В: Как заставить веб-страницу запустить код JavaScript?

О: В большинстве случаев такой код запускается после какого-то действия, например загрузки страницы или щелчка пользователя на кнопке. Включение кода JavaScript после каких-то действий производится при помощи механизма, называемого «событием».

В: Насколько безопасно работать с JavaScript?

О: По большей части безопасно. JavaScript исходно разработан таким образом, чтобы затруднить выполнение вредоносного кода. К примеру, средствами JavaScript невозможно осуществлять чтение и запись файлов на жестком диске пользователя. Это ограничивает возможность применения большинства вирусов. Разумеется, можно написать вредоносный код и на JavaScript. В прошлом, из-за конструктивных недостатков браузеров хакеры нашли ряд дыр в безопасности JavaScript, так что назвать этот язык совершенно надежным, увы, не получится.

В: Тег `<script>` в программе House Finder относится к HTML или к JavaScript?

О: Сам по себе тег `<script>` принадлежит HTML и предназначен для встройки сценариев в код веб-страниц. **Внутри** тега `<script>` вы видите код JavaScript. Так как сам тег разработан для поддержки нескольких языков написания сценариев, вы указываете, на каком языке будет написан ваш сценарий, при помощи атрибута `type`.

В: Кажется, я встречал интерактивные страницы, к примеру, с формами, проверяющими корректность введенных данных, созданные без JavaScript. Такое возможно?

О: Конечно. Сделать страницу интерактивной можно и без JavaScript, но в большинстве случаев такая реализация будет неэффективной и тяжеловесной. К примеру, проверка корректности введенных данных может быть осуществлена на сервере при отправке формы. При этом вы ждете ответа сервера в виде новой страницы. С таким же успехом для проверки можно воспользоваться карандашом и бумагой! JavaScript позволяет обойтись без загрузки новых страниц и ненужной передачи данных на сервер и обратно. А многие функции, реализуемые при помощи JavaScript, альтернативно могут быть реализованы только при помощи сторонних встроенных программ для браузеров.

Упражнение

Определите, является ли фрагмент кода стандартным выражением JavaScript или же это выражение введено программистами, которые написали страницу House Finder (вариант Custom):

`alert`	JavaScript / Custom	`onblur`	JavaScript / Custom
`calcPrice`	JavaScript / Custom	`onclick`	JavaScript / Custom
`zipCode`	JavaScript / Custom	`findHouses`	JavaScript / Custom
`var`	JavaScript / Custom	`value`	JavaScript / Custom

Упражнение Решение

Итак, какие же куски кода являются стандартными выражениями языка JavaScript, а какие были введены программистами:

```
<head>
  <title>Поиск домов</title>
  <script type="text/javascript">
    function validateNumber(value) {
      // Проверка ввода числа
      // if (!isNumber(value))
      alert("Пожалуйста, введите число.");
    }

    function validateZIPCode(value) {
      // Проверка ввода индекса
      // if (!isZIPCode(value))
      alert("Пожалуйста, введите индекс в формате XXXXX.");
    }

    function calcPrice() {
      var maxPrice = document.getElementById("income").value * 4;
      alert("Вы можете позволить дом стоимостью до $" + maxPrice
+ ".");
    }

    function findHouses(form) {
      var bedrooms = document.getElementById("bedrooms").value;
      var zipCode = document.getElementById("zip").value;

      // Отображение списка подходящих домов с сервера
      form.submit();
    }
  </script>
</head>

<body>
  <div id="frame">
    <div id="header">Готовы к поиску нового дома?</div>
    <div id="left">
      <img src="house.png" alt="House" />
    </div>
    <form name="orderform" action="…" method="POST">
      <div class="field">Укажите ваш годовой доход:
        <input id="income" type="text" size="12"
        onblur="validateNumber(this.value)"/></div>
      <div class="field">Введите число спален:
        <input id="bedrooms" type="text" size="6"
        onblur="validateNumber(this.value)"/></div>
      <div class="field">Введите индекс:
        <input id="zip" type="text" size="10"
        onblur="validateZIPCode(this.value)"/></div>
      <input type="button" value="Вычислить цену"
        onclick="calcPrice();" />
      <input type="button" value="Купить дом"
        onclick="findHouses(this.form);" />
    </form>
  </div>
</body>
</html>
```

Всплывающее окно, указывающее на некорректное число.

alert — JavaScript / Custom

Пользовательский кусок кода, вычисляющий стоимость дома.

calcPrice — JavaScript / Custom

Задает внешнее место хранения данных.

var — JavaScript / Custom

Пользовательский код поиска подходящих домов.

findHouses — JavaScript / Custom

zipCode — JavaScript / Custom

Место хранения введенного пользователем индекса.

onblur — JavaScript / Custom

Указывает, что пользователь перешел к следующему полю ввода.

value — JavaScript / Custom

Текущее значение поля ввода индекса.

onclick — JavaScript / Custom

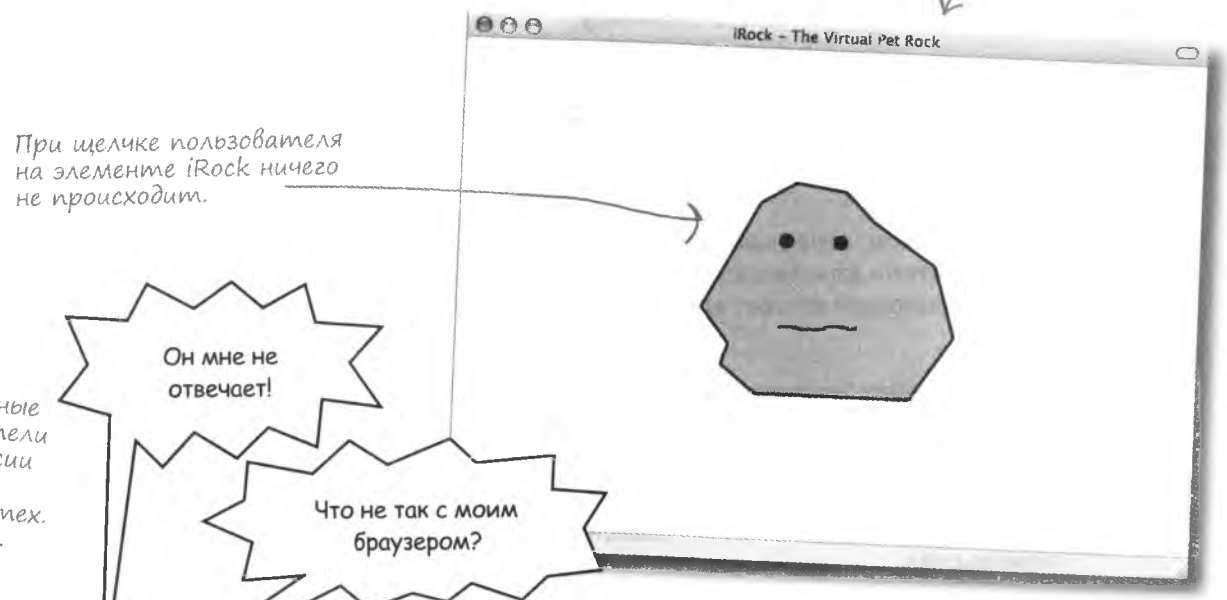
Указывает, что была нажата кнопка „Купить дом"

Помоги виртуальному другу человека

Свеженанятый успешный программист на HTML и CSS вызван к начальству для демонстрации последнего изобретения, которое называется iRock. Виртуальный домашний любимец наделал шума на всех конференциях игрушек, но пользователи недовольны его поведением.

Они щелкают на изображении и ждут чего-то необычного... Значит, вы должны **сделать iRock интерактивным** и прославиться... или кануть в небытие вместе с ним.

Перед вами iRock. Он состоит из кода HTML и CSS. И никак не взаимодействует с пользователями.



При щелчке пользователя на элементе iRock ничего не происходит.

Разгневанные пользователи бета-версии обрывают телефон тех. поддержки.

Что должен делать iRock, взаимодействуя с пользователями?

Сделайте iRock интерактивным

Вы не просто сделаете объект iRock интерактивным, но попутно познакомитесь с JavaScript. Вы быстро научите вашего домашнего любимца здороваться.

Перечислим список ваших задач.

1. **Создайте для iRock HTML-страницу.**

Вы уже знаете, как это сделать.

2. **Заставьте объект iRock приветствовать пользователей при загрузке страницы.**

Для вызова простого окна с сообщением используется метод alert.

3. **Напишите код, запрашивающий имя пользователя для личного приветствия и заставляющий объект улыбаться.**

Вы соединяете действия пользователя, например щелчок на изображении домашнего любимца...

4. **Добавьте обработчик события, запускающий написанный на шаге 3 код в ответ на щелчок на объекте.**

...с запрограммированными вами действиями.

5. **Заслужите одобрение и благодарность начальства.**

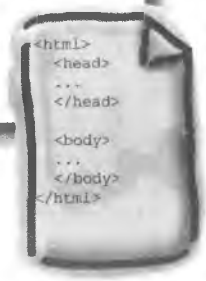
Веб-страница iRock

Более простую HTML-страницу, чем iRock, найти невозможно. Введите указанный ниже код в ваш любимый редактор и сохраните его под именем iRock.html. Нужные изображения скачайте с нашего сайта *http://www.headfirstlabs.com*.

HTML-страница вашего домашнего любимца так же уныла, как и он сам... Впрочем, это не наше дело, ведь начальник дал задание.

```
<html>
  <head>
    <title>iRock - Виртуальный любимец Rock</title>
  </head>

  <body>
    <div style="margin-top:100px; text-align:center">
      <img id="rockImg" src="rock.png" alt="iRock" />
    </div>
  </body>
</html>
```

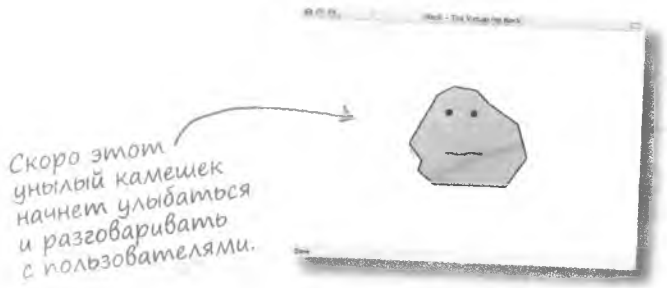
Не забудьте скачать изображение rock.png с сайта Head First Labs.



irock.html

Тестирование

Перед тем как двигаться дальше, сохраните и протестируйте страницу iRock в браузере. Убедитесь, что результат выглядит именно так, как показано на рисунке, так как все уже фактически готово, чтобы сделать его интерактивным при помощи JavaScript.



Часто Задаваемые Вопросы

В: Внутри тега находится CSS?

О: Да. Вы угадали.

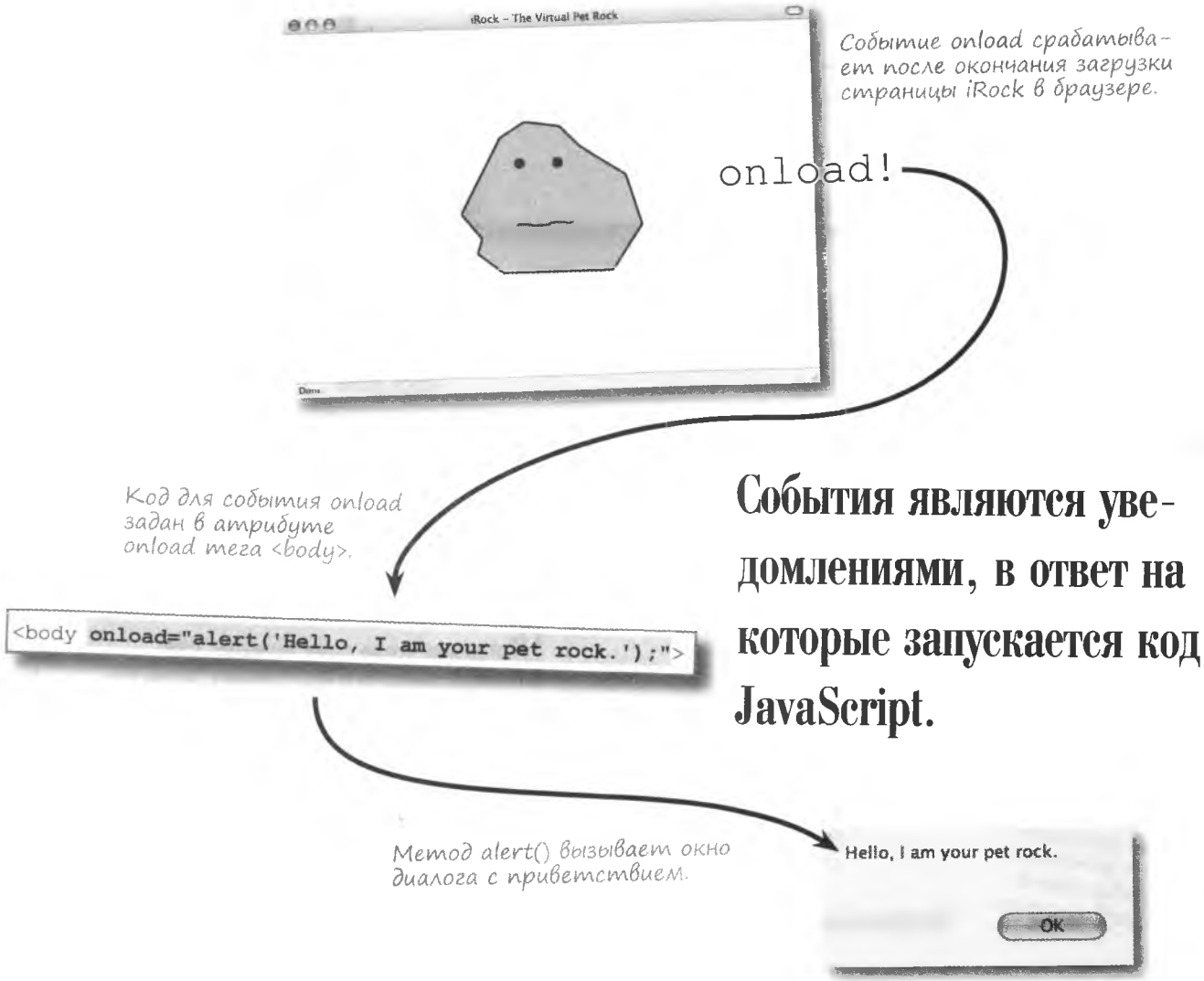
В: Я думал, что не стоит помещать CSS непосредственно на HTML-страницу. Зачем это сделано?

О: Да, обычно лучше поместить CSS в тег <style> в заголовке страницы или на отдельную таблицу стилей. Но ваш начальник не очень хорошо в этом разбирается, а такая запись упрощает пример. Но если вы хотите написать собственную таблицу стилей для файла iRock, обязательно сделайте это.

События

Чтобы заставить объект приветствовать пользователя после загрузки страницы, нужно решить две задачи: определить момент окончания загрузки и придумать способ отображения приветствия.

Решение первой задачи связано с реакцией на событие (загрузку страницы), а во втором случае вам потребуется встроенная функция «alert». *События* (*Events*) в JavaScript называются уведомления о происходящем, например о загрузке страницы (`onload`) или о щелчке на кнопке (`onclick`). В качестве ответа на подобные действия можно написать ваш собственный код JavaScript.



События являются уведомлениями, в ответ на которые запускается код JavaScript.

Оповещение пользователей

JavaScript позволяет вызвать отдельное окно с информацией для пользователей. Для этого вам потребуется написать код, вызывающий метод `alert()` и передающий ему отображаемый текст. *Методами* называются фрагменты кода многократного использования, предназначенные для решения общих задач.

alert( ) ← Если сразу после ключевого слова JavaScript вы видите скобки, скорее всего, это название метода.

Подробно про Методы

Указывает, что нужно щелкнуть на кнопке «Купить дом».

alert — это встроенный метод, отображающий всплывающее окно.

Этот текст появится во всплывающем окне. Не забудьте заключить его в кавычки.

```
alert + ( + Текст + ) + ;
```

Информация, передаваемая методам JavaScript — в нашем случае это отображаемый текст, — заключается в скобки.

Если в конце предложения принято ставить точку, то каждую строку кода JavaScript завершает точка с запятой.

Соединив все вместе, вы получите строку кода JavaScript. Она вызывает метод, отображающий во всплывающем окне указанное вами приветствие:

```
alert('Hello, I am your pet rock.');
```

Отображаемый текст заключается в кавычки.

Методами называются фрагменты кода многократного использования.

Спокойно!

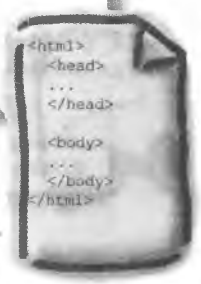
События в этой книге будут фигурировать часто, и постепенно вы выучите синтаксис и поймете, как им пользоваться.

iRock приветствует вас

Чтобы поприветствовать пользователей после загрузки страницы iRock, вам потребуются обработчик события `onload` и метод `alert()`. Добавьте следующую строчку на JavaScript в код `irock.html`:

```
<html>
  <head>
    <title>iRock - The Virtual Pet Rock</title>
  </head>

  <body onload="alert('Hello, I am your pet rock.');">
    <div style="margin-top:100px; text-align:center">
      <img id="rockImg" src="rock.png" alt="iRock" />
    </div>
  </body>
</html>
```

Хотя событие onload event работает для всей страницы, вы задаете его в качестве атрибута тега <body>, так как именно эта часть страницы видна в браузере.

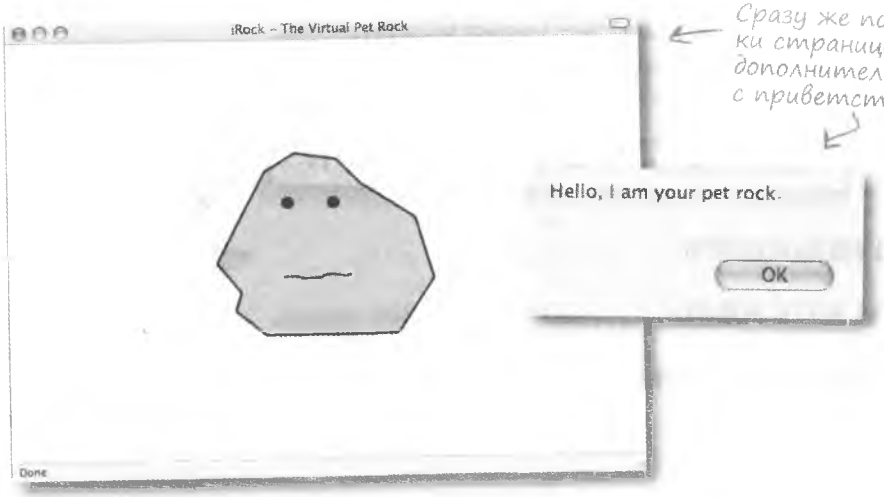


irock.html

Помните, что код на JavaScript встроен непосредственно в веб-страницу. Браузеры умеют обрабатывать его так же, как и HTML с CSS.

Проверка интерактивности

Теперь страница iRock стала более интерактивной благодаря окну с приветствием, появляющемуся в ответ на событие `onload`. Загрузите в браузер страницу irock.html и посмотрите, что произойдет.



Сразу же после загрузки страницы появится дополнительное окно с приветствием.

Часто Задаваемые Вопросы

В: Откуда берутся события?

О: События, конечно, инициируются пользователем, но реализуются браузером. К примеру, при нажатии кнопки браузер должен распаковать информацию (какая именно кнопка была нажата) и передать ее методу, который отвечает за данное событие.

В: А что будет, если не связать с событием никакого кода?

О: Если рядом с падающим деревом никого нет, производит ли оно шум? Вот также и с событиями. Если на событие нет ответа, браузер просто продолжает свою работу.

В: Код JavaScript это то, что заключено между тегами `<script>`?

О: Обычно да. Но этот код можно поместить и непосредственно в обработчик события, как вы видели на примере события `onload`. Именно так обычно и делают, когда требуется запустить всего одну строчку кода JavaScript как для объекта iRock.

В: Существуют ли методы кроме `alert()` ?

О: Да, их множество. Метод `alert()` — это только верхушка айсберга. По мере вашего знакомства с JavaScript вы даже научитесь создавать свои собственные методы.

В: Почему код события `onload` содержит кавычки и апострофы?

О: В HTML и JavaScript перед тем, как начать новое предложение, требуется закрыть предыдущее... если вы не используете другого разделителя. Поэтому, когда код JavaScript появляется внутри атрибута HTML (текст внутри другого текста), решить эту проблему можно совместным употреблением кавычек и апострофов. При этом не важно, что именно вы используете для атрибута или текста JavaScript, главное — быть последовательным. Приведем пример из обычного языка: «При щелчке iRock говорит „Hello there“». Именно по такому принципу должны располагаться кавычки и апострофы.

КЛЮЧЕВЫЕ МОМЕНТЫ

- События отвечают на происходящее на странице кодом JavaScript.
- Событие `onload` происходит в момент окончания загрузки страницы.
- Вы отвечаете на событие `onload`, задав атрибут `onload` в теге `<body>`.
- Методы связывают код JavaScript в модули многократного использования.
- Некоторым методам нужно передавать информацию.
- Метод `alert()` отображает окно с текстовым сообщением.

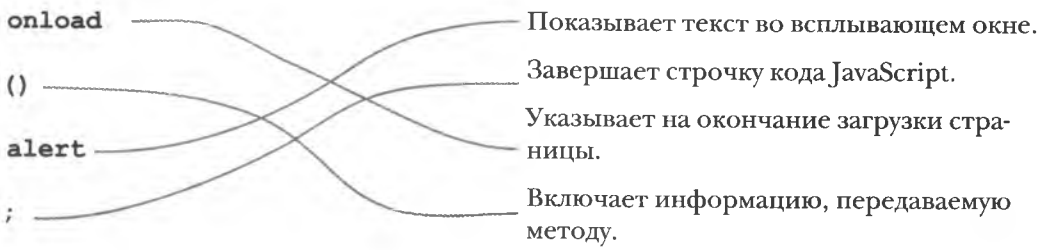
КТО И ЧТО ДЕЛАЕТ?

Укажите, что делает каждый фрагмент кода JavaScript.

`onload`	Показывает текст во всплывающем окне.
`()`	Завершает строчку кода JavaScript.
`alert`	Указывает на окончание загрузки страницы.
`;`	Включает информацию, передаваемую методу.

КТО И ЧТО ДЕЛАЕТ?

Вот, что делает каждый из указанных слева фрагментов кода JavaScript.



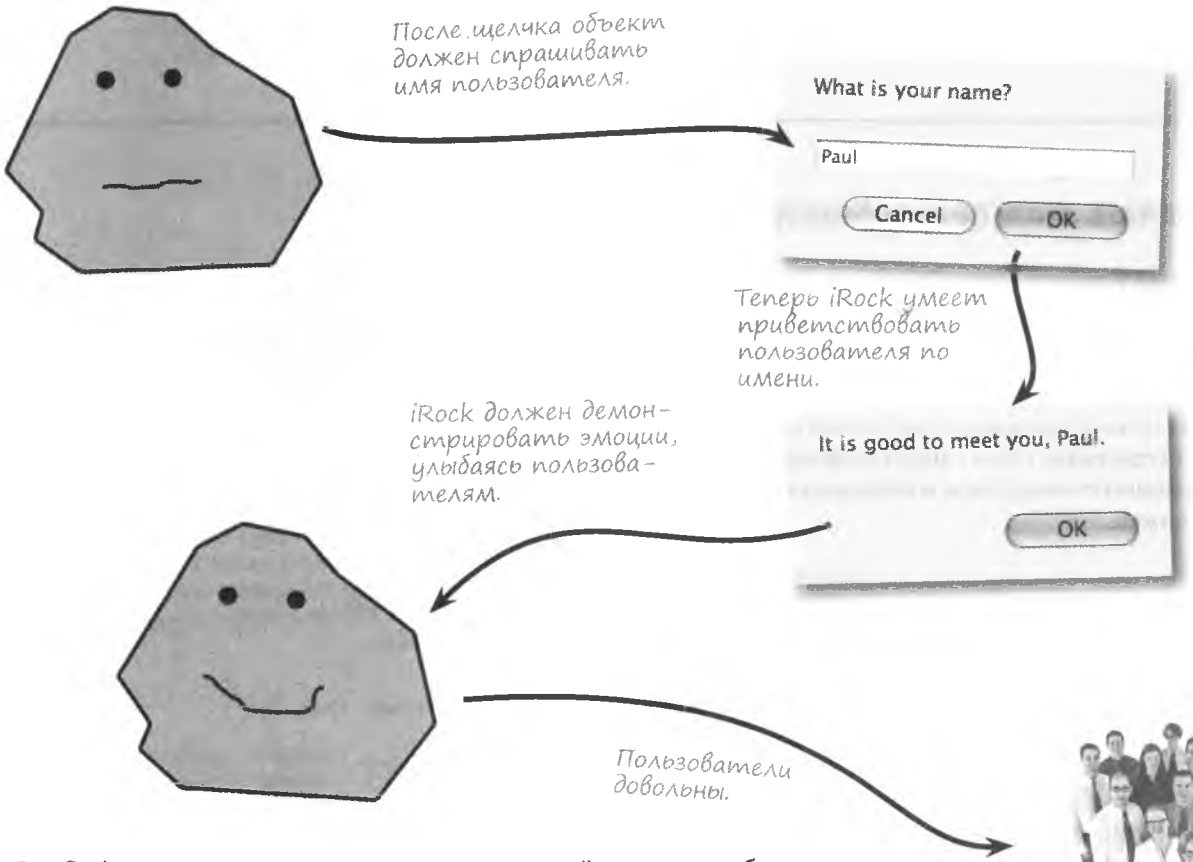
Сделаем объект iRock действительно интерактивным

Вы уже сделали первые шаги в сторону увеличения интерактивности объекта iRock, но до момента, когда виртуального домашнего любимца можно будет представить пользователям, еще далеко... Помните наш список?

1. ~~Создайте для iRock HTML-страницу.~~ ← Сделано!
2. ~~Заставьте объект iRock приветствовать пользователей при загрузке страницы.~~ ← И эту задачу мы тоже решили!
3. **Напишите код, запрашивающий имя пользователя для личного приветствия и заставляющий объект улыбаться.**
4. **Добавьте обработчик события, запускающий написанный на шаге 3 код в ответ на щелчок на объекте.**
5. **Заслужите одобрение и благодарность начальства.**

Взаимодействие должно быть двусторонним

В настоящий момент камешек говорит «Привет», но не позволяет поучаствовать в диалоге. А хочется, чтобы он *отвечал* пользователям. С небольшой помощью JavaScript iRock можно превратить в очаровательного зверька, меняющего выражение лица и приветствующего посетителя страницы по имени...



JavaScript позволяет пользователям взаимодействовать с объектом iRock, превращая нажатия клавиш и щелчки мыши (дополнительные события) в удовольствие от общения хозяина с домашним любимцем. Так благодаря JavaScript рождается дружба!

Возьми в руку карандаш

Напишите, как, по вашему мнению, должно называться событие JavaScript, отвечающее на щелчок мыши.

.....

Возьми в руку карандаш

Решение

Вот как должно называться событие JavaScript, отвечающее на щелчок мыши.

onclick

Событие onclick возникает, когда пользователь щелкает на элементе страницы — при этом каждому элементу можно сопоставить свой собственный уникальный код ответа на это событие.

Как узнать имя пользователя

Готовый код JavaScript

Вот готовый к работе метод JavaScrip. Именно такие куски кода вы будете находить под данным значком. Со временем вы изучите все его детали и сможете писать свой собственный код.

В данном случае перед нами пользовательский метод `touchRock()`, который сначала предлагает пользователю указать свое имя, а затем вызывает отдельное окно с персонифицированным приветствием. Одновременно стандартное изображение объекта iRock заменяется улыбающимся.

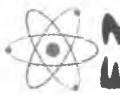
Все методы JavaScript имеют свои имена. Этот метод называется touchRock.

Метод prompt() вызывает всплывающее окно, в которое пользователю предлагается ввести некую информацию.

```
function touchRock() {
  var userName = prompt("Как вас зовут?", "Введите ваше имя.");

  if (userName) {
    alert("Рад вас видеть, " + userName + ".");
    document.getElementById("rockImg").src = "rock_happy.png";
  }
}
```

Теперь, когда мы знаем имя, можно поприветствовать пользователя...

...и заменить обычное изображение камешка улыбающимся.



МОЗГОВОЙ ШТУРМ

В какое место кода страницы irock.html нужно поместить этот метод?

Магниты с кодом JavaScript

Код iRock лишился некоторых фрагментов. Можете ли вы его восстановить?

Подсказка: проверить правильность своих ответов можно методом их ввода в код страницы irock.html.

```
<html>
  <head>
    <title>iRock - The Virtual Pet Rock</title>

    < ..............  type="text/javascript">
      function touchRock() {
        var userName = prompt("Как вас зовут?", "Введите ваше имя.");

        if (userName) {
          alert("Рад вас видеть, " + userName + ".");
          document.getElementById("rockImg").src = "rock_happy.png";
        }
      }
    </script>
  </head>

  <body ..............=" ..............( ..................................................);">
    <div style="margin-top:100px; text-align:center">

      <img id="rockImg" src="rock.png" alt="iRock" style="cursor:pointer"
        ..............=" ........................;" />
    </div>
  </body>
</html>
```

touchRock()

alert

onload

script

onclick

'Hello, I am your pet rock.'

Решение задачи с магнитами

Вот как выглядит целый код.

Методы JavaScript располагаются внутри тега <script>, находящегося в заголовке страницы.

Атрибут type тега <script> задает язык, на котором будет написан сценарий. В нашем случае это JavaScript.

```
<html>
  <head>
    <title>iRock - The Virtual Pet Rock</title>

    <script type="text/javascript">
      function touchRock() {
        var userName = prompt("Как вас зовут?", "Введите ваше имя.");

        if (userName) {
          alert("Рад вас видеть, " + userName + ".");
          document.getElementById("rockImg").src = "rock_happy.png";
        }
      }
    </script>
  </head>

  <body onload="alert('Hello, I am your pet rock.');">
    <div style="margin-top:10px; text-align:center">

      <img id="rockImg" src="rock.png" alt="iRock" style="cursor:pointer"
      onclick="touchRock();" />
    </div>
  </body>
</html>
```

Атрибут события onload в теге <body> привязывает к странице всплывающее окно с приветствием.

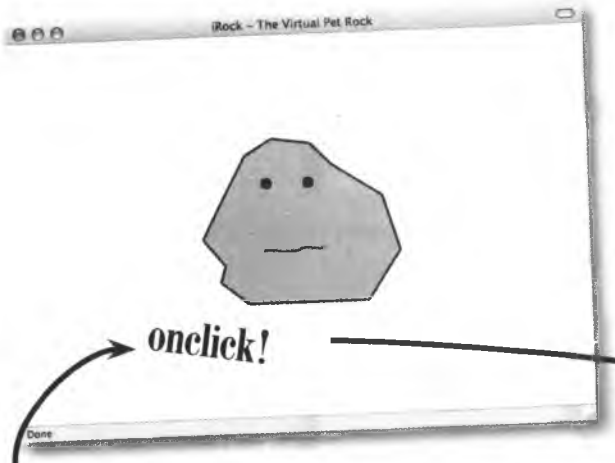
Загрузите изображение улыбающегося камешка.

Атрибут onclick изображения объекта вызывает метод touchRock() при щелчке на объекте.

Наведенный на камешек курсор должен принимать форму руки.

Полученный результат

Небольшой код JavaScript вызвал множество изменений, сделав наш объект iRock намного более интересным. Давайте посмотрим, как теперь выглядит наша страница.

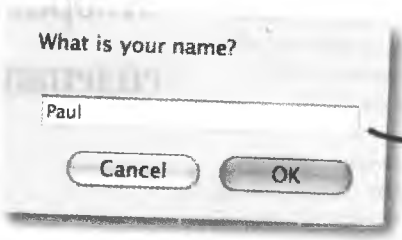


Щелчок на изображении iRock вызывает заданный пользователем метод.

```
<img id="rockImg" src="rock.png" alt="iRock"
  style="cursor:pointer" onclick="touchRock();"
/>
```

```
function touchRock() {
  var userName = prompt("Как вас зовут?",
                        "Введите ваше имя.");
  if (userName) {
    alert("Рад вас видеть, " + userName + ".");
    document.getElementById("rockImg").src = "rock_happy.png";
  }
}
```

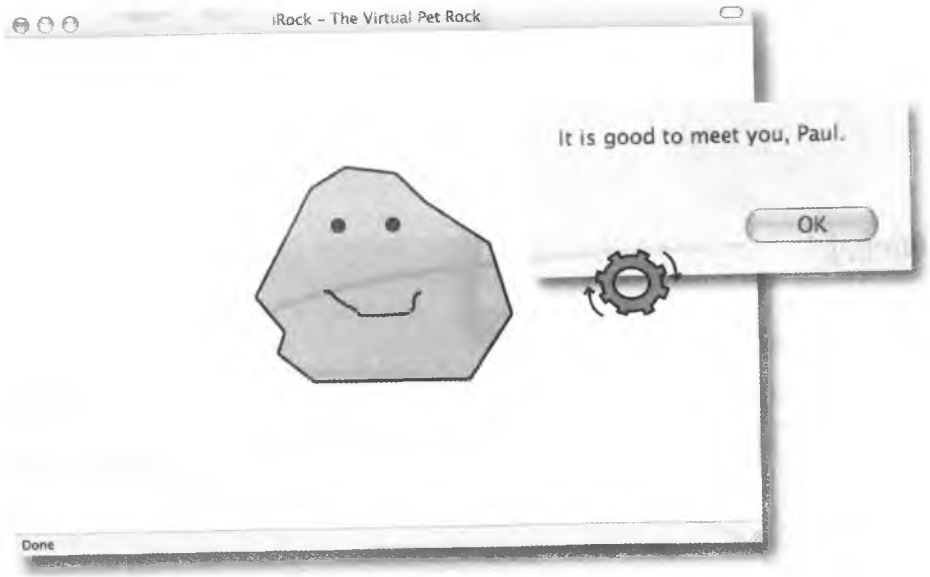
Метод спрашивает, как зовут пользователя, а затем приветствует его по имени.



Камешек начинает улыбаться..

Проверка приложения iRock 1.0

Убедитесь, в том что ваша версия страницы `irock.html` совпадает с показанной на странице 62 и что вы загрузили с сайта Head First Labs (*http://www.headfirstlabs.com/books/hfjs/*) оба необходимых вам изображения. Теперь откройте вашу веб-страницу, и пусть камешек вертится:



1. ~~Создайте для iRock HTML-страницу.~~ — Сделано!
2. ~~Заставьте объект iRock приветствовать пользователей при загрузке страницы.~~ — И эта задача выполнена.
3. ~~Напишите код, запрашивающий имя пользователя для личного приветствия и заставляющий объект улыбаться.~~ — Здесь мы воспользовались методом touchRock().
4. ~~Добавьте обработчик события, запускающий написанный на шаге 3 код в ответ на щелчок на объекте.~~ — Для решения этой задачи нам потребовался обработчик события onclick.
5. Заслужите одобрение и благодарность начальства. — Начальник доволен... Может быть, он выделит нам большой монитор?

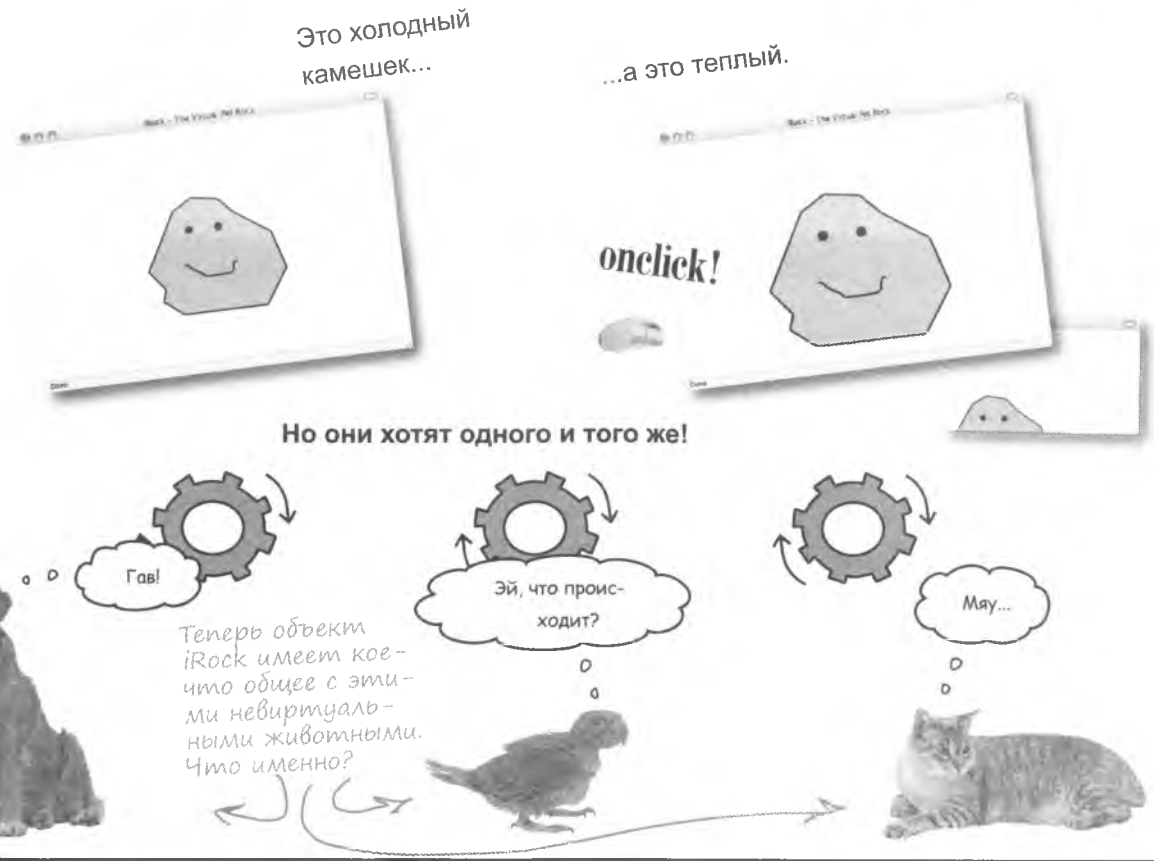
JavaScript позволяет веб-страницам действовать, а не просто отображать содержимое.

Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Что добавляет JavaScript?

Ум хорошо, а два лучше!



Поиск ответов в Интернете, скорее всего, не принесет вам особой пользы.

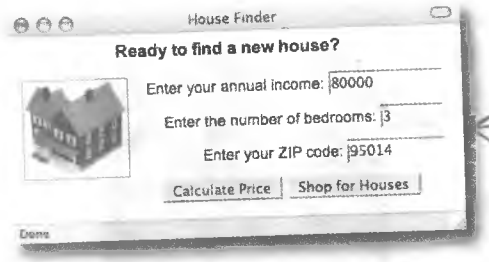
Лучше вместо этого провести время с пользователями.

Все на своем месте

В реальном мире люди часто не придают значения местам для хранения своего имущества. В JavaScript такое поведение невозможно. Ведь там не существует роскоши в виде огромных шкафов и гаражей на три машины. В JavaScript **все имеет свое место**, и ваша задача в этом убедиться. Мы поговорим о **данных** — как их ***представить***, как ***хранить их*** и как их ***найти*** после сохранения. Вы научитесь превращать захламленые комнаты с данными в аккуратные помещения с ящиками, каждый из которых имеет пометку.

Сохранение данных

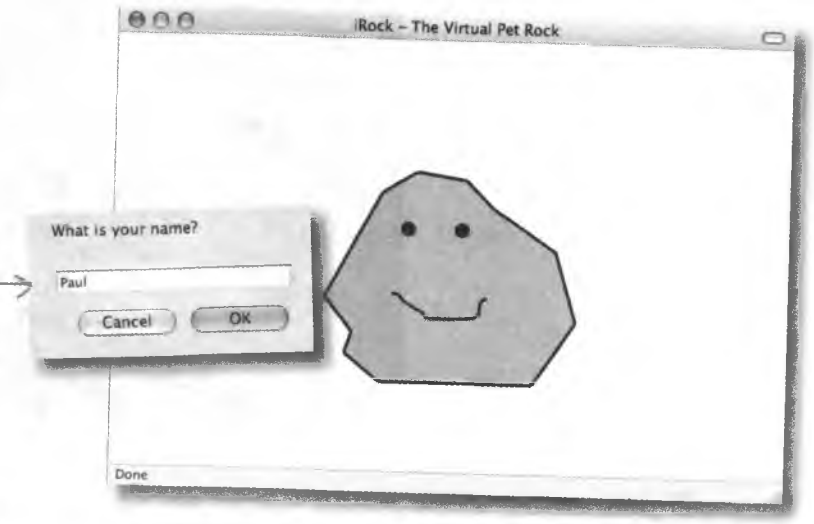
Практически каждый сценарий имеет дело с данными в той или иной ипостаси. Обычно это подразумевает сохранение данных в памяти. За выбор места для хранения отвечает интерпретатор JavaScript. Вам же нужно объяснить ему, **что это за данные** и **как вы собираетесь их использовать**.



Информация, связанная с поиском домов, должна быть сохранена в сценарии, производящем вычисления.

На основе сохраненных данных сценарии выполняют вычисления и **помнят** информацию о пользователе. Без возможности сохранения данных вы никогда не нашли бы новый дом и не познакомились бы с объектом iRock.

Благодаря сохранению введенного на странице iRock имени пользователя становится возможным персонализированное приветствие.

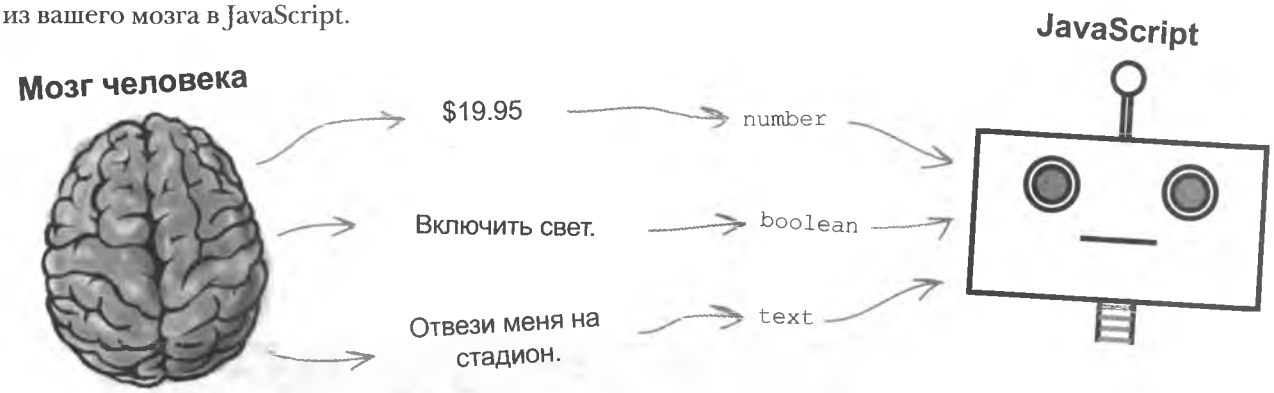


МОЗГОВОЙ ШТУРМ

Подумайте о той информации, с которой приходится иметь дело ежедневно. На что она похожа? Как различается? Каким образом вы бы ее систематизировали?

Типы данных

Данные из реального мира мы категоризируем и систематизируем, даже не задумываясь над этим: имена, числа, звуки и т. п. JavaScript также разделяет данные на **типы**. Именно это является ключом при передаче информации из вашего мозга в JavaScript.



В JavaScript три основных типа данных: text, number и boolean.

Text

Текстовые данные используются для хранения набора символов, например названия вашего любимого завтрака. Можно не ограничиваться словами и предложениями. Текст в JavaScript всегда заключается в кавычки (" ") или в апострофы (' ').

Number

Цифры используются для хранения числовых данных, например массы или количества предметов. В JavaScript они могут быть как целыми (2 кг), так и десятичными (2.5 кг).

Boolean

Логические данные имеют два значения — true и false. Поэтому с их помощью представляются процессы и объекты, имеющие два состояния, например: тостер может быть включен или выключен. Подробно об этом типе данных мы поговорим в главе 4.

Тип данных **определяет** способ его обработки кодом JavaScript. Например, всплывающие окна отображают только текст. Поэтому числа перед отображением следует преобразовать в текстовый формат.

упражнение

Возьми в руку карандаш

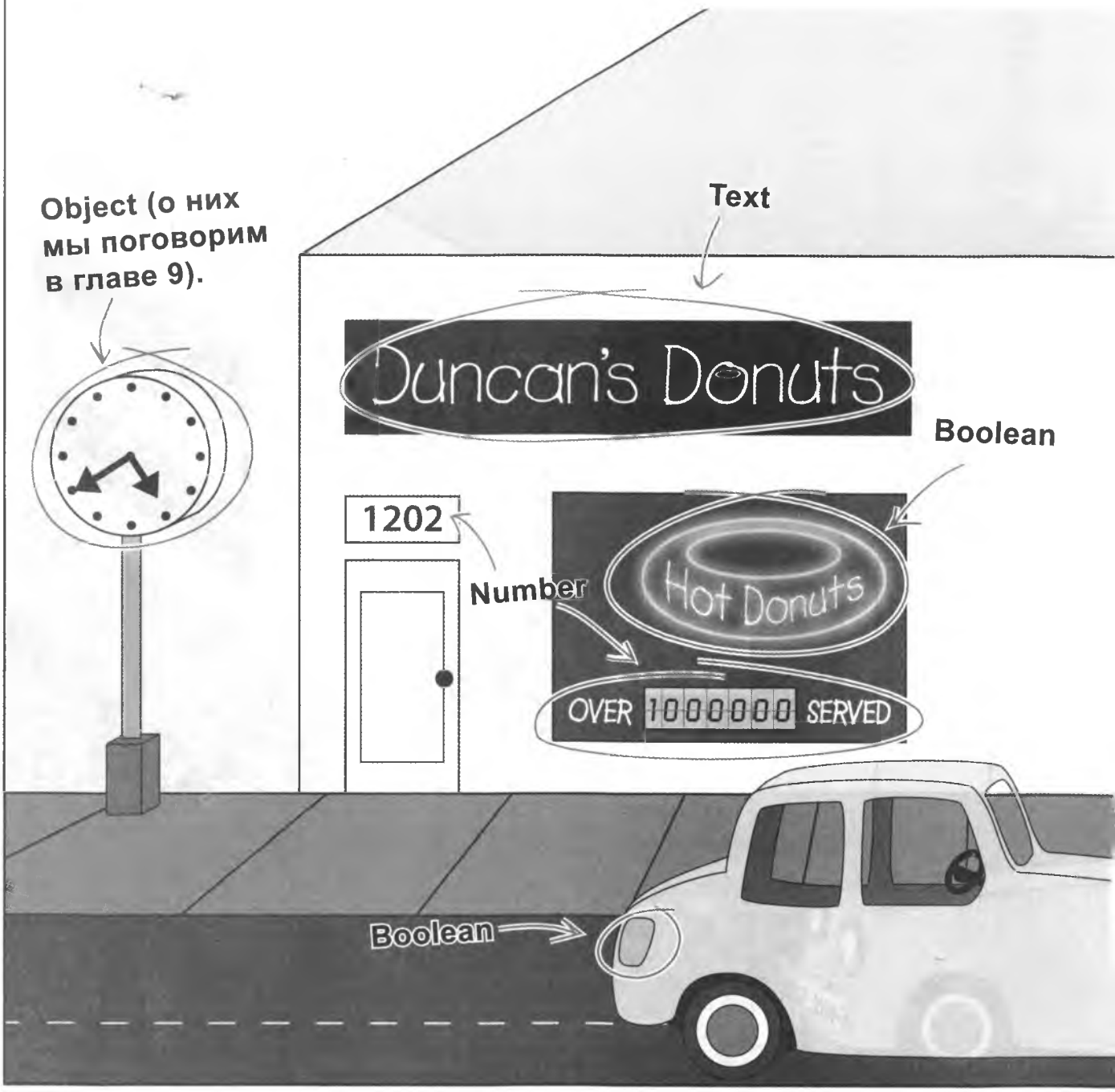
Найдите информацию, которая может быть представлена данными JavaScript, и укажите, к какому типу она должна относиться.

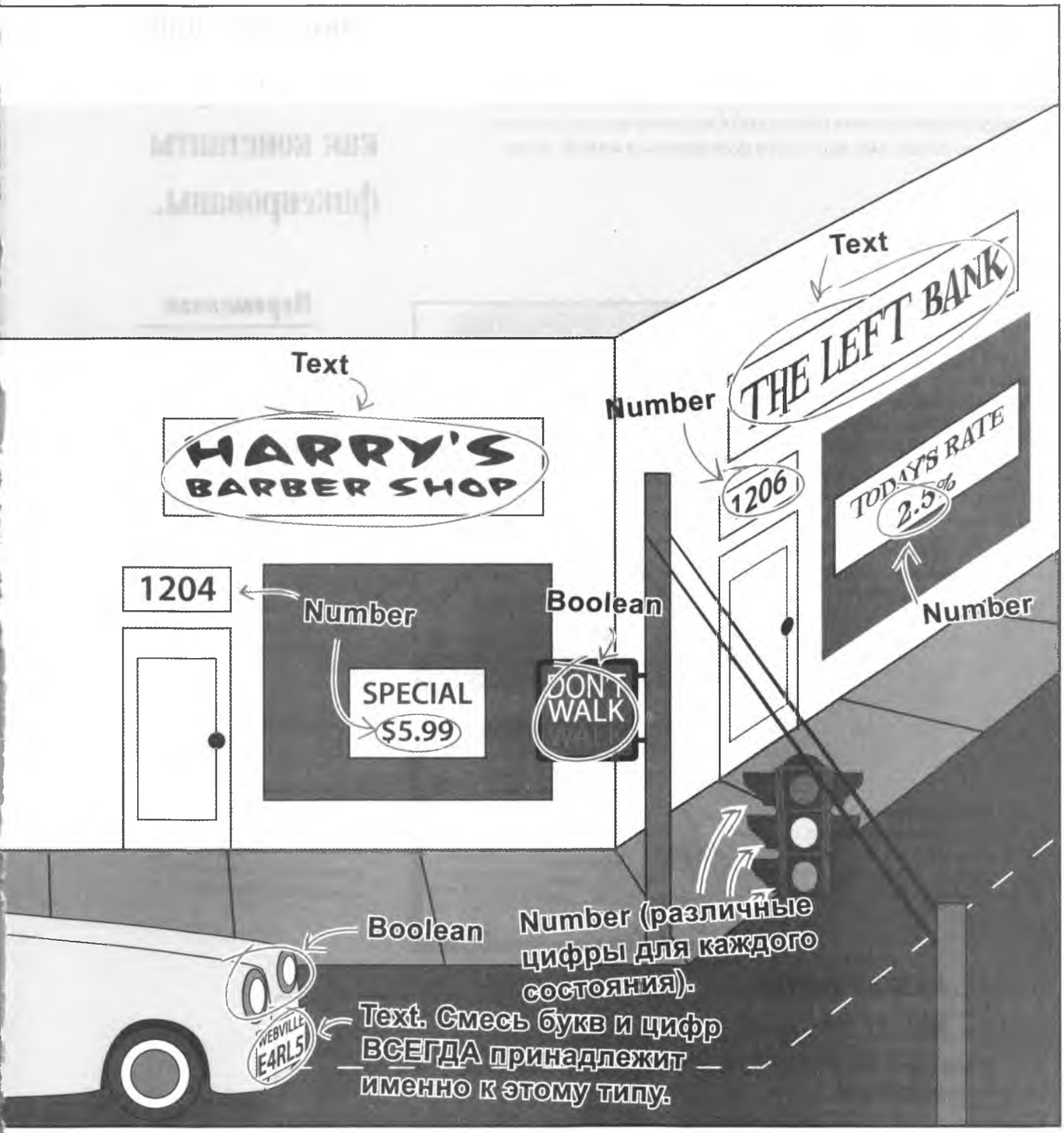

HARRY'S
BARBER SHOP
1204
SPECIAL
$5.99
DON'T
WALK
THE LEFT BANK
1206
TODAYS RATE
2.5%
WEBVILLE
E4RL5

Возьми в руку карандаш

Решение

Вот к каким типам относились бы данные с картинки в сценарии JavaScript.




Text
HARRY'S
BARBER SHOP
Text
THE LEFT BANK
Number
1206
TODAYS RATE
2.5%
Number
1204
Number
SPECIAL
$5.99
Boolean
DON'T
WALK
Boolean
Number (различные цифры для каждого состояния).
Text. Смесь букв и цифр ВСЕГДА принадлежит именно к этому типу.
WEBVILLE
E4RL5

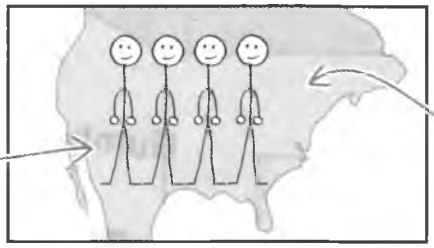
Константы и переменные

Сохраняя данные в JavaScript, нужно помнить не только про их тип, но и про назначение. Будут ли данные меняться в процессе выполнения сценария? От ответа на этот вопрос зависит, константами вы будете пользоваться или же переменными.

Переменные меняют свое значение, в то время как константы фиксированы.

Константа

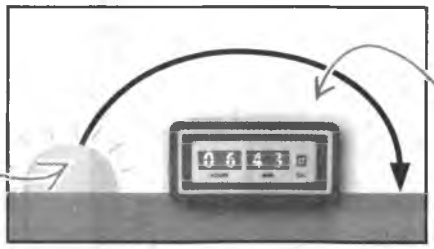
Площадь страны составляет 3.5 миллиона квадратных миль — это константа, если, конечно, не сдвинутся тектонические плиты.



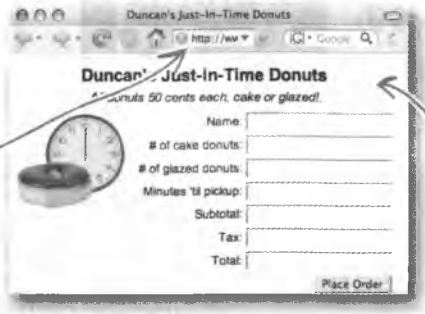
Переменная

Население в 300 миллионов человек — переменная, так как население США возрастает.

В сутках 24 часа. Это константа, известная всем людям.



Восход солнца в 6:43 — это переменная, так как время восхода солнца меняется каждый день.



URL веб-страницы www.duncansdonuts.com — это константа, по крайней мере до тех пор, пока бизнес не придет в упадок.

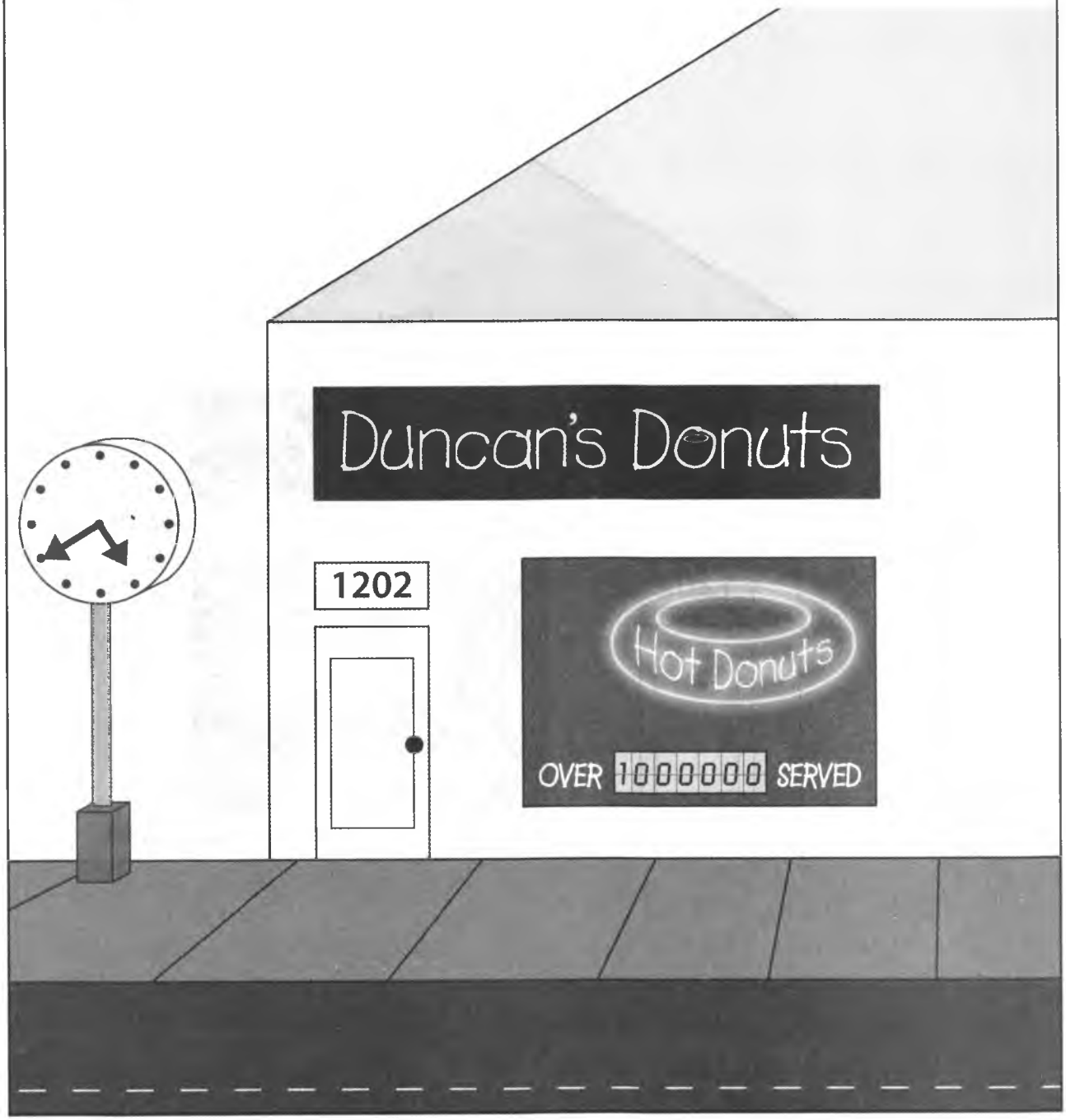
324 посещения — переменная, так как пользователи продолжают заходить на страницу и менять показания счетчика.

МОЗГОВОЙ ШТУРМ

Какие еще типы информации работают как с переменными, так и с константами?

Возьми в руку карандаш

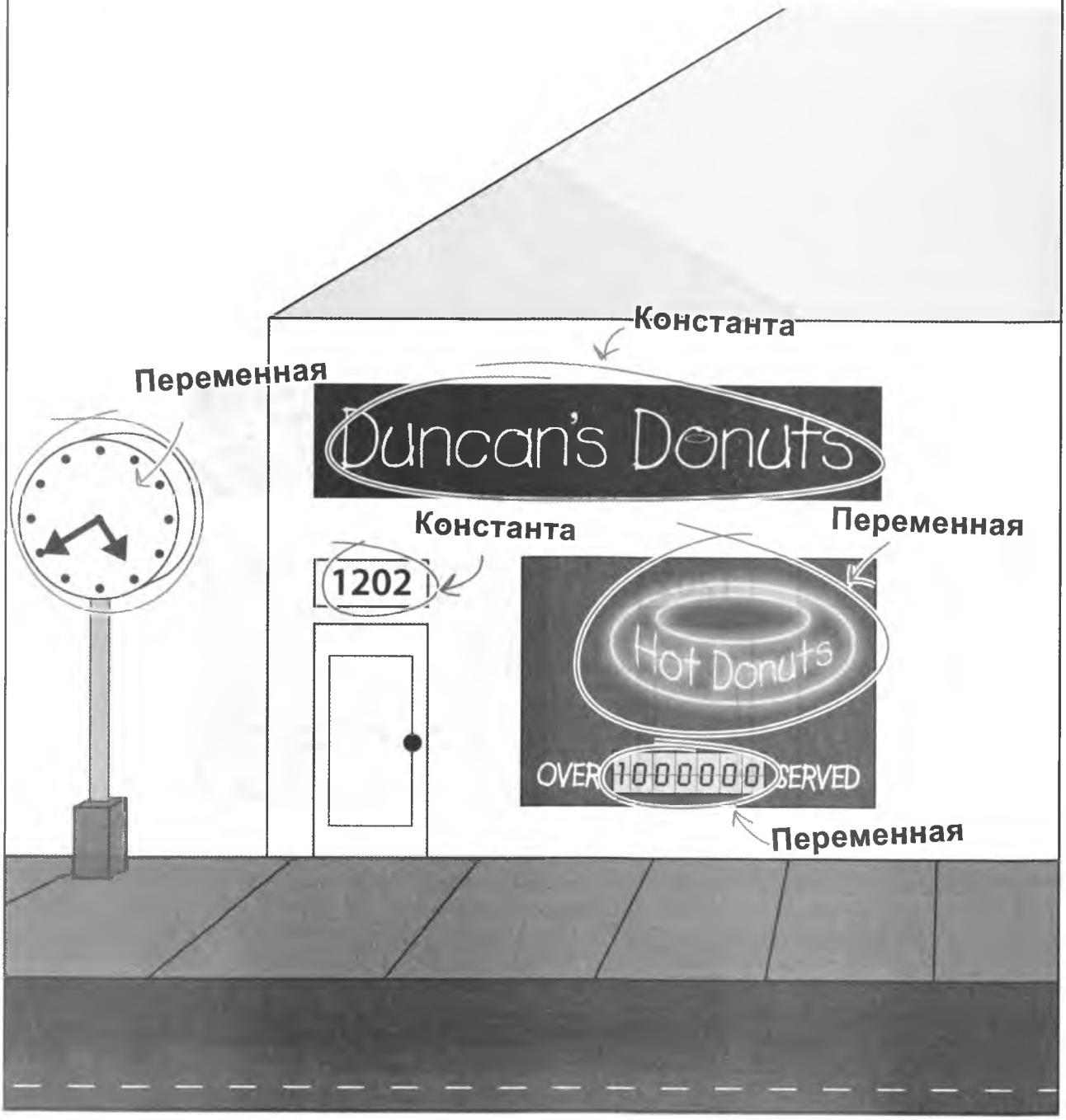
Обведите все данные, связанные с кондитерской Дункана, и укажите, к переменным или к константам они относятся.



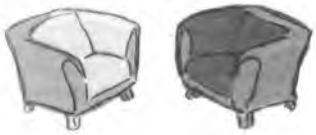
Возьми в руку карандаш

Решение

Вам нужно было определить, какие данные относятся к переменным, а какие к константам.



Беседа у камина



Переменные и константы обсуждают аспекты хранения данных.

Переменная:

Я предлагаю наиболее гибкий способ сохранения данных. Вы можете менять мое значение по своему желанию — именно это я называю свободой.

Разумеется, но твое упорное нежелание меняться ставит в тупик в ситуациях с изменяющимися данными. Например, при запуске ракеты нужно произвести отсчет от 10 до 1. И что ты будешь делать?

Смешно смотреть, как ты радуешься, говоря, что изменения это плохо. Ты просто не понимаешь, что на самом деле это хорошо, особенно когда нужно сохранить информацию, введенную пользователем, выполнить вычисления и прочее в этом роде.

Я думаю, тут можно только соглашаться. Или не соглашаться.

Константа:

А я называю это нерешительностью! Я говорю, что значения нужно выбирать раз и навсегда. И именно моя последовательность делает меня столь ценной... Программисты ценят мою предсказуемость.

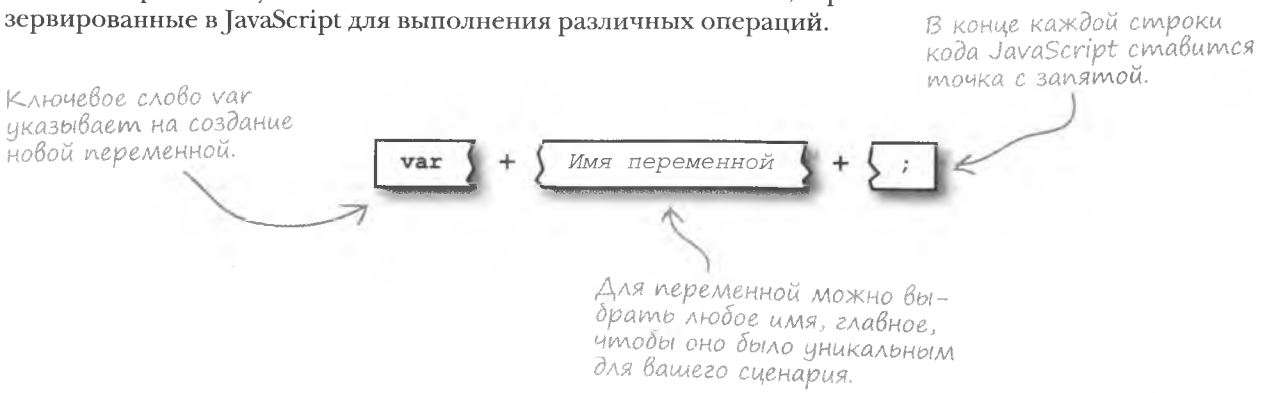
И вот поэтому *ты* считаешь себя единственным вариантом хранения данных для важных приложений? Угадай, почему ракета оказалась на стартовой площадке. Лишь потому, что параметры запуска являются константами. Ты когда-нибудь видела переменный срок окончания проекта?

Чем больше вещи меняются, тем более неизменными они остаются. Почему изменения нужно ставить на первое место? Присвой изначально нужное значение и не трогай его больше. Это же так удобно, когда параметр не может быть случайно изменен.

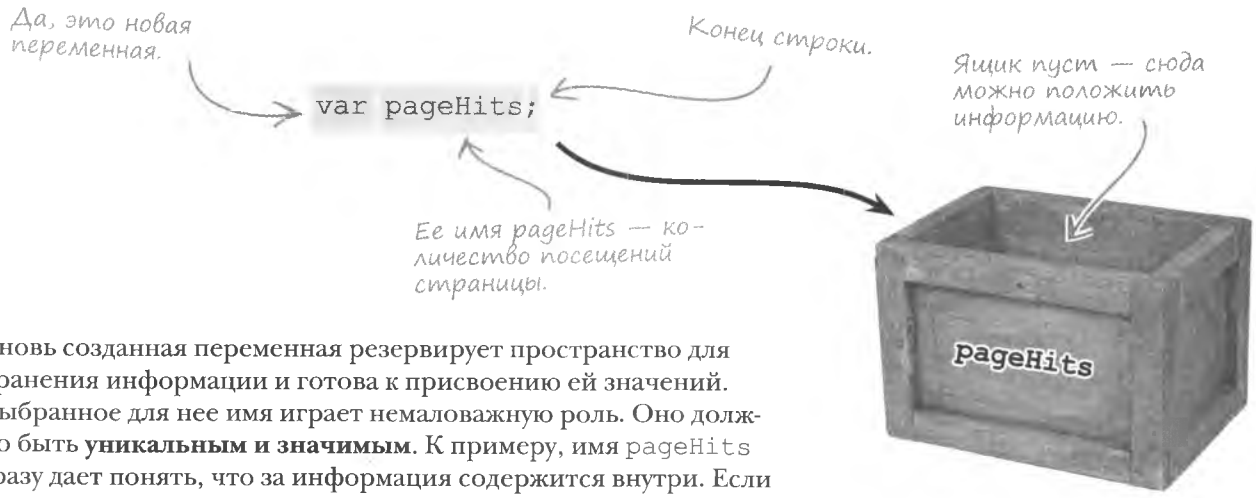
Именно так. И я с тобой полностью несогласна.

Исходное состояние переменных

Переменной называется **место хранения** информации в памяти, имеющее **уникальное имя**. Это как метка на коробке для хранения вещей. Для создания переменной применяется ключевое слово `var`, после которого следует ее имя. **Ключевыми** называются слова, зарезервированные в JavaScript для выполнения различных операций.



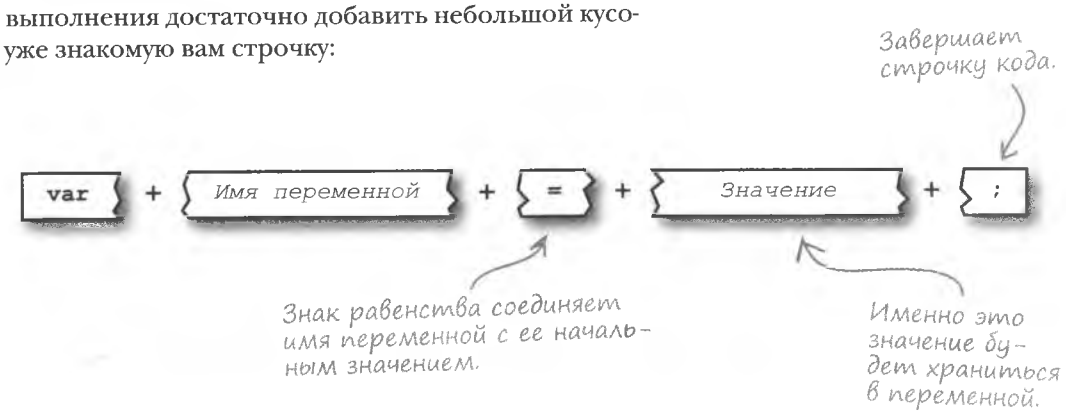
Созданная при помощи ключевого слова `var` переменная изначально пуста — она не имеет никакого значения. Поэтому не имеет смысла пользоваться переменной до операции присвоения. Вы же не включаете MP3-проигрыватель, не вставив в него диск.



Вновь созданная переменная резервирует пространство для хранения информации и готова к присвоению ей значений. Выбранное для нее имя играет немаловажную роль. Оно должно быть **уникальным и значимым**. К примеру, имя `pageHits` сразу дает понять, что за информация содержится внутри. Если бы переменной для подсчета количества посещений страницы было присвоено имя `x` или `gerkin`, вы бы уже не смогли с первого взгляда определить, для чего она предназначена.

Присвоение значений

Не следует создавать переменную, не имеющую значения. Более того, имеет смысл присваивать значение переменной уже в момент создания. Эта процедура называется **инициализацией**. Для ее выполнения достаточно добавить небольшой кусочек кода в уже знакомую вам строчку:

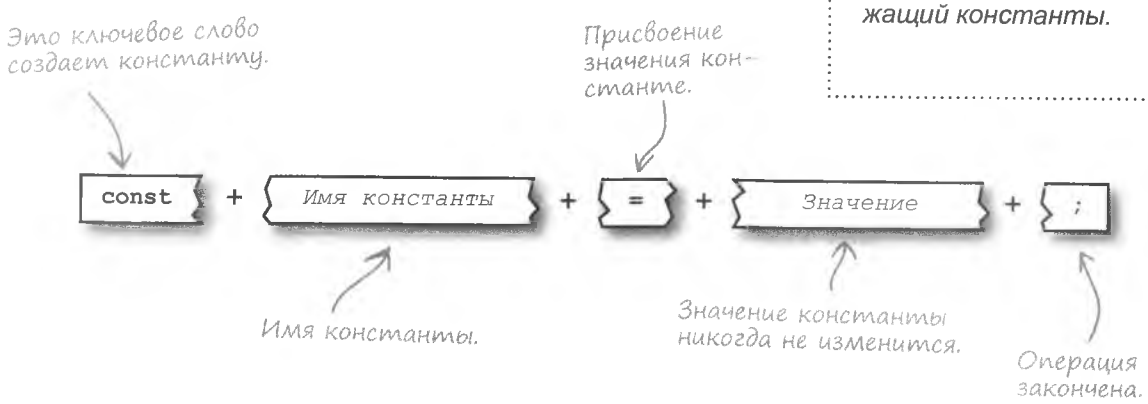


В отличие от своего пустого дубликата, инициализированная переменная немедленно готова к использованию... Поскольку она уже имеет значение. Вы как будто купили MP3-проигрыватель со вставленным внутрь диском — и сразу можете слушать музыку.

Помните о типах данных? Показанная выше строчка кода автоматически назначает переменной тип данных. В данном случае JavaScript создает переменную `population` числового типа, потому что вы присваиваете ей значение 300. После присвоения другого значения тип переменной может измениться. В большинстве случаев JavaScript делает это автоматически, хотя бывают случаи, когда переход от одного типа к другому следует выполнять вручную. Но о них мы поговорим позже.

Константы

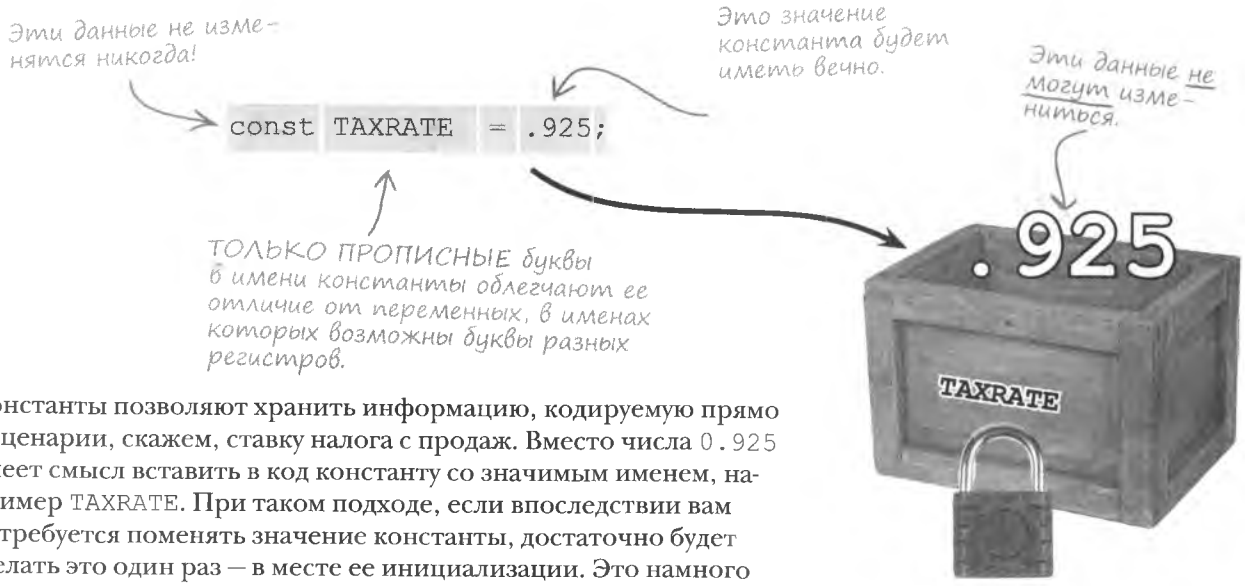
В процессе инициализации переменной присваивается **начальное** значение, но ничто не мешает его **позже** поменять. А вот для хранения данных, которые никогда не будут меняться, вам потребуется константа. Константы создаются аналогично инициализированным переменным, но вместо ключевого слова `var` используется ключевое слово `const`. И «начальное» значение становится **постоянным**!



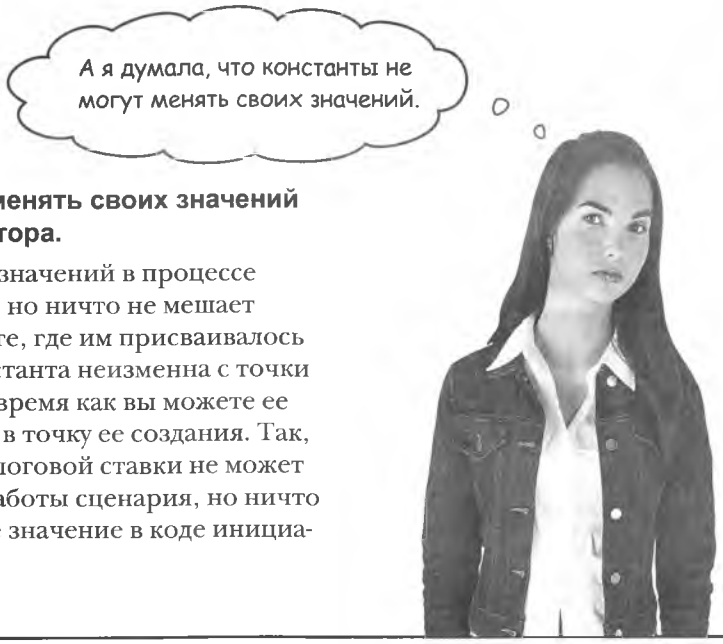
Не все браузеры поддерживают ключевое слово const.

Ключевое слово `const` — новинка для JavaScript, и не все браузеры его понимают. Поэтому тщательно проверяйте те браузеры, для которых пишется код, содержащий константы.

Процесс создания константы отличается только используемым ключевым словом. Остальной синтаксис остается таким же, как при инициализации переменной. Чтобы выделить константы, им часто присваивают имена, написанные большими буквами.



Константы позволяют хранить информацию, кодируемую прямо в сценарии, скажем, ставку налога с продаж. Вместо числа `0.925` имеет смысл вставить в код константу со значимым именем, например `TAXRATE`. При таком подходе, если впоследствии вам потребуется поменять значение константы, достаточно будет сделать это один раз — в месте ее инициализации. Это намного проще, чем искать и заменять числа по всему сценарию.



Константы не могут менять своих значений без текстового редактора.

Константы не меняют значений в процессе выполнения сценария, но ничто не мешает поменять их в том месте, где им присваивалось значение. Так что константа неизменна с точки зрения сценария, в то время как вы можете ее поменять, вернувшись в точку ее создания. Так, ваша константа для налоговой ставки не может меняться в процессе работы сценария, но ничто не мешает поменять ее значение в коде инициализации.

Упражнение

Как вы думаете, переменной или константой должно быть каждое из указанных ниже значений? При желании вы можете написать код для создания и инициализации каждого из них.

Температура воздуха в данный момент. Начальное значение неизвестно.

Коэффициент преобразования возраста собаки в возраст человека (1 год человека = 7 годам собаки).

Обратный отсчет перед запуском ракеты (от 10 до 0).

Цена вкусного пирожка (50 центов).

.....

.....

.....

.....

Упражнение Решение

Вам требовалось решить, чем будет каждое из указанных ниже значений: переменной или константой, а затем написать код для их создания и инициализировать их при необходимости.

Температура воздуха в данный момент. Начальное значение неизвестно.

Коэффициент преобразования возраста собаки в возраст человека (1 год человека = 7 годам собаки).

Обратный отсчет перед запуском ракеты (от 10 до 0).

Цена вкусного пирожка (50 центов).

```
var temp;
```

Температура все время меняется, а ее исходное значение неизвестно, поэтому для нее мы резервируем пустую переменную.

```
const HUMANTODOG = 7;
```

Коэффициент преобразования не меняется, поэтому для него превосходно подойдет константа.

```
var countdown = 10;
```

Обратный отсчет меняется от 10 до 1, поэтому нам нужна переменная с начальным значением 10.

```
var donutPrice = 0.50;  или  const DONUTPRICE = 0.50;
```

Меняющуюся цену на пирожки нужно представить переменной с указанным в задаче начальным значением.

Если же цена фиксированная, лучше всего выбрать для ее представления константу.

Часто Задаваемые Вопросы

В: Каким образом JavaScript определяет тип данных, если я их не указываю?

О: В отличие от других языков программирования, JavaScript не позволяет в явном виде задавать тип констант и переменных. Тип **выражается неявно** в момент присвоения значения. Это дает переменным JavaScript бóльшую гибкость. К примеру, присвоив число 17 переменной `x`, вы приведете ее к числовому типу. Но если затем присвоить ей же текст "seventeen", тип изменится на строковый.

В: Если JavaScript автоматически заботится о типах данных, зачем мне вообще о них задумываться?

О: Во многих ситуациях нельзя целиком положиться на автоматическую обработку данных средствами JavaScript. К примеру, у вас есть число, сохраненное в виде текста, которое нужно использовать в вычислениях. Для этого его предварительно нужно преобразовать из строкового типа в числовой. Обратная ситуация возникает при необходимости отобразить число во всплывающем окне. JavaScript умеет автоматически преобразовывать числа в текст и обратно, но не всегда делает это так, как вам нужно.

В: Можно ли оставить переменную неинициализированной, если я заранее не знаю ее значения?

О: Конечно. Инициализация призвана предотвратить проблемы, которые могут возникнуть при попытке доступа к переменной, не имеющей значения. Но бывают и случаи, когда в момент создания переменной ее значение еще неизвестно. Тогда следить за тем, чтобы обращение не происходило к пустым переменным, уже вам. Кстати, переменным можно присваивать и «пустые» значения, например `""` для текста, `0` для числа, или `false` логического типа. Это уменьшает риск обращения к неинициализированным данным.

В: Как определить, когда мне нужна переменная, а когда константа?

О: Часто пользователи начинают работать исключительно с переменными, а позже обнаруживают, что некоторые из них можно превратить в константы. Скорее всего, это будут повторяющиеся строки текста или число, которое встречается в нескольких местах кода, к примеру, повторяющееся приветствие или коэффициент преобразования. Вместо того чтобы писать этот текст или число снова и снова, имеет смысл создать для них константы. Это облегчит вам работу в будущем, если вдруг потребуется поменять значение этих параметров.

В: Что происходит с данными сценария при перезагрузке страницы?

О: Все данные принимают свои начальные значения, как будто сценарий перед этим и не запускался. Другими словами, после перезагрузки страницы сценарий запускается сначала.

Тип данных определяется после того, как переменной или константе было присвоено значение.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Данные в сценариях обычно принадлежат к одному из трех типов: **text**, **number** или **boolean**.
- **Переменной** называется кусок данных, который **меняется** в процессе работы сценария.
- **Константа** — это **неизменяемый** кусок информации.
- Ключевое слова `var` **создает переменные**, в то время как `const` **создает константы**.
- **Тип данных** JavaScript определяется в момент **присвоения значения**. У переменных тип данных может меняться.

Что в имени тебе моем?

Переменные, константы и другие синтаксические конструкции JavaScript определяются по своим уникальным именам или, как их еще называют, идентификаторам. Индентификаторы JavaScript напоминают имена людей из реального мира, хотя и с некоторыми ограничениями (люди могут носить одинаковые имена, а вот переменные JavaScript – нет). Кроме того, имеет смысл придерживаться следующих правил именования:

- Идентификатор должен быть длиной хотя бы в один символ.
- Первым символом идентификатора должна быть буква, знак подчеркивания (_) или знак доллара ($).
- В символах после первого допускаются буквы, цифры, знаки (_) и ($).
- Пробелы и специальные символы, отличные от _ и $, в идентификаторах недопустимы.

Когда вы создаете идентификатор JavaScript для переменной или константы, давайте им «говорящие» имена. То есть вам недостаточно просто выполнить перечисленные выше требования. Имена должны быть такими, чтобы по их виду было понятно, для чего предназначена та или иная переменная или константа.

Разумеется, бывают и случаи, когда можно обойтись простым x – далеко не каждый фрагмент информации в сценарии можно легко описать.

Идентификаторам следует присваивать значимые имена.

Корректные и некорректные имена

Некорректно: имя не может начинаться с цифры.

firstName

Корректно: имя состоит только из букв.

top100

Корректно: цифры расположены на позициях, отличных от первой.

ka_chow

Корректно: буквы с нижним подчеркиванием составляют допустимую комбинацию.

Некорректно: имя не может начинаться со специального символа, отличного от «_» или «$».

_topSecret

Корректно: начинать имя с нижнего подчеркивания можно — некоторые пользователи даже специально применяют такое именование, чтобы подчеркнуть особое значение переменной.

$total

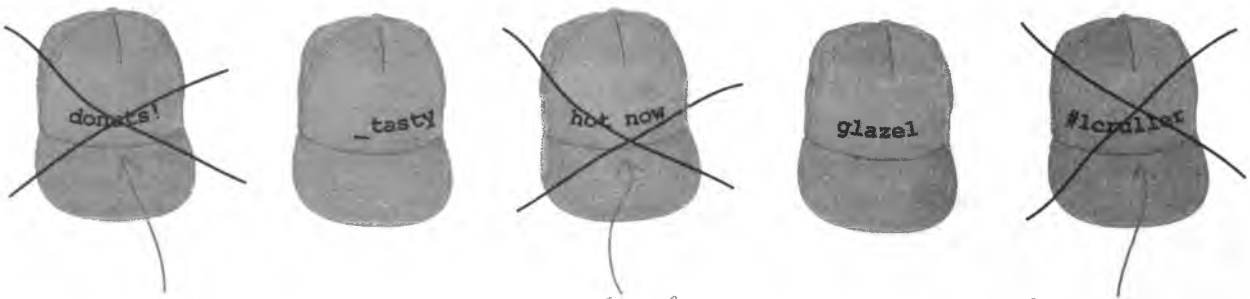
Корректно: хотя такие имена и выглядят несколько странно, но знак доллара на первой позиции вполне допустим.

Упражнение

Вот примеры бейсболок для агентов, рекламирующих кондитерскую Дункана. К сожалению, при их разработке не учитывались стандарты именования идентификаторов в JavaScript. Зачеркните неподходящие с точки зрения JavaScript варианты.

Упражнение Решение

Вам нужно было вычеркнуть бейсболки с именами, некорректными с точки зрения JavaScript.



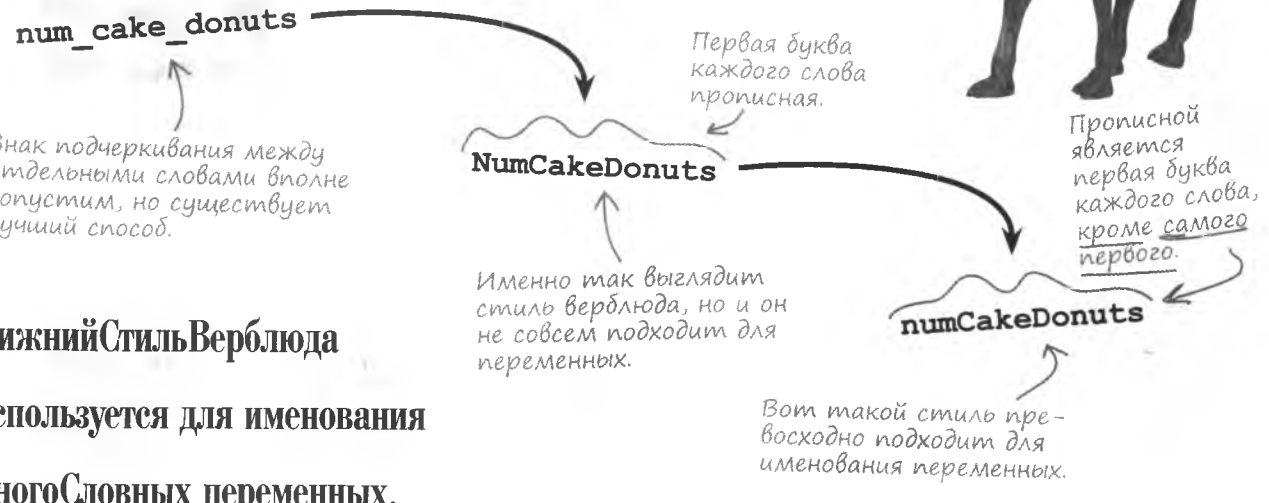
В именах идентификаторов недопустимо использовать восклицательный знак.

И пробелы в них также запрещены..

Символ # вызовет ярость у шерифа Правосудова.

СтильВерблюда

Законов, обязующих использовать определенные стандарты именования в JavaScript, не существует, но есть **неофициальные** правила, которых придерживается большинство. Одно из таких правил — использовать **СтильВерблюда** в составных именах идентификаторов. Такое название появилось из-за того, что заглавные буквы внутри слова напоминают горбы верблюда. В именах переменных первое слово обычно пишется буквами нижнего регистра, а все прочие — смешанным регистром.



НижнийСтильВерблюда используется для именования многоСловных переменных.

Магниты JavaScript

От магнитов с описаниями отвалились имена переменных и констант. Поместите на каждый магнит нужное имя, особо проследив за их корректностью. В качестве дополнительного задания в каждом случае укажите тип данных.

Количество проданных сегодня чашек кофе (number of cups).

Имя работника месяца (the employee of the month).

Количество муки (flour) для приготовления одной порции (batch) пирожков.

Налоговый номер (tax number) для фиксации налога с продаж.

Состояние (status) аварийной сигнализации (alarm).

Рекордсмен (record holder) по поеданию эклеров (eclairs).

`employee*of*the*Month`

`cups-o-coffee`

`Employee of the Month`

`alarmStatus`

`Tax#`

`FLOURPERBATCH`

`alarm_status`

`TAXNUM`

`eclairRECORDHOLDER`

`numCups`

`eclairRecord`

`eclairWinner!`

`employeeOfMonth`

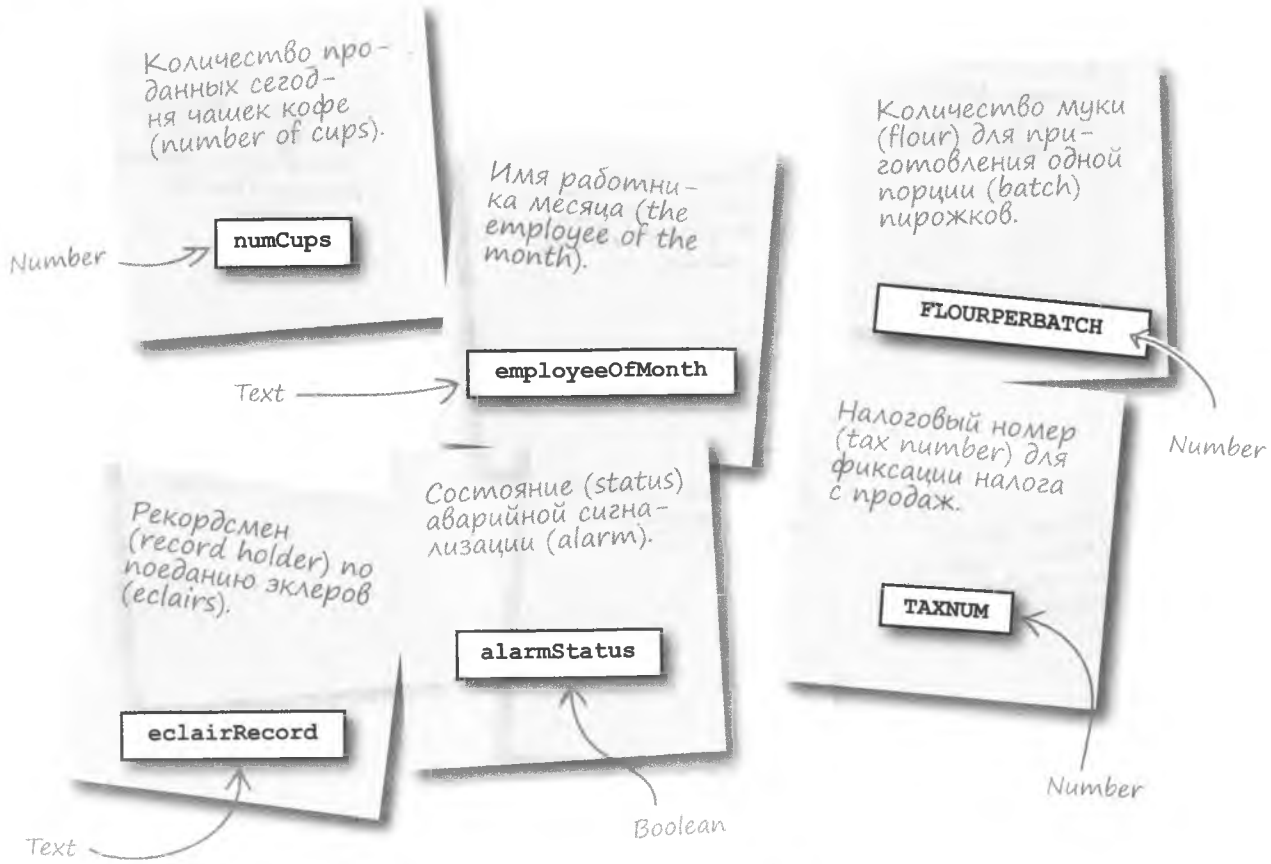
`ALARM-STATUS`

`flour quantity`

`#OfCups`

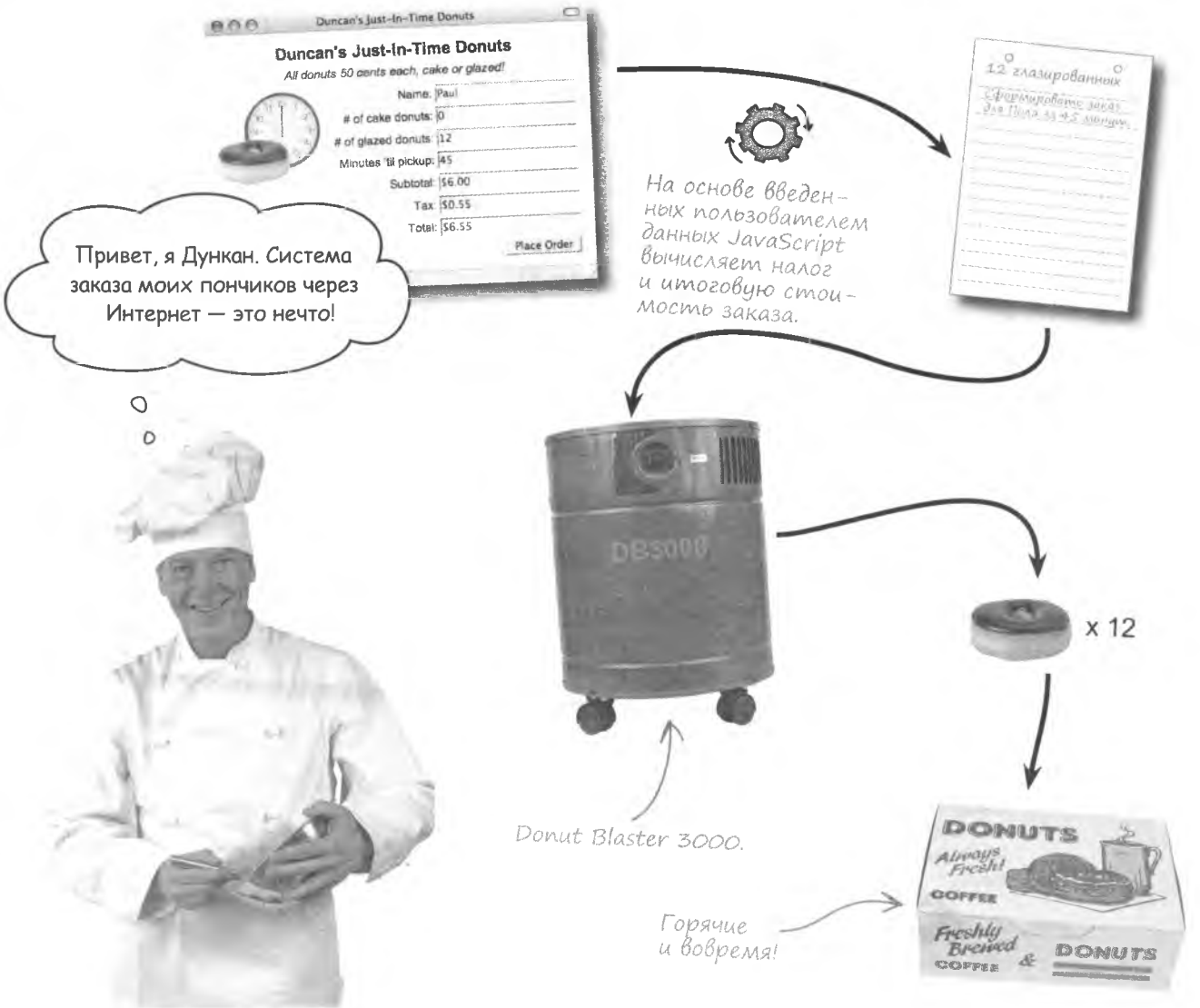
Решение задачи с магнитами

Вам нужно было поместить на каждый магнит имя подходящей переменной или константы. В качестве дополнительного задания предлагалось указать тип данных.



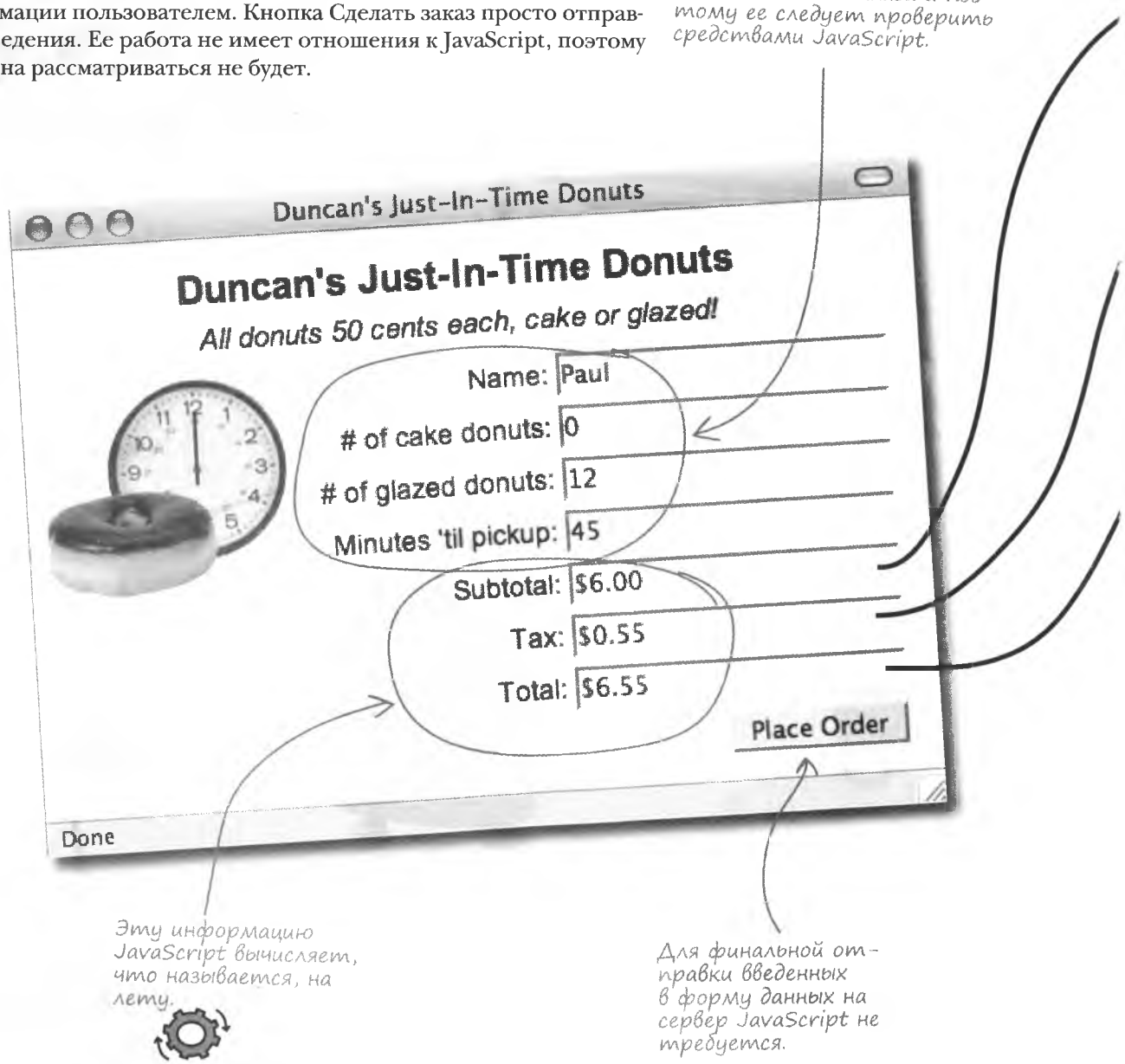
Следующий этап

Вы уже читали о заведении Дункана, но пока ничего не знаете о планах его хозяина. Дункан хочет перевести свой бизнес на новый уровень... Продавать пончики через Интернет! Только представьте, пользователь вводит в форму количество товара и время доставки и точно в указанное время получает горячие пончики. Вам нужно убедиться, что пользователь ввел все требуемые данные, и вычислить налог и итоговую сумму заказа.



Планируем Веб-страницу

Обработка заказов включает в себя как проверку введенных в форму данных, так и вычисление на их основе суммы заказа. Промежуточная и полная суммы должны отображаться сразу же после ввода информации пользователем. Кнопка Сделать заказ просто отправляет сведения. Ее работа не имеет отношения к JavaScript, поэтому здесь она рассматриваться не будет.



Промежуточная сумма вычисляется умножением требуемого количества пончиков на стоимость одного пончика:

(кол-во пончиков + кол-о глазированных) x цена пончика

Налог вычисляется умножением промежуточной суммы на ставку:

сумма x налоговая ставка

Общая стоимость заказа вычисляется сложением промежуточной суммы и налога:

сумма + налог

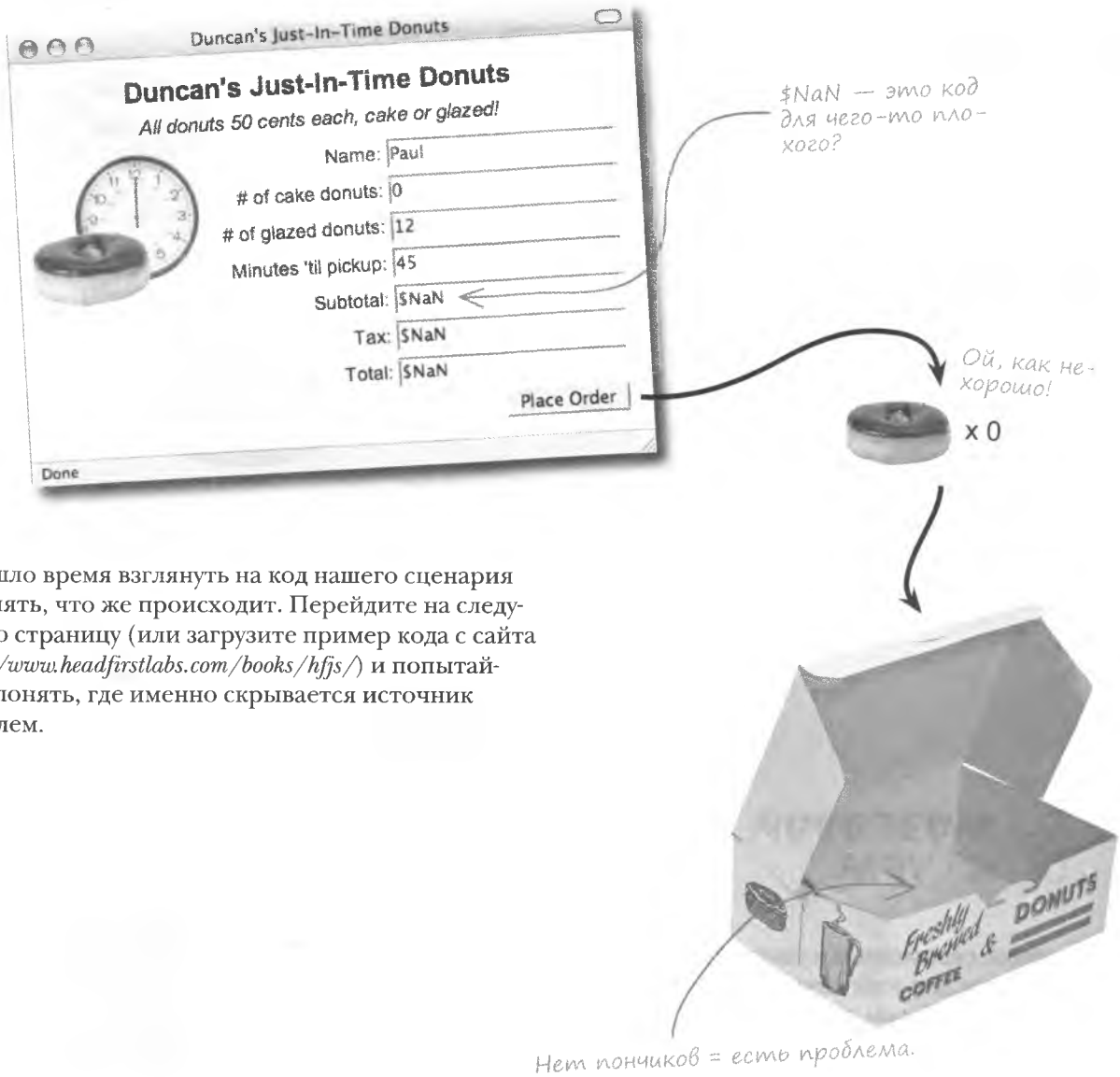
Кажется, у Дункана достаточно данных, чтобы отслеживать их при помощи формы. Ему нужно быть не только в курсе введенной пользователем информации, но и вычислить при помощи кода JavaScript ряд параметров.

МОЗГОВОЙ ШТУРМ

Какие переменные и константы понадобятся для этих вычислений? Какие имена вы бы им присвоили?

Начнем с вычислений

Дункан попытался самостоятельно написать код JavaScript, выполняющий подсчеты, но столкнулся с проблемой. После ввода пользователем количества требуемых пончиков в полях с результатами вычислений стало появляться не имеющее никакого смысла значение `$NaN`. Более того, заказ не заполнялся. И клиенты были совершенно не в восторге от технологического «усовершенствования» Дункана.



Пришло время взглянуть на код нашего сценария и понять, что же происходит. Перейдите на следующую страницу (или загрузите пример кода с сайта *http://www.headfirstlabs.com/books/hfjs/*) и попытайтесь понять, где именно скрывается источник проблем.

Этот код обновляет заказ, мгновенно вычисляя промежуточную и полную суммы.

Так как введенные пользователем данные в порядке, для поиска проблемы обратим внимание на константы.

Этот код отправляет заказ на сервер и подтверждает его принятие.

Заказ обновляется при изменении количества пончиков.

Заказ принимается после щелчка на кнопке Сделать заказ.

```
<html>
  <head>
    <title>Пончики Дункана к указанному времени</title>
    <link rel="stylesheet" type="text/css" href="donuts.css" />
    <script type="text/javascript">
      function updateOrder() {
        const TAXRATE;
        const DONUTPRICE;
        var numCakeDonuts = document.getElementById("cakedonuts").value;
        var numGlazedDonuts = document.getElementById("glazeddonuts").value;
        var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
        var tax = subTotal * TAXRATE;
        var total = subTotal + tax;
        document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
        document.getElementById("tax").value = "$" + tax.toFixed(2);
        document.getElementById("total").value = "$" + total.toFixed(2);
      }
      function placeOrder() {
        // Передать заказ на сервер...
        form.submit();
      }
    </script>
  </head>
  <body>
    <div id="frame">
      ...
      <form name="orderform" action="donuts.php" method="POST">
        ...
        <div class="field">
        Число пончиков: <input type="text" id="cakedonuts" name="cakedonuts"
          value="" onchange="updateOrder();" />
        </div>
        <div class="field">
        Глазированных: <input type="text" id="glazeddonuts"
          name="glazeddonuts" value="" onchange="updateOrder();" />
        </div>
        ...
        <div class="field">
        <input type="button" value="Сделать заказ"
        onclick="placeOrder(this.form);" />
        </div>
      </form>
    </div>
  </body>
</html>
```

Возьми в руку карандаш

Запишите, что, по вашему мнению, не так с кодом данного сценария.

.....

.....

Возьми в руку карандаш

Решение

Итак, вот в чем была проблема с кодом сценария для заказа пончиков через Интернет.

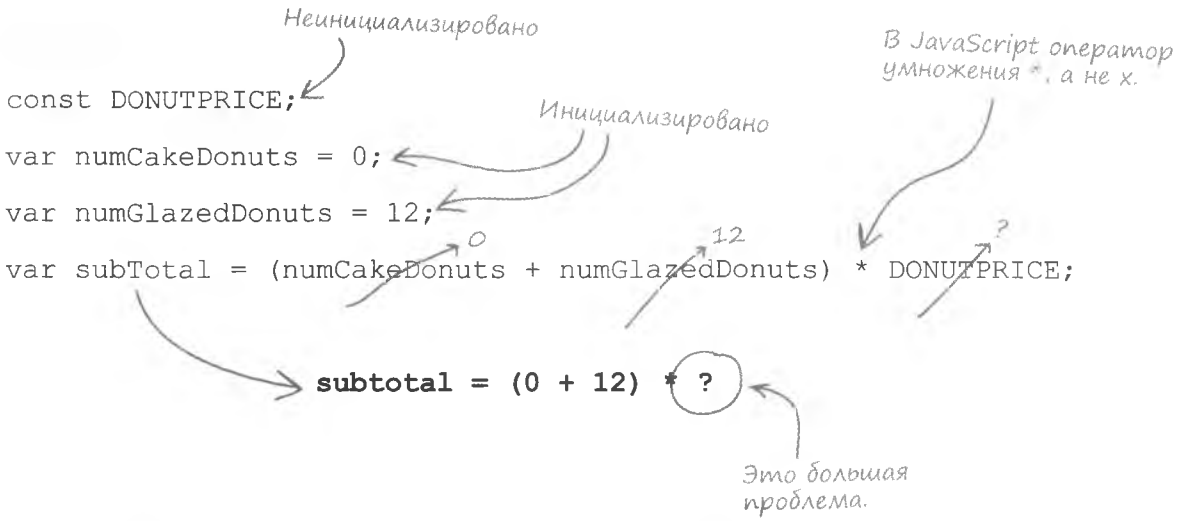
Константы TAXRATE и DONUTPRICE не были инициализированы, и вычисления, в которые они входят, стали невозможны.

Никогда не оставляйте константы без значения.

Если не инициализировать константу в момент ее создания, она не будет иметь значения — и, что хуже всего, ей будет невозможно его присвоить. Такие константы являются ошибками кода, хотя браузеры об этом не сообщают.

Инициализируйте данные... или...

Неинициализированые данные считаются неопределенными. Другими словами, они не имеют никакого значения. Это не значит, что они совершенно бесполезны. Они просто не содержат информации... пока. Проблемы начинаются, когда вы собираетесь использовать подобные переменные или константы, например, для вычислений.

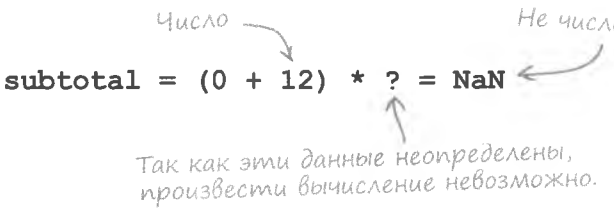


Константа `DONUTPRICE` неинициализирована. В JavaScript такие данные обозначают специальным состоянием: `undefined`. Это все равно что оставить на автоответчике сообщение «сообщений нет», если вам нечего сказать — вы оставляете сообщение, обозначающее его отсутствие. Так же и с состоянием `undefined` — оно обозначает отсутствие данных.

Данные не определены, если им не было присвоено значение.

NaN — это НЕ число

Подобно значению `undefined`, которое представляет особое состояние данных, существует еще одно значение переменных JavaScrip: `NaN`. Эта аббревиатура расшифровывается как Not a Number – не число. Именно такое значение присваивается переменной `subTotal` из-за того, что нам не хватает информации для расчетов. Другими словами, вы воспринимали отстутствующее значение как число... а получили `NaN`.



NaN — это значение, не являющееся числом.

Так, для решения возникшей проблемы инициализируйте константу `DONUTPRICE` в момент ее создания:

```
const DONUTPRICE = 0.50;
```

Часто Задаваемые Вопросы

В: Что означает «идентификаторы должны быть уникальны в пределах сценария»?

О: Идентификатор выступает как уникальное имя для фрагмента информации в сценарии. В реальном мире люди часто имеют одинаковые имена... но при этом они вполне в состоянии разобраться «кто есть кто». JavaScript не умеет решать подобные неопределенности, поэтому различные фрагменты информации различают, присваивая им *разные* имена. И вы должны следить за уникальностью всех идентификаторов в ваших сценариях.

В: Идентификаторы должны быть уникальными вообще или только в пределах сценария?

О: Уникальность идентификатора важна именно **в пределах** сценария, а иногда и в пределах части сценария. Но следует помнить, что для больших приложений иногда требуются очень большие сценарии, распределенные по файлам. В этом случае имеет смысл делать идентификаторы полностью уникальными. Но для этого достаточно давать им значимые имена в контексте выполняемой задачи.

В: Я так и не понял, зачем нужен стиль верблюда?

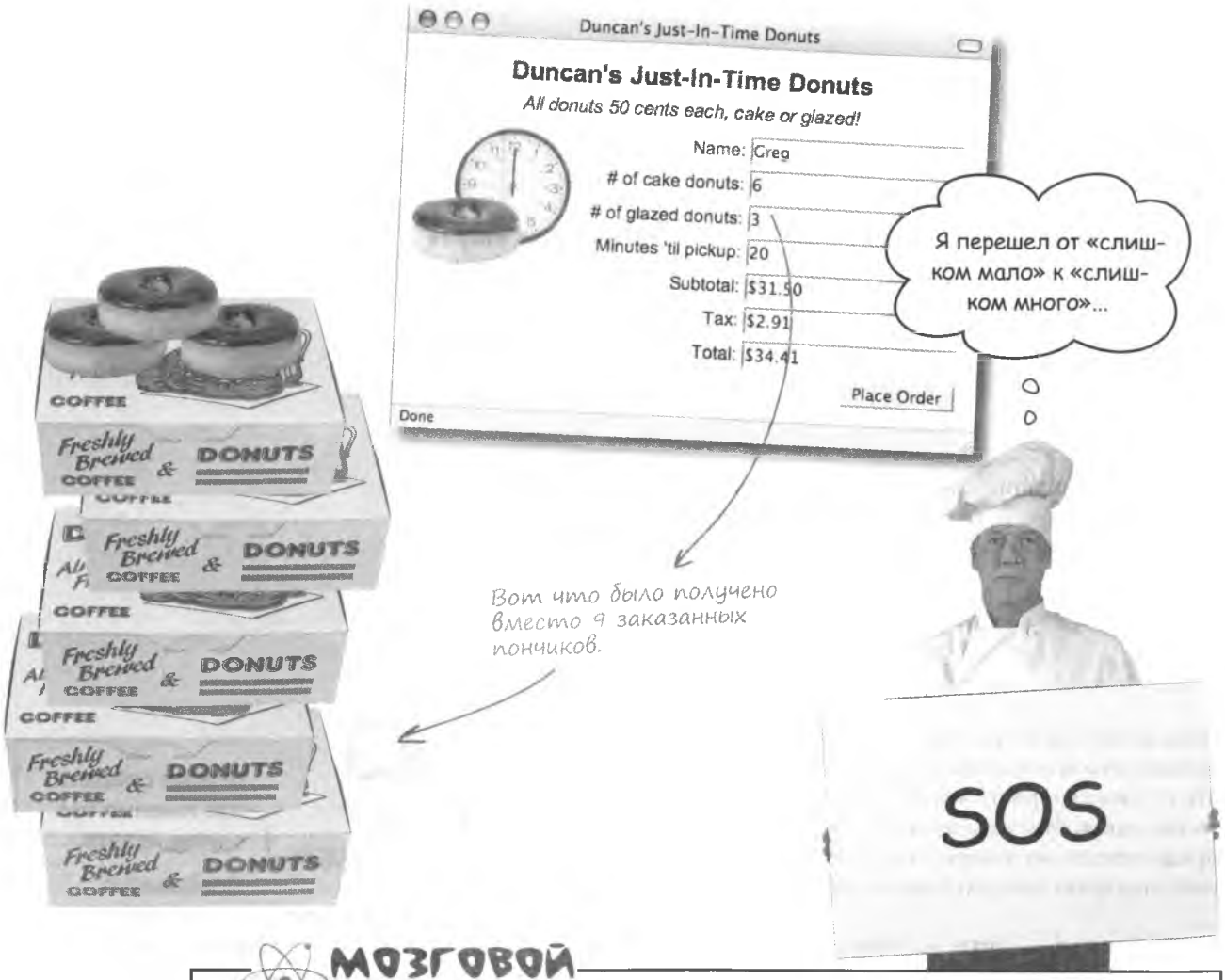
О: Стиль верблюда с заглавной буквой первого слова используется для именования объектов JavaScript, о которых мы поговорим в главе 9. Для переменных и методов используется стиль верблюда, в котором первое слово пишется с прописной буквы. Соответственно, объекту мы присвоим имя `Donut`, методу — имя `getDonut()`, а переменной — имя `numDonuts`. Для констант подобный подход не применяется. Имена констант пишутся полностью заглавными буквами.

В: Существует ли значение NaN для текстовых и логических данных?

О: Теоретически да, так как параметры этих типов не являются числами. В реальности же это не так. Значение `NaN` указывает, что число — это не то, что вы думаете. Другими словами, `NaN` — это не описание данных JavaScript, а индикатор ошибки для численных типов. Это значение обычно появляется в результате вычислений, в которые в качестве параметра по какой-то причине попали нечисловые данные.

А тем временем у Дункана...

У Дункана дела идут все хуже и хуже. Вместо пустых коробок теперь везде пончики — в каждый заказ добавляются лишние. Дункан утонул в жалобах на избыток пончиков и другой выпечки...



МОЗГОВОЙ ШТУРМ

Что может быть не так с обработкой данных о количестве пончиков?

Складывать можно не только числа

В JavaScript многое значит контекст. Особенно важен тип данных, с которыми осуществляется манипуляция в каждом конкретном куске кода. Ведь даже такая простая операция, как сложение, дает разные результаты для данных разных типов.

```
1 + 2 = 3
```

Сложение чисел

Это прекрасно вам знакомая из начальной школы арифметическая операция.

```
"do" + "nuts" = "donuts"
```

Красивое название для «склеивания слов друг с другом».

Соединение строк

При этой операции начало второго слова «приклеивается»к концу первого.

Итак, вооружившись знанием о том, что строки складываются не так, как числа, ответьте, что произойдет при сложении двух чисел в текстовом формате?

```
"1" + "2" = ?
```

Сложение, соединение, что именно?

Язык JavaScript не разбирает смысл текста — с его точки зрения, это всего лишь набор символов. Поэтому тот факт, что строка содержит численные символы, не имеет никакого значения, будет выполнено соединение строк. Если вы предполагали математическую операцию сложения, данный результат окажется неожиданным.

```
"1" + "2" = "12"
```

Так как это строки, а не числа, они «складываются» путем соединения строк.

Вы получите строку, которая никак не напоминает результат математической операции.

Всегда следите, что именно вы складываете.

Соединение строк вместо сложения чисел — не самая редкая ошибка в JavaScript. Если требуется сложить два числа, убедитесь, что они находятся в нужном формате.

Методы parseInt() и parseFloat()

Периодически возникают ситуации, когда требуется сложить числа, сохраненные в текстовом формате. В этом случае вам следует сначала преобразовать строку в число. Для подобных преобразований в JavaScript предусмотрены два метода:

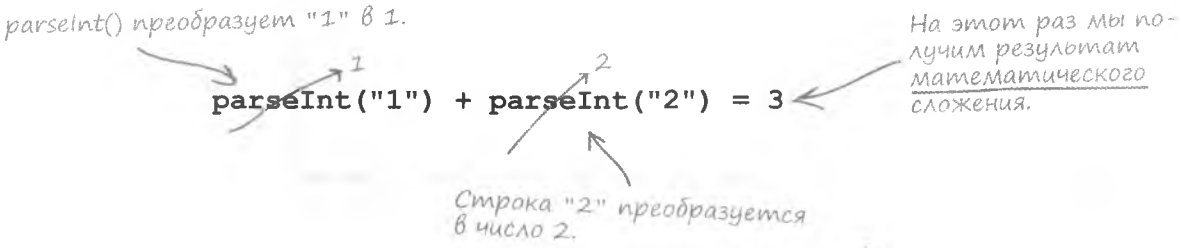
parseInt()

Переданная данному методу строка преобразуется в целое число.

parseFloat()

А этот метод преобразует строку в число с десятичной (плавающей) точкой.

Оба метода берут в качестве параметра строку и преобразуют ее в число:



Следует помнить, что методы `parseInt()` и `parseFloat()` могут и не сработать. Все зависит от параметров, которые вы им передадите. Они превосходно преобразуют строки в цифры, но важно дать им в качестве параметра строку, содержащую только численные символы.

```
parseFloat("$31.50") = NaN
```

Этот код не будет работать, так как метод не понимает символа $.

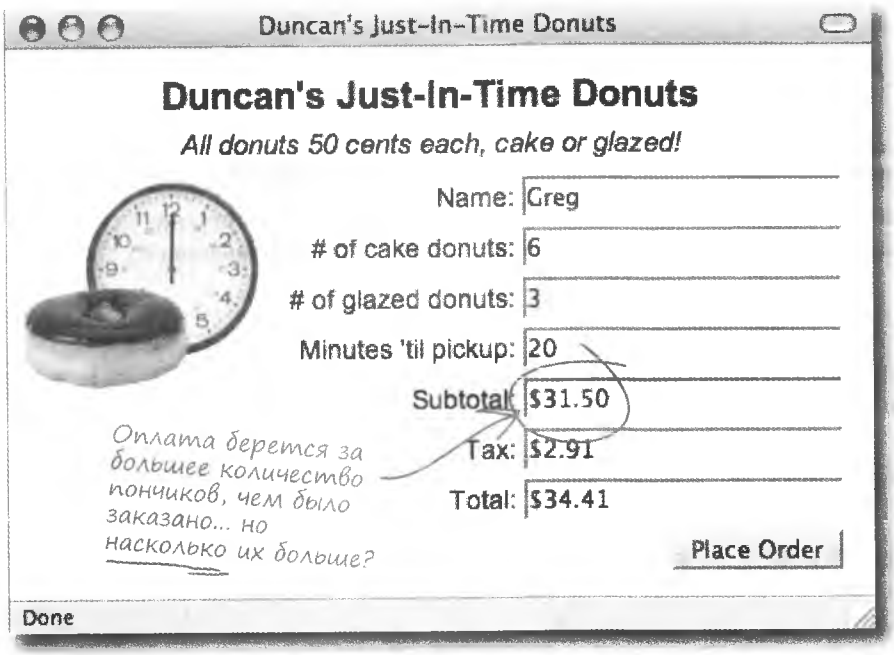
Сюрприз! В результате вы получите все тот же NaN...

РАССЛАБЬТЕСЬ **Если вы до сих пор не поняли принцип работы методов, не волнуйтесь.**

Эта тема будет подробно разбираться чуть позже — пока достаточно знать, что методу можно передать данные и он возвратит вам некий результат.

Откуда берутся лишние пончики?

Взгляните внимательно на форму заказа пончиков. Попытаемся понять, почему возрастает количество заказанного...



Разделив промежуточную сумму на стоимость одного пончика, мы узнаем, сколько же пончиков было заказано.

Промежуточная сумма заказа.

```
$31.50 / $0.50 = 63 пончика
```

Цена одного пончика.

Вот сколько пончиков было заказано на самом деле... странно...

Кажется, мы столкнулись с проблемой объединения строк. Ведь данные, введенные в форму, всегда сохраняются в строковом формате. Даже введенные вами цифры с точки зрения JavaScript являются текстом. Поэтому мы избавимся от ошибки, просто преобразовав строки в числовой формат.

Вы помните, что "1" + "2" = "12"? Кажется, здесь мы столкнулись с аналогичным случаем.

Возьми в руку карандаш

Показанные ниже строчки кода забирают введенную в поля информацию о количестве заказанных пончиков. Воспользуйтесь ими для заполнения недостающих строк метода `updateOrder()` таким образом, чтобы информация о размере заказа была преобразована в числовой формат.

```
document.getElementById("cakedonuts").value
```

Эта строчка кода берет данные о количестве обычных пончиков, введенные пользователем в форму заказа.

А эта строчка получает из формы данные о количестве заказанных глазированных пончиков.

```
document.getElementById("glazeddonuts").value
```

```
function updateOrder() {
  const TAXRATE = 0.0925;
  const DONUTPRICE = 0.50;
  var numCakeDonuts =

    .....
  var numGlazedDonuts =

    .....
  if (isNaN(numCakeDonuts))
    numCakeDonuts = 0;
  if (isNaN(numGlazedDonuts))
    numGlazedDonuts = 0;
  var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
  var tax = subTotal * TAXRATE;
  var total = subTotal + tax;
  document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
  document.getElementById("tax").value = "$" + tax.toFixed(2);
  document.getElementById("total").value = "$" + total.toFixed(2);
}
```

Возьми в руку карандаш

Решение

Итак, вот каким образом нужно было вставить данные в строчки кода, чтобы завершить метод `updateOrder()`.

```
document.getElementById("cakedonuts").value
```

```
document.getElementById("glazeddonuts").value
```

Так как в обоих случаях мы имеем дело с целыми числами, используем для преобразования метод parseInt().

```
function updateOrder() {
  const TAXRATE = 0.0925;
  const DONUTPRICE = 0.50;
  var numCakeDonuts =
    parseInt(document.getElementById("cakedonuts").value);
  var numGlazedDonuts =
    parseInt(document.getElementById("glazeddonuts").value);
  if (isNaN(numCakeDonuts))
    numCakeDonuts = 0;
  if (isNaN(numGlazedDonuts))
    numGlazedDonuts = 0;
  var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
  var tax = subTotal * TAXRATE;
  var total = subTotal + tax;
  document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
  document.getElementById("tax").value = "$" + tax.toFixed(2);
  document.getElementById("total").value = "$" + total.toFixed(2);
}
```

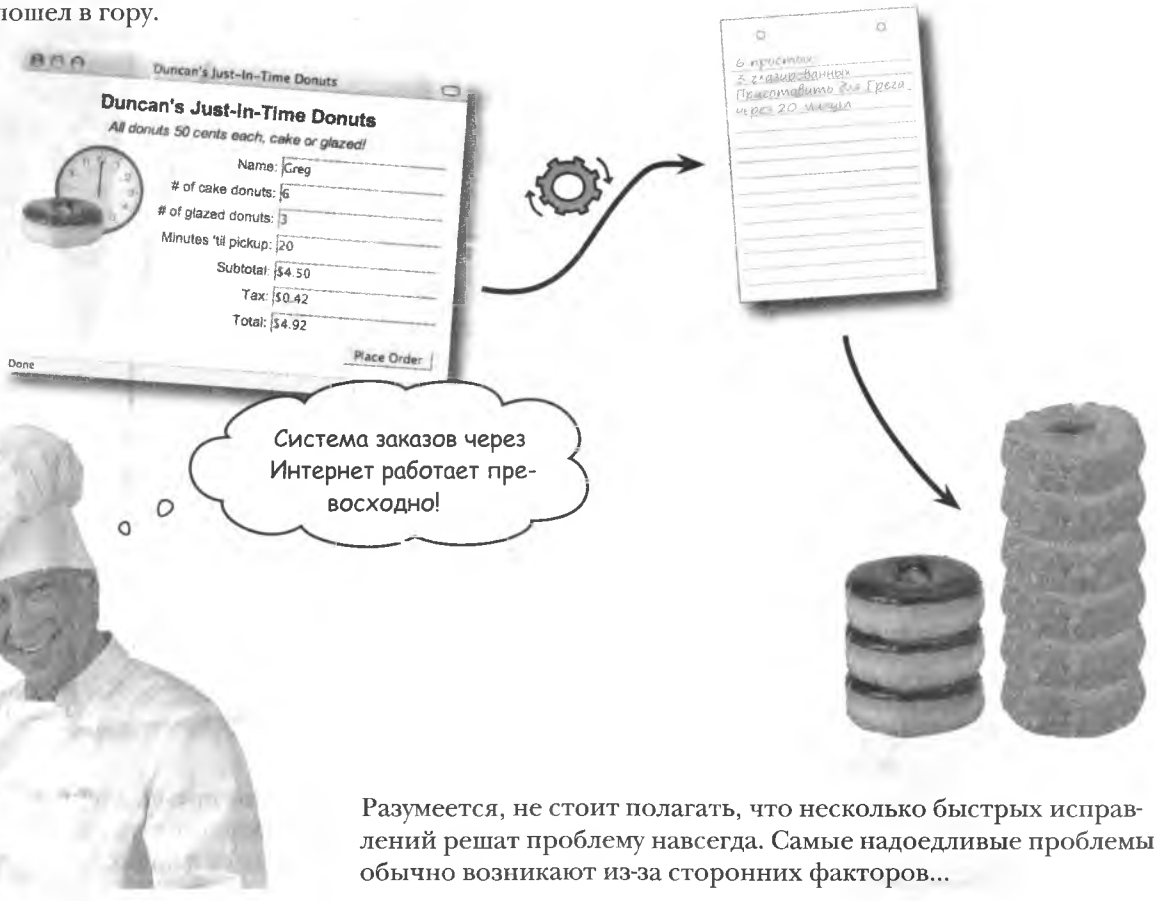
Метод toFixed() округляет сумму в долларах, оставляя после запятой всего два знака.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Имена **констант** имеет смысл писать **ПРОПИСНЫМИ БУКВАМИ**, а имена **переменных** — с использованием **стиляВерблюда**.
- **Всегда инициализируйте константы** в момент создания, а переменные по мере возможности.
- Неинициализированная переменная имеет статус **undefined**, пока ей не будет присвоено значение.
- `NaN` означает **не число** и используется для указания на отсутствие численных данных там, где они должны быть.
- Соединение строк отличается от математического сложения, хотя и выполняется одним оператором (+).
- Встроенные методы `parseInt()` и `parseFloat()` **преобразуют строки в числа**.

Вы решили проблему...

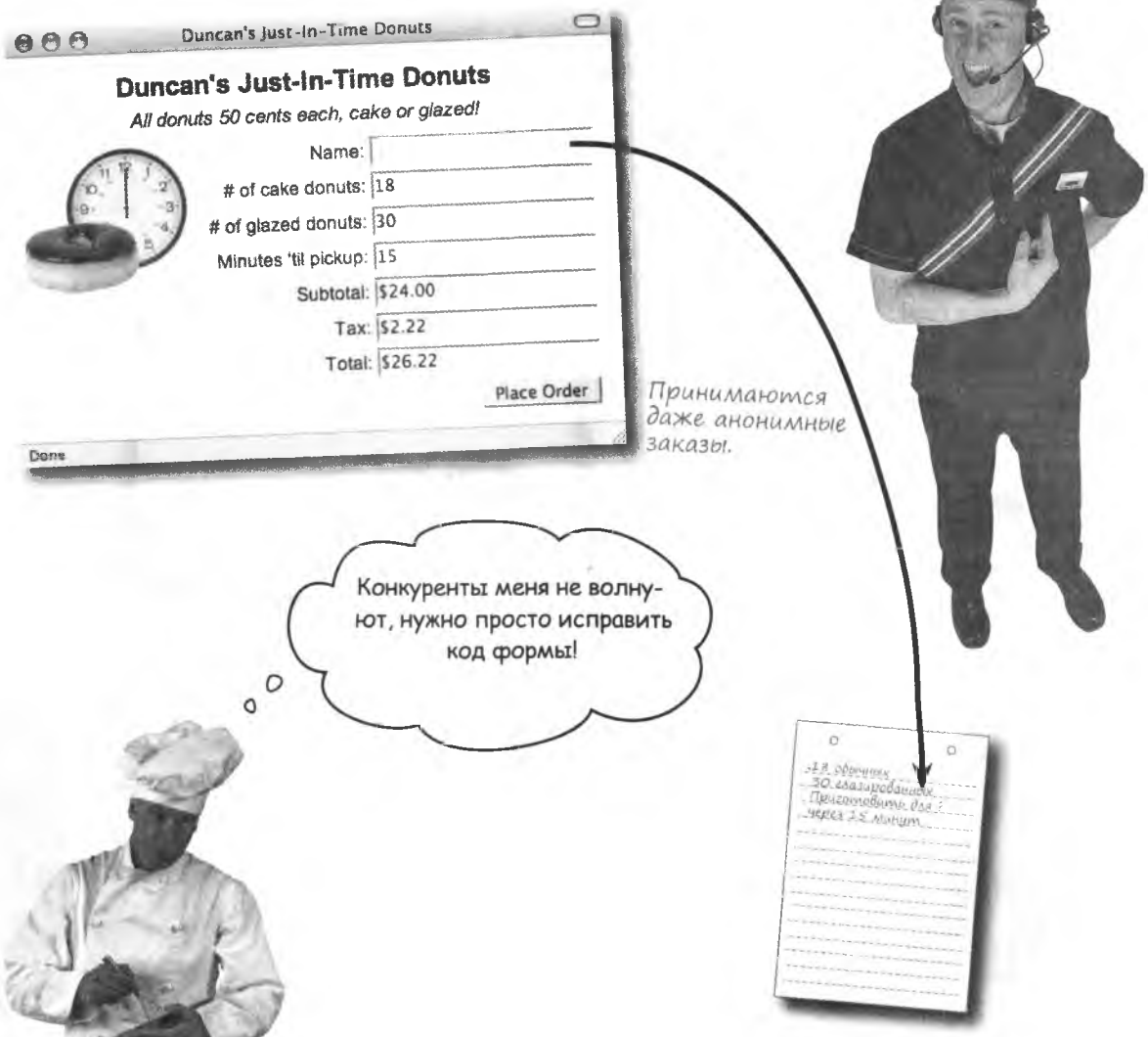
Дункан восхищен исправлениями, которые вы внесли в код JavaScript. Теперь с получаемыми им заказами все в порядке и бизнес пошел в гору.



Разумеется, не стоит полагать, что несколько быстрых исправлений решат проблему навсегда. Самые надоедливые проблемы обычно возникают из-за сторонних факторов...

Дункан обнаруживает шпиона

У Дункана новая проблема: пронырливый конкурент Френки. Френки торгует хот-догами на той же самой улице. Чтобы навредить Дункану, он отправляет через Интернет фальшивые безымянные заказы. В итоге Дункану приходится работать впустую.



Дункан впустую тратит драгоценное время, энергию и пончики, выполняя фальшивые заказы. Нужно сделать так, чтобы форма отправлялась только после ввода всех данных.

Метод getElementById()

Для проверки корректности вводимых в форму данных требуется способ получения этих данных со страницы. Ключом, дающим JavaScript доступ к элементам страницы, является атрибут `id` в теге HTML:

```
<input type="text" id="cakedonuts" name="cakedonuts" />
```

Атрибут id используется для доступа к полям формы в коде JavaScript.

Вводимый в форму элемент представляет количество обычных пончиков (cake donuts).

JavaScript позволяет восстанавливать элементы веб-страницы по их номеру ID при помощи метода `getElementById()`. Этот метод не забирает данные напрямую, а представляет поле HTML в виде объекта JavaScript. Для доступа к данным достаточно воспользоваться свойством `value` этого поля.

Формально getElementByID() не функция, а метод объекта документа.

document.getElementById()

Взяв в качестве параметра ID элемента страницы, метод возвращает вам сам элемент, который затем можно использовать для доступа к данным.

Метод getElementById() принадлежит объекту документа.

Не волнуйтесь по поводу объектов, методов и свойств!

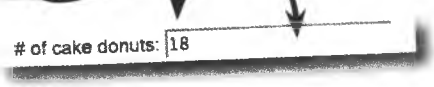
В JavaScript поддерживается усовершенствованный тип данных, называемый объектом, позволяющий делать потрясающие вещи. Да и сам язык JavaScript по сути является набором объектов. Но об этом мы поговорим позднее. Пока же вам достаточно помнить, что метод подобен математической функции, а свойство — это своего рода переменная.

```
document.getElementById("cakedonuts")
```

ID — это ключ доступа к элементам.

```
document.getElementById("cakedonuts").value
```

Свойство value дает доступ к данным.

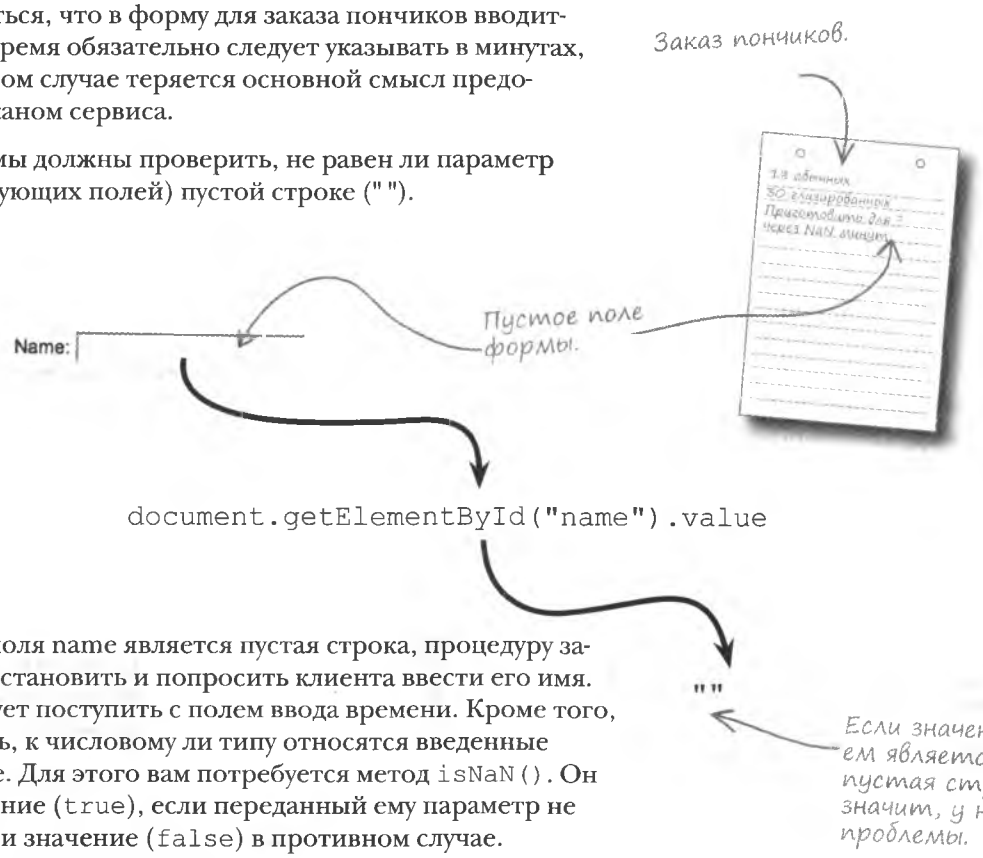


Вооружившись этим кодом, проверим форму Дункана на корректность вводимых в нее данных.

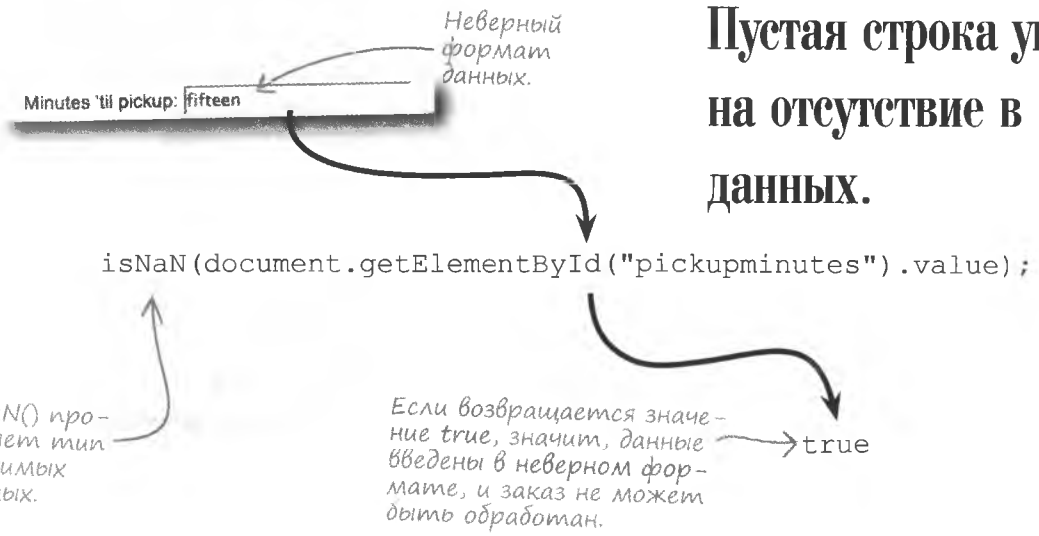
Проверка данных формы

Вам нужно убедиться, что в форму для заказа пончиков вводится имя клиента. Время обязательно следует указывать в минутах, так как в противном случае теряется основной смысл предоставляемого Дунканом сервиса.

В данном случае мы должны проверить, не равен ли параметр value (соответствующих полей) пустой строке (" ").



Если значением поля name является пустая строка, процедуру заказа следует приостановить и попросить клиента ввести его имя. Аналогично следует поступить с полем ввода времени. Кроме того, следует проверить, к числовому ли типу относятся введенные в это поле данные. Для этого вам потребуется метод `isNaN()`. Он возвращает значение (`true`), если переданный ему параметр не является числом, и значение (`false`) в противном случае.



Пустая строка указывает на отсутствие в форме данных.

Магниты JavaScript

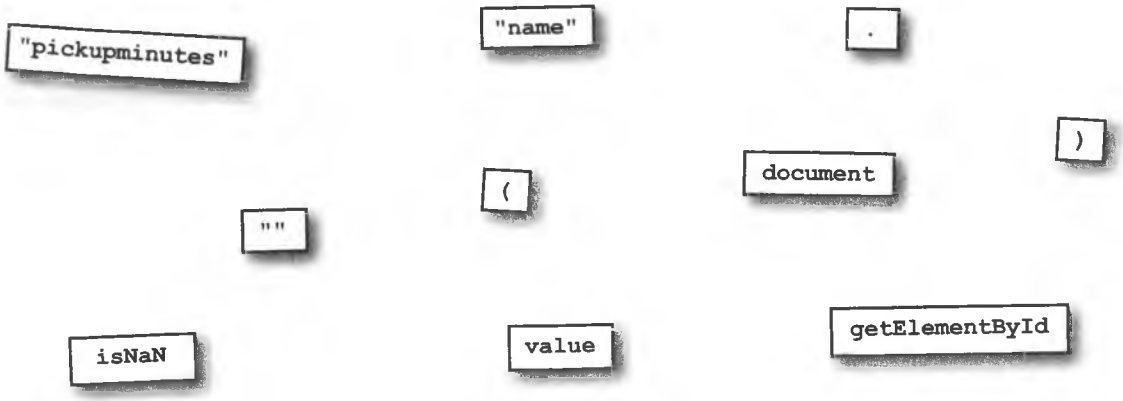
Метод `placeOrder()` выполняет проверку корректности введенных в поля данных. Воспользуйтесь магнитами, чтобы получить код, проверяющий ввод имени клиента и времени в минутах, а также то, что во второе поле введено именно число. Вам потребуются все магниты, а некоторые из них будут использоваться больше одного раза.

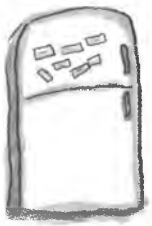
Оператор "if" проверяет соблюдение условия и в зависимости от результата осуществляет некое действие.

Это проверка равенства.

Это означает, что к действиям приводит одно из двух условий — если так ИЛИ так, то следует некое действие.

```
function placeOrder() {
  if (.....==.....)
    alert("I'm sorry but you must provide your name before submitting an order.");
  else if (.....||
    .....)
    alert("I'm sorry but you must provide the number of minutes until pick-up" +
      " before submitting an order.");
  else
    // Отправка заказа на сервер
    form.submit();
}
```





Решение задачи с магнитами

Вот как выглядит код метода `placeOrder()`, проверяющего корректность ввода данных о пользователе и времени до доставки заказа, после заполнения пустых мест.

Если имя не введено, то появляется окно с сообщением.

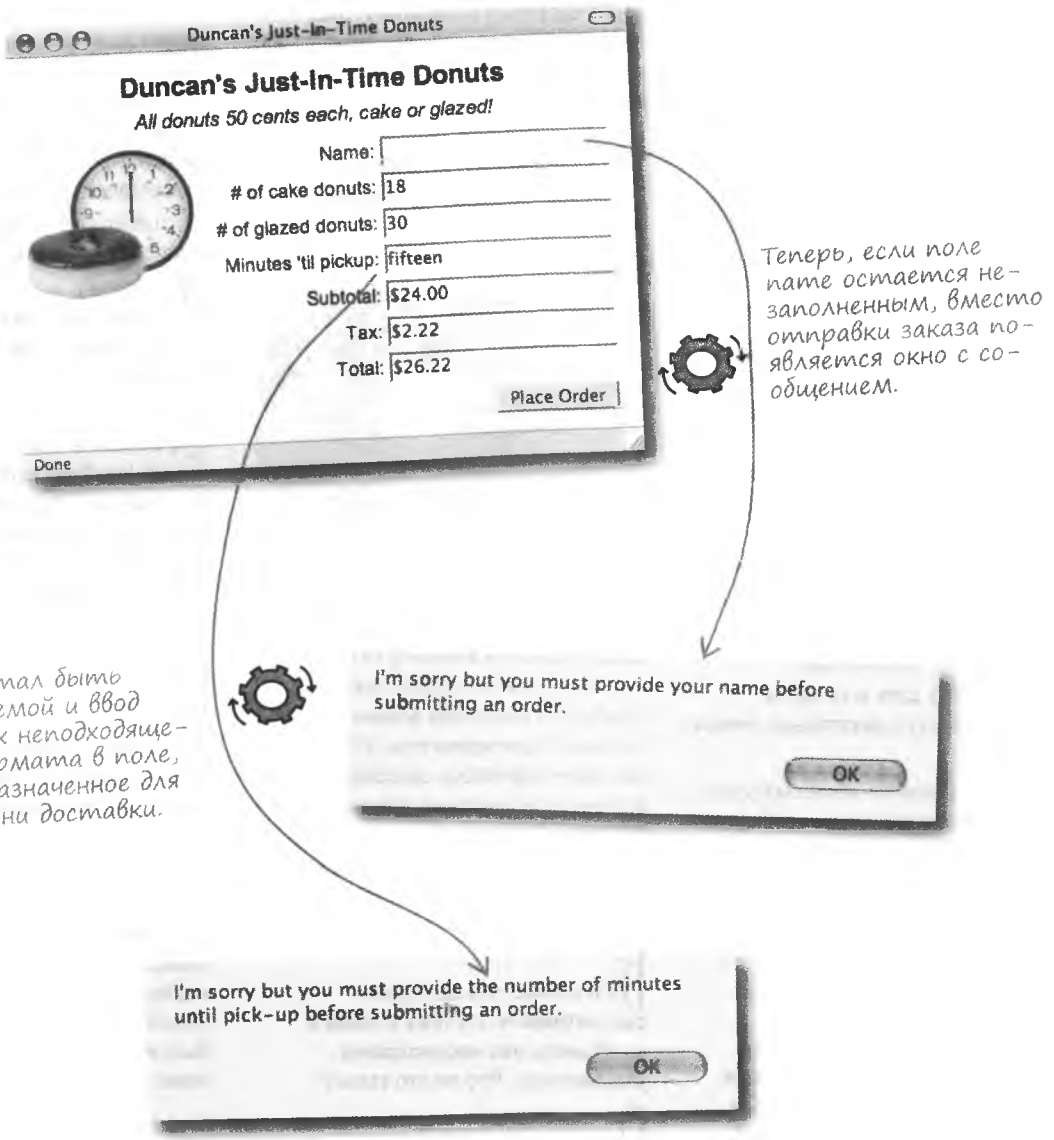
Здесь проверяется, не равняется ли значение поля name пустой строке " ".

Эту фразу с компьютерного языка на обычный можно перевести так: «если значением является пустая строка ИЛИ значение не является числом».

```
function placeOrder() {
  if (document . getElementById ( "name" ) . value == "" )
    alert("I'm sorry but you must provide your name before submitting an order.");
  else if ( document . getElementById ( "pickupminutes" ) . value "" ||
    isNaN ( document . getElementById ( "pickupminutes" ) . value )
    alert("I'm sorry but you must provide the number of minutes until pick-up" +
      " before submitting an order.");
  else
    // Отправка заказа на сервер
    form.submit();
}
```

И снова вы спасли Дункана!

Новая, улучшенная форма приема заказов положила конец вредоносной деятельности Френки и сделала страницу более устойчивой. Применение JavaScript для защиты целостности вводимых клиентами данных – беспроигрышный вариант, особенно в жестоком ресторанном бизнесе!



Часто Задаваемые Вопросы

В: Как оператор (+) распознает, складывать ему или соединять?

О: Как это часто бывает в JavaScript, функциональность определяется контекстом. То есть оператор смотрит, что он «складывает», и определяет, осуществлять ему арифметическое сложение или соединение строк. Проблемы возникают, когда вы путаете тип данных. Это дополнительная причина, по которой стоит перепроверять данные перед операцией сложения.

В: Что произойдет при попытке сложить строку и число?

О: Так как в JavaScript преобразование чисел в строки происходит автоматически, такая операция приведет к объединению строк. Поэтому сначала число преобразуется в строку, а потом две строки соединяются. Чтобы сложить два числа, сначала преобразуйте строку методом `parseInt()` или `parseFloat()`.

В: Что будет, если методу `parseInt()` дать в качестве параметра строку с десятичной точкой?

О: Ничего страшного. JavaScript решит, что вам просто не интересна дробная часть, и вернет только целую.

В: Каким образом атрибут `id` связывает элементы с кодом JavaScript?

О: Представим атрибут `id` в виде портала, через который код JavaScript получает доступ к коду HTML. Говоря, что код JavaScript запускается на веб-странице, обычно имеют в виду не саму страницу, а браузер. На самом деле код JavaScript изолирован от кода HTML и получает доступ к нему при помощи специальных механизмов. Один из таких механизмов включает в себя атрибут `id`, позволяющий языку JavaScript воспользоваться элементом HTML. Пометив элемент страницы атрибутом ID, вы даете JavaScript возможность работать с ним.

В: А как именно код JavaScript осуществляет доступ к элементам HTML-кода?

О: Метод `getElementById()` объекта `document` является ключом, дающим доступ к элементу HTML из JavaScript. Этот метод использует атрибут `id` для поиска элемента на странице. Идентификаторы HTML, как и идентификаторы JavaScript, должны быть уникальны в пределах страницы. В противном случае метод `getElementById()` не сможет определить, какой из элементов следует вернуть.

В: Я помню, что мы будем рассматривать эту тему в главе 9, но объекты уже неоднократно упоминались. Что же это такое?

О: Мы в данном случае слегка забегаем вперед. Объектами называется усовершенствованный тип данных JavaScript, объединяющий в себе методы, константы и переменные. Метод — это всего лишь функция, являющаяся частью объекта, в то время как свойство является его переменной или константой. С практической точки зрения JavaScript использует объекты для представления всего на свете — и окно браузера, и веб-страница являются объектами. Именно поэтому вызов метода `getElementById()` осуществляется посредством объекта `document`. А теперь вернемся к материалу главы 2...

В: Я так и не понял, в чем разница между элементом веб-страницы и его значением.

О: Элементы веб-страницы с точки зрения JavaScript являются объектами. Это означает, что у них есть свойства и они могут управляться методами. Одним из таких свойств является `value`, содержащее хранящееся в элементе значение. Например, *значением* поля формы являются введенные в него данные.

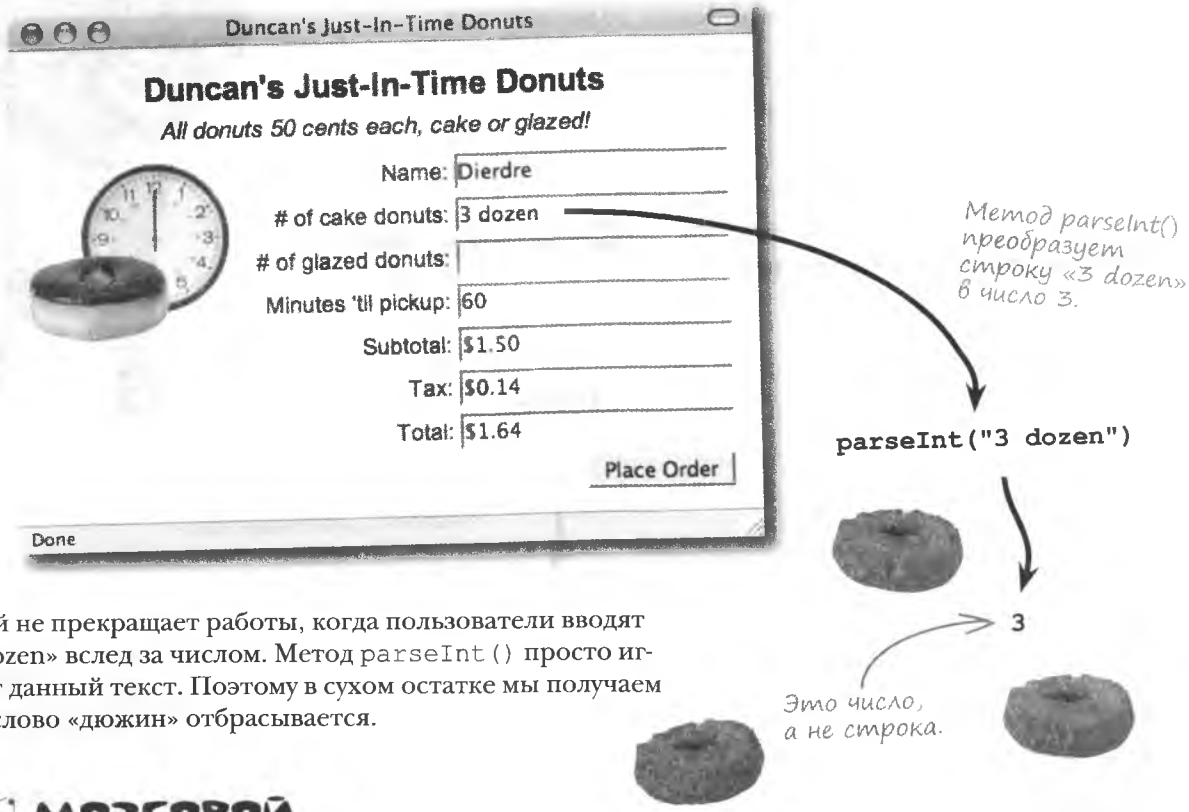
В: Зачем мне знать, что значение *не* является числом? Может быть, лучше проверять, *является* ли оно числом?

О: В большинстве случаев предполагается, что вы имеете дело с числами, поэтому имеет смысл проверять исключения. Этим вы делаете код более устойчивым и избегаете странных вычислений, в которые могут быть включены данные неподходящих типов.

Интуитивный ввод данных

Теперь, когда Дункан решил основные проблемы, он хочет улучшить форму заказа пончиков. На вывеске его заведения красуется «горячий пончик», и всем проходящим мимо сразу понятно, что находится внутри. Такой же интуитивно понятной Дункан мечтает сделать форму заказа. Он знает, что пончики обычно берут дюжинами. Редко кто просит 12 или 24 штуки — спрашивают 1 или 2 дюжины. Так почему бы не дать клиентам возможность вводить данные в привычном им виде.

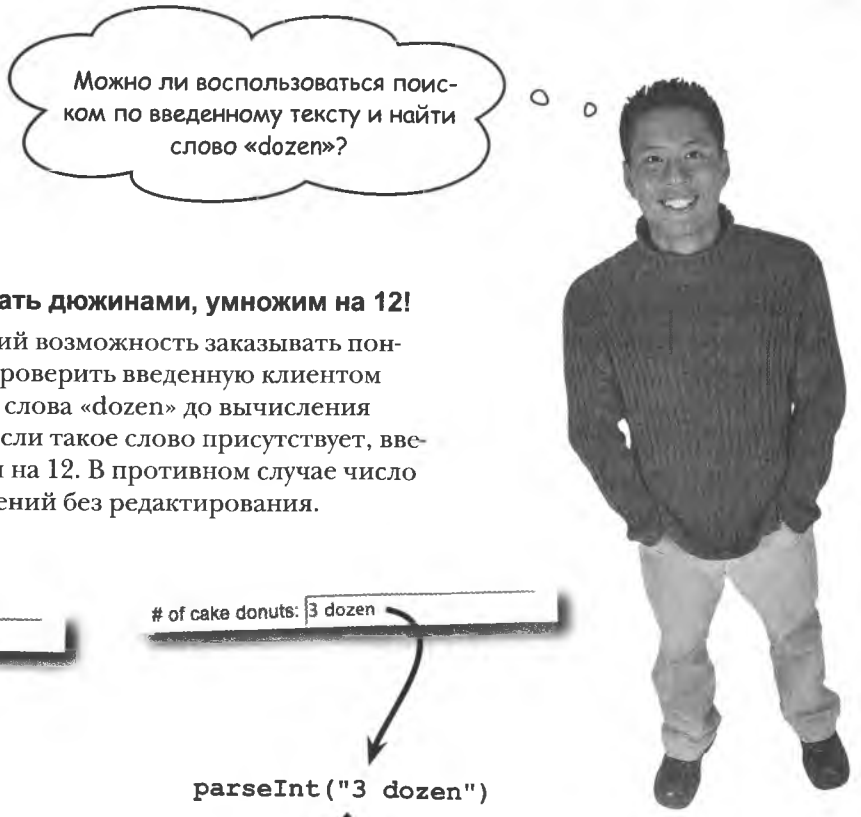
Проблема в том, что наш сценарий не воспринимает слово «дюжина» как руководство к действию и перестает понимать, сколько же пончиков требуется.



Сценарий не прекращает работы, когда пользователи вводят слово «dozen» вслед за числом. Метод `parseInt()` просто игнорирует данный текст. Поэтому в сухом остатке мы получаем число, а слово «дюжин» отбрасывается.

МОЗГОВОЙ ШТУРМ

Можно ли сделать так, чтобы пользователь вводил или количество пончиков одним числом, или же число дюжин и слово «dozen» следом? Как этого добиться?



Если клиент хочет считать дюжинами, умножим на 12!

Чтобы добавить в сценарий возможность заказывать пончики дюжинами, нужно проверить введенную клиентом информацию на наличие слова «dozen» до вычисления промежуточной суммы. Если такое слово присутствует, введенное число умножается на 12. В противном случае число используется для вычислений без редактирования.

Пользовательский метод `parseDonuts()` обрабатывает информацию о количестве заказанных пончиков. Сначала он преобразует введенные данные в численный формат, затем проверяет наличие там слова «dozen». При обнаружении этого слова введенное число уножается на 12. Получите рецепт по адресу *http://www.headfirstlabs.com/books/hfjs/*.

```
function parseDonuts(donutString) {
  numDonuts = parseInt(donutString);
  if (donutString.indexOf("dozen") != -1)
    numDonuts *= 12;
  return numDonuts;
}
```

Проверяем, есть ли во введенных клиентом данных слово «dozen».

Умножаем число пончиков на 12.

Анализируя дюжины пончиков...

Метод `parseDonuts()` вызывает метод `updateOrder()`, вычисляющий на основе введенных клиентом данных промежуточную и конечную суммы.

Инициализация двух констант.

Получаем число заказанных пончиков из поля формы.

```
function updateOrder() {
  const TAXRATE = 0.0925;
  const DONUTPRICE = 0.50;
  var numCakeDonuts = parseDonuts(document.getElementById("cakedonuts").value);
  var numGlazedDonuts = parseDonuts(document.getElementById("glazeddonuts").value);
  if (isNaN(numCakeDonuts))
    numCakeDonuts = 0;
  if (isNaN(numGlazedDonuts))
    numGlazedDonuts = 0;
  var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
  var tax = subTotal * TAXRATE;
  var total = subTotal + tax;
  document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
  document.getElementById("tax").value = "$" + tax.toFixed(2);
  document.getElementById("total").value = "$" + total.toFixed(2);
}
```

Если в поля для ввода данных о количестве пончиков были введены не числа, присвоим этим параметрам значение 0.

Считаем промежуточную сумму, налог и полную сумму.

Показываем на странице сумму в долларах.

Округляем сумму в долларах до двух знаков после запятой (количество центов).

Полный успех предприятия!

Идея Дункана о доставке горячих пончиков к указанному клиентом времени заработала благодаря JavaScript, который позволяет тщательно проверить вводимые в форму данные.

Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Чего мы все хотим для данных нашего сценария?

Ум хорошо, а два лучше!

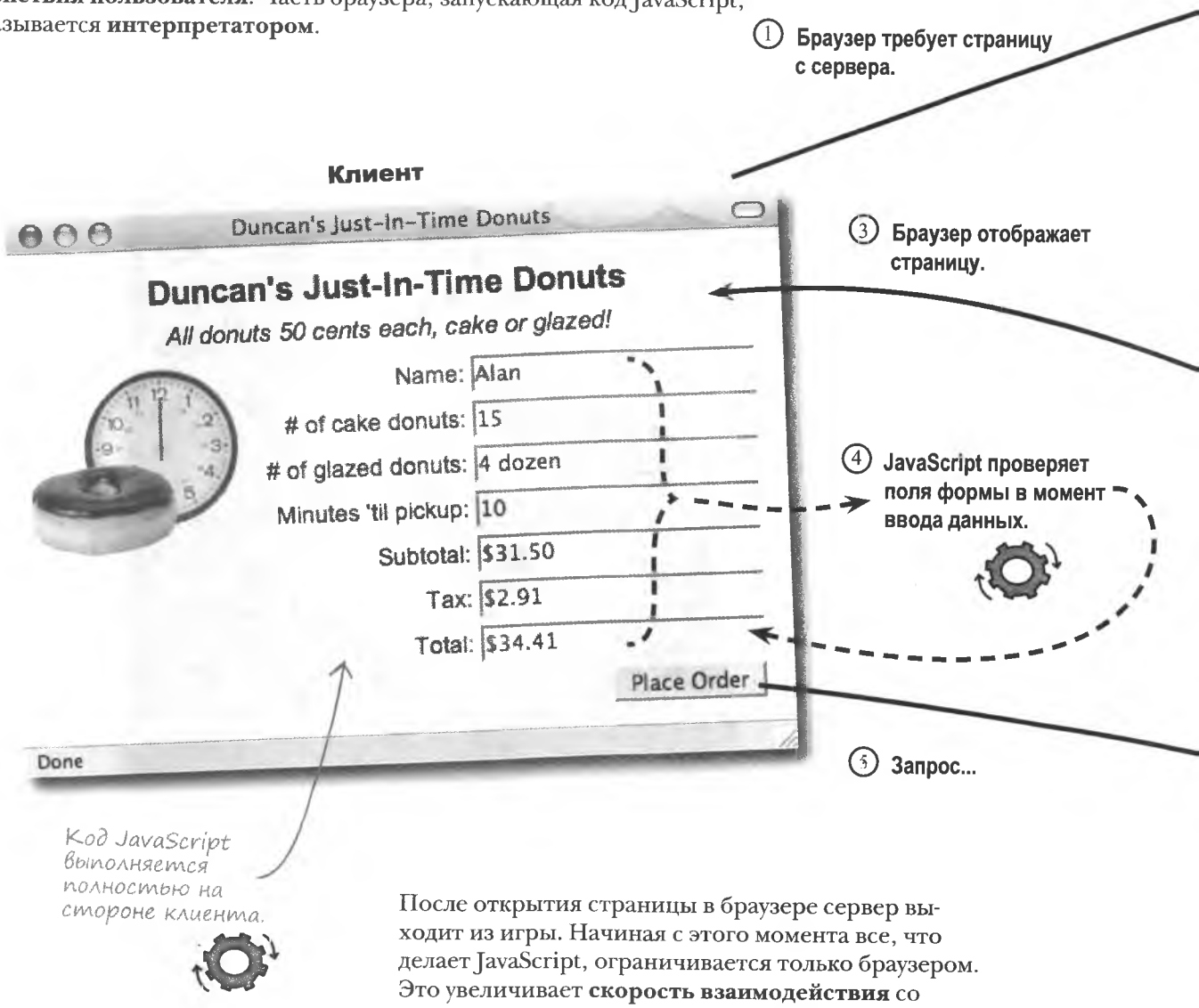
Вводимым пользователями данным доверять нельзя. Предполагать, что пользователи при вводе данных будут их проверять, неразумно. В этом деле следует положиться на JavaScript.

Знакомство с браузером

Иногда JavaScript хочет знать, что происходит в окружающем мире. Ваши сценарии могут существовать в качестве кода на веб-страницах, но по большей части они живут в мире, создаваемом браузером или клиентом. **Умным сценариям** часто необходимо знать больше о мире, в котором они живут, в этом случае они могут **общаться с браузером**, чтобы узнать про него как можно больше. Независимо от того, что требуется узнать: размер экрана или нажата ли кнопка в браузере, они постоянно поддерживают отношения с браузером.

Клиент, сервер и JavaScript

После щелчка на ссылке или ввода URL в адресную строку браузер требует указанную страницу с сервера и доставляет ее **клиенту**. JavaScript не начинает работу, пока страница не будет доставлена. Его код встроен в страницу и работает вместе с браузером, отвечая **на действия пользователя**. Часть браузера, запускающая код JavaScript, называется **интерпретатором**.



После открытия страницы в браузере сервер выходит из игры. Начиная с этого момента все, что делает JavaScript, ограничивается только браузером. Это увеличивает **скорость взаимодействия** со страницей, ведь вам не приходится ждать, пока сервер обработает и вернет данные. Именно поэтому JavaScript называется **языком клиента**.

Запрос страницы

```
GET / HTTP/1.1
Host: www.duncansdonuts.com
Connection: close
Accept-Encoding: gzip
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, ...
Accept-Language: en-us
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.7) ...
```

HTML описывает структуру документа.

Ответ на запрос

```
<html>
  <head>
    <title>Duncan's Just-In-Time Donuts</title>
    <link rel="stylesheet" type="text/css" href="donuts.css" />
    <script type="text/javascript">
      function updateOrder() {
        ...
      }

      function parseDonuts(donutString) {
        ...
      }

      function placeOrder() {
        ...
      }
    </script>
  </head>

  <body>
    <div id="frame">
      <div class="heading">Duncan's Just-In-Time Donuts</div>
      <div class="subheading">All donuts 50 cents each, cake or glazed!</div>
      <div id="left">
        <img src="donuttime.png" alt="Just-In-Time Donuts" />
      </div>
      <div id="right">
        ...
      </div>
    </div>
  </body>
</html>
```

CSS делает страницу красивой.

JavaScript добавляет интерактивность. В данном случае он проверяет введенные пользователем данные.

② Сервер отправляет запрошенную страницу.

⑤ ...отправлен...

⑤ ...на сервер.

Какие еще задачи разумнее выполнять на стороне клиента, а не на стороне сервера?

Что браузер может сделать для Вас?

Ваш **браузер отвечает за запуск кода** JavaScript, давая сценариям допуск к клиентской среде. Например, сценарий может узнать ширину и высоту окна браузера или историю посещений веб-страницы. Другими интересными функциями браузеров, открытыми для JavaScript, являются таймеры и возможность сохранения куки. Так называются создаваемые серверами фрагменты данных, которые хранятся на компьютере пользователя даже после того, как он ушел со страницы или вообще закрыл окно браузера.

Измерения браузера

Вы можете узнать не только размер окна браузера и видимой части веб-страницы, но и информацию о производителе, и даже номер версии.

История браузера

С помощью JavaScript вы можете получить доступ к списку последних посещенных страниц и перейти на любую из них, эффективно создавая ваши собственные элементы управления браузером.

Таймеры

Таймеры позволяют запускать код JavaScript по истечении заданного времени.

Куки

Куки подобны переменным, которые сохраняются браузером на жестком диске пользовательского компьютера после завершения веб-сессии. То есть вы можете покинуть страницу, снова на нее вернуться, но они все равно никуда не денутся.

Это далеко не все функции, которые клиент предоставляет сценариям. Мы всего лишь попытались дать вам представление о том, что JavaScript умеет намного больше, чем просто присутствовать на странице. Более того, часто возникают ситуации, когда имеет смысл выглянуть **за пределы страницы** и получить от браузера помощь.

Часто Задаваемые Вопросы

В: Итак, JavaScript является частью клиента?

О: Да. Поддерживающие JavaScript браузеры имеют встроенный интерпретатор, отвечающий за чтение и запуск кода на странице.

В: Если код JavaScript запускается на стороне клиента, как он связан с сервером?

О: Код JavaScript не имеет прямого доступа к серверу, так как он работает на стороне клиента. Обычно он используется для перехвата данных, транслируемых с сервера в браузер. Но можно написать и сценарий, который будет запрашивать информацию с сервера, а затем обрабатывать ее и отображать на странице. Эта техника называется Ajax. О ней мы поговорим в главе 12.

В: Позволяет ли JavaScript контролировать клиента?

О: И да и нет. Браузеры дают JavaScript доступ к определенным частям клиентской среды, но он весьма ограничен по причинам безопасности. Например, большинство браузеров не позволяет сценариям открывать и закрывать окна без согласия пользователя.

Объект iRock слишком счастлив

Помните объект iRock? Ваш код JavaScript имел такой успех, что его выкупил молодой предприниматель Ален. Но он снова обратился к вам, так как появилась проблема... Пользователей раздражает **вечное счастье** объекта iRock. Разумеется, мы все хотим, чтобы наши домашние любимцы были счастливы, но в данном случае диапазон эмоций уж слишком ограничен.

Итак, вам следует подстроиться под **ожидания пользователей**. Постараемся сделать вашего виртуального домашнего любимца как можно более реальным. Вам следует понять, как нужно поменять поведение объекта iRock. И кажется, в решении проблемы вам может в некоторой степени помочь браузер клиента.

iRock должен быть более отзывчивым

Рассмотрим возможные варианты поведения объекта iRock, которые сделают его более **реалистичным** и привлекательным для пользователя, не говоря уже о большей интерактивности. В идеале нам следует увеличить диапазон испытываемых объектом эмоций.

Последнюю версию кода iRock 2.0 вы найдете по адресу http://www.headfirstlabs.com/books/hfjs.

Разгневанный

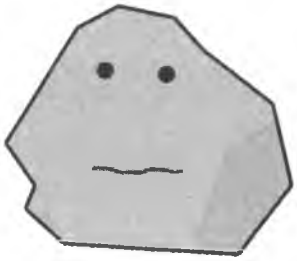
Камешек может без причины разгневаться, и владельцу нужно будет его успокаивать.

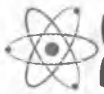
Подавленный

Каждый раз, когда вы закрываете страницу, камешек начинает плакать, требуя, чтобы пользователь оставил окно браузера открытым.

Одинокий

Если долго не обращать внимания на iRock, он начинает чувствовать себя одиноким. Щелкайте на нем периодически, чтобы он чувствовал вашу заботу.





МОЗГОВОЙ ШТУРМ

Какие из этих эмоций имеет смысл приобрести объекту iRock? Каким образом их можно реализовать при помощи JavaScript?

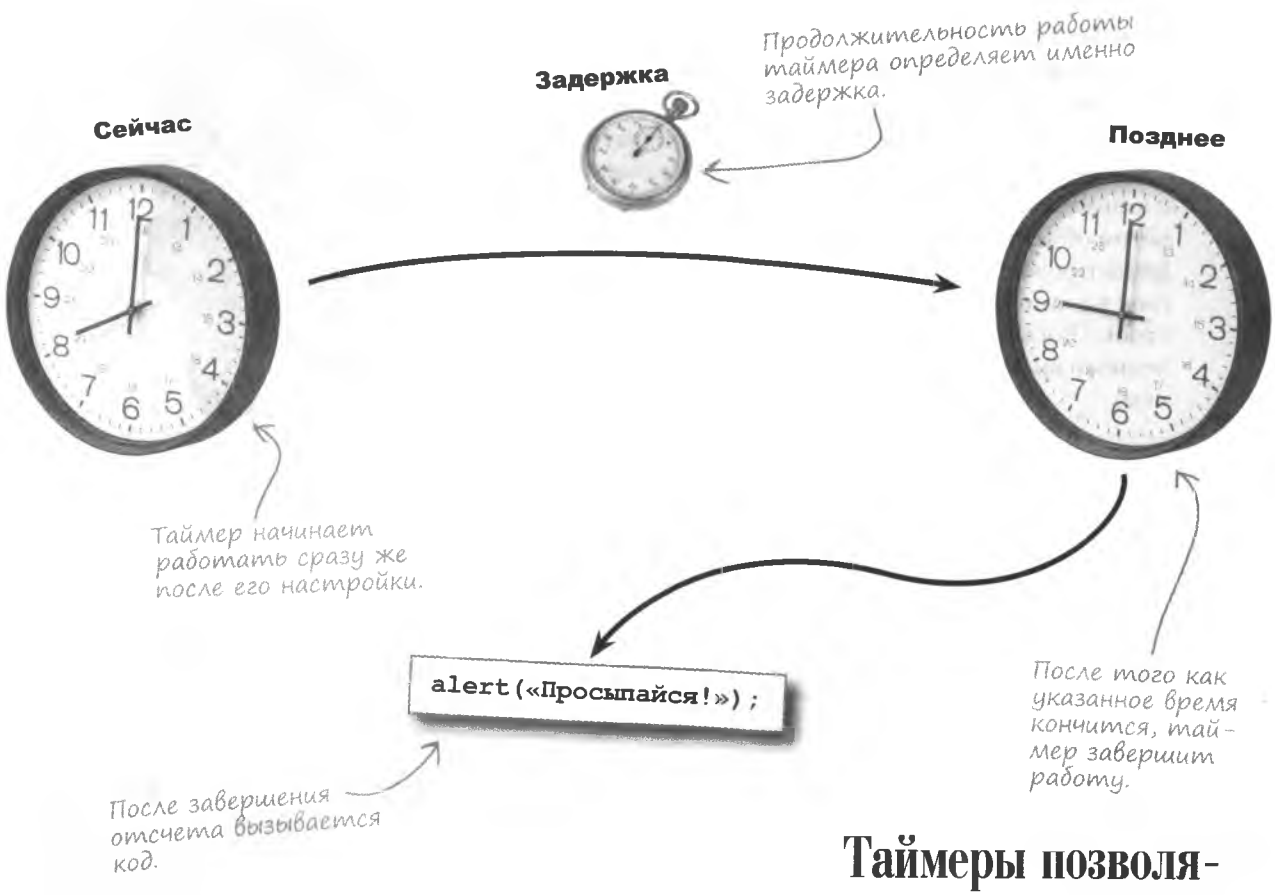
Мне нравится идея, что камешек может почувствовать одиночество, так как именно такое поведение свойственно реальным домашним животным. Можно ли сделать так, чтобы поведение объекта iRock менялось с течением времени?

JavaScript позволяет узнать, когда пользователь выполняет некие действия... И когда он этого не делает.

Было бы интересно заставить камешек испытывать одиночество, так как это подтолкнет пользователя к взаимодействию и вознаградит позитивным ответом объекта iRock. Вопрос в том, как при помощи JavaScript заставить объект iRock со временем менять свое эмоциональное состояние. Нужно сделать так, чтобы iRock начинал грустить, если в течение определенного времени пользователь ни разу не щелкнул на нем.

Таймеры

Таймеры в JavaScript по принципу действия напоминают будильники. Но если будильник вы устанавливаете на определенное время, то в случае таймера следует указать, сколько времени осталось до события. По прошествии этого времени выполняется указанный кусок кода.



По истечении заданного таймером времени вы можете запустить любой код. Если содержимое страницы регулярно редактируется, таймер можно использовать для ее периодического обновления. В других случаях таймер помогает определить отсутствие активности пользователя в течение заданного времени.

Таймеры позволяют запускать код JavaScript через нужное вам время.

Как работает таймер

Чтобы запустить таймер, вам нужно 1) указать время задержки; 2) указать код, который следует запустить после указанного времени. Таймер начинает работу сразу же после его настройки.

Время задержки отсчитывается в миллисекундах, то есть в тысячных долях секунды. К примеру, 2 секунды это 2000 миллисекунд.

Отобразить сообщение.

```
alert(«Просыпайся!»);
```

```
refresh(); setTimeout(refresh, 120000);
```

Обновить страницу.

Установить следующий таймер.

Код, который запускается по истечении срока действия таймера, зависит только от вашего желания. Это может быть один оператор, набор операторов (каждый из которых завершается точкой с запятой), и даже вызов встроенного или пользовательского метода.

После завершения отсчета таймер запускает код JavaScript и исчезает. Можно создать таймер, который будет запускать код через определенные промежутки времени, пока вы его не остановите. Но в данном случае нам потребуется одноразовый таймер.

Упражнение

Совместите одинаковые показатели.

500 мс	5 минут
300 000 мс	5 секунд
5000 мс	1/2 секунды

Упражнение
Решение

Вот как соотносятся различные записи.

Метод setTimeout()

В JavaScript одноразовые таймеры запускаются при помощи встроенного метода `setTimeout()`. Для его работы вам нужно указать время задержки и код, который следует запустить (доступен по адресу *http://www.headfirstlabs.com/books/hfsd/*). Рассмотрим небольшой пример:

Метод setTimeout() устанавливает однократный таймер.

Задержка таймера составляет 600 000 миллисекунд, что равняется 600 секундам или 10 минутам.

```
setTimeout("alert('Wake up!');", 600000);
```

Код JavaScript, который следует выполнить по таймеру, передается методу setTimeout() в виде текстовой строки. Вот почему он помещен в кавычки.

После завершения отсчета появляется окно с сообщением.

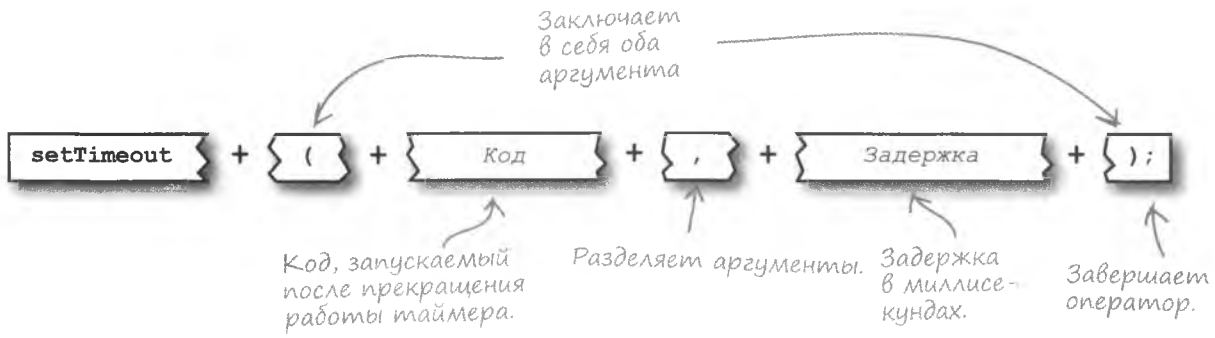
Никогда не отделяйте разряды даже при написании очень больших чисел.

В данном случае метод `setTimeout()` создает таймер, который через 10 минут вызывает окно с сообщением.

600 000 миллисекунд!

Анализ метода setTimeout()

Вот синтаксис метода `setTimeout()`:



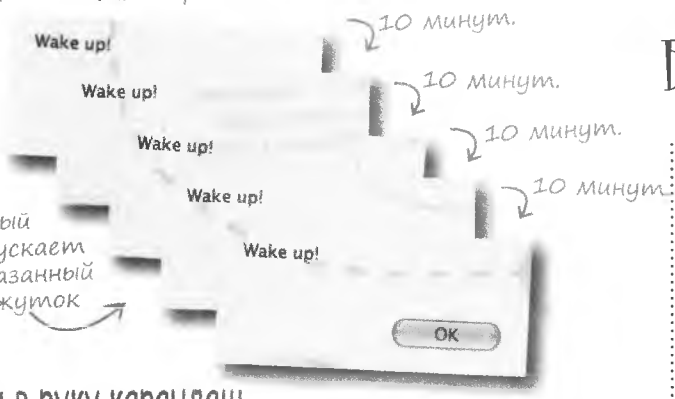
Задание таймера, срабатывающего через определенные промежутки времени, осуществляется с помощью метода `setInterval()`. Его синтаксис аналогичен предыдущему. В результате один и тот же код будет запускаться много раз:

```
var timerID = setInterval("alert('Wake up!');", 600000);
```

Минуты в миллисекундах.

Хранимый отдельно идентификатор таймера.

Устанавливает повторяющийся таймер.



Многократный таймер запускает код через указанный вами промежуток времени.

Будьте осторожны!

Задержка таймера указывается в миллисекундах.

Миллисекунды являются долями секунд, поэтому, забыв, что надо их использовать, можно получить несуразно быстрые или, наоборот, медленные таймеры.

Возьми в руку карандаш

Напишите код, меняющий после 5 минут изображение объекта iRock со счастливого на одинокое. Подсказка: идентификатор изображения элемента iRock называется `rockImg`, а файл с изображением одиночества — `rock.png`.*

Создайте свою версию файла irock.html и протестируйте свой вариант кода перед тем, как перевернуть страницу.

.....

.....

* Этот файл находится по адресу http://www.headfirstlabs.com/books/hfjs/

Возьми в руку карандаш

Решение

Вот как выглядит код, меняющий через 5 минут изображение камешка со счастливого на одинокое.

Кавычки и апострофы позволяют осуществить корректное вложение методов.

```
setTimeout("document.getElementById('rockImg').src = 'rock.png';",
    5 * 60 * 1000);
```

Для перевода в миллисекунды пятиминутной задержки сначала нужно преобразовать все в минуты (x60), затем — в миллисекунды (x1000).

Замена изображения iRock осуществляется подстановкой нового графического файла в атрибут src.

Идентификатор изображения.

Изображение камешка, которому одиноко.

Теперь iRock чувствует одиночество!

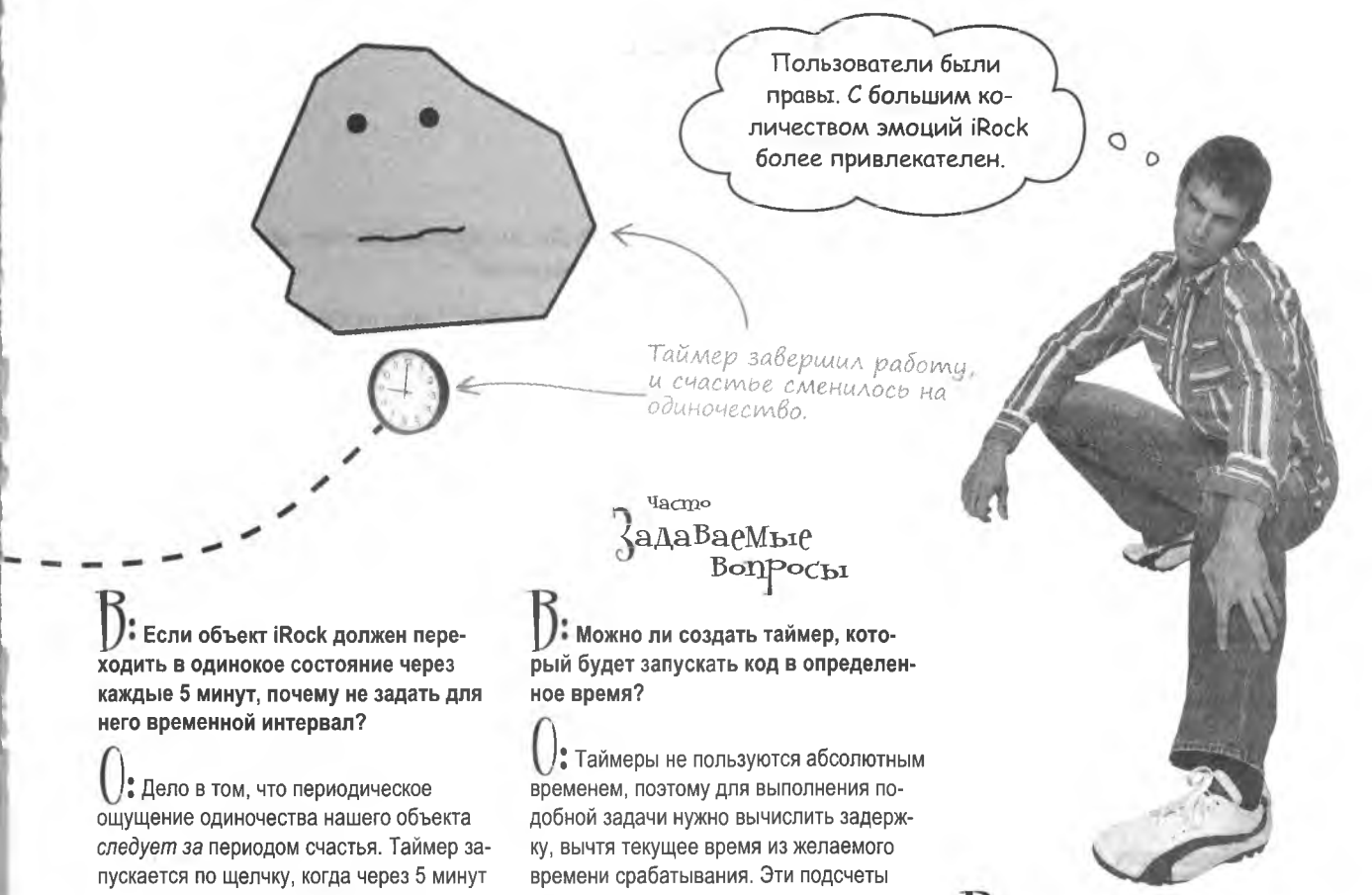
Не забудьте внести вышеуказанные изменения в файл `irock.html` и проверьте, как все работает. Объект iRock должен демонстрировать одиночество, если в течение пяти минут на нем ни разу не щелкнул пользователь. Небольшая временная задержка может привести к тому, что камешек покажется вам чересчур требовательным, но ведь мы и хотели все время держать пользователя занятым. Как будто рядом с ним действительно живое существо со своими нуждами.

Таймер ведет обратный отсчет «счастливого времени» объекта iRock.

Мимолетность счастливого состояния сделала объект iRock более живым.

Подсказка

Ускорить смену эмоциональных состояний объекта можно, уменьшив задержку вызова метода `setTimeout()`. Это позволит протестировать сценарий без долгих ожиданий.



Часто Задаваемые Вопросы

В: Если объект iRock должен переходить в одинокое состояние через каждые 5 минут, почему не задать для него временной интервал?

О: Дело в том, что периодическое ощущение одиночества нашего объекта *следует за* периодом счастья. Таймер запускается по щелчку, когда через 5 минут он завершает работу, объект переходит в грустное состояние и остается в нем до следующего щелчка. Для реализации подобного сценария многократный таймер не подходит, ведь он будет срабатывать каждые 5 минут независимо от действий пользователя.

В: Что произойдет, если закрыть браузер до завершения обратного отсчета?

О: Ничего. Интерпретатор JavaScript при этом прекращает свою работу, и все сценарии JavaScript, в том числе и таймер, завершаются.

В: Можно ли создать таймер, который будет запускать код в определенное время?

О: Таймеры не пользуются абсолютным временем, поэтому для выполнения подобной задачи нужно вычислить задержку, вычтя текущее время из желаемого времени срабатывания. Эти подсчеты выполняются при помощи объекта `Date`, с которым вы познакомитесь в главе 9.

В: Данные на моей странице часто редактируются, поэтому я хотел бы обновлять ее каждые 15 минут. Как это сделать?

О: С помощью метода `setInterval()` установите таймер на 15 минут, то есть на 900 000 миллисекунд (15 × 60 × 1000). Обновление страницы реализуется методом `reload()` объекта `location` вот таким образом:

```
location.reload();
```

Можно также воспользоваться технологией Ajax, с которой вы познакомитесь в главе 12, для динамической загрузки данных.

В: Как остановить таймер, срабатывающий через заданные промежутки времени?

О: Для прекращения работы таймера, запущенного методом `setInterval()`, используется метод `clearInterval()`. Ему следует передать идентификатор останавливаемого таймера, который возвращает метод `setInterval()`. Сохранив возвращенное значение, например, под именем `timerID`, передайте его методу `clearInterval()` вот таким образом: `clearInterval(timerID)`.

БРАУЗЕР О СЕБЕ

Интервью недели: Признания веб-клиента

Head First: Спасибо, что, несмотря на занятость, нашли время и зашли к нам в гости.

Браузер: Да, я крайне занят. Как будто мало мне было HTML и CSS и связанных с ними особенностей визуализации страниц, теперь мне приходится иметь дело еще и с JavaScript. Это совсем другой зверь.

Head First: Что вы имеете в виду? JavaScript дик и неприручен?

Браузер: Конечно, нет. Я употребил слово «зверь» в переносном смысле. Я всего лишь имел в виду новые проблемы, которые появились вместе с JavaScript. Я должен читать этот код, молясь, чтобы он был написан корректно, а потом запускать его, не забывая при этом про HTML и CSS.

Head First: И как вы справляетесь?

Браузер: К счастью, эти три сущности прекрасно работают вместе, хотя иногда JavaScript шалит и искажает HTML-код. Проблема в том, что я никак не могу на это повлиять. Ведь я всего лишь делаю то, что мне приказывают делать.

Head First: То есть вы склонны к подчинению?

Браузер: Можно сказать и так, но более точным будет признать, что больше всего я ценю взаимодействие. Моя задача брать код, который отдает мне сервер, и поступать с ним так, как мне приказывают.

Head First: Даже если код является ошибочным?

Браузер: Я стараюсь решать проблемы, когда вижу их, но это тяжелая работа. Кроме того, это тема для другого обсуждения (из главы 11). А пока давайте поговорим о моей работе в качестве веб-клиента.

Head First: Да, вы правы. Так что же это такое — быть веб-клиентом?

Браузер: Это значит быть принимающим концом канала доставки веб-страниц, запрошенных с сервера.

Head First: А как это связано с JavaScript?

Браузер: Очень тесно. Я выполняю всю работу по отображению страниц и обработке вводимых пользователями данных. А JavaScript в это время во все сует свой нос и вносит изменения. Хотя все не так плохо. Многие вещи я никогда не смог бы сделать самостоятельно.

Head First: Например?

Браузер: Ну, я бы не сделал ничего особенного в ситуации, когда пользователь наводит указатель мыши на изображение или меняет мой размер. А JavaScript позволяет легко поменять вид страницы в ответ на действия пользователя. Я не возражаю, так как код JavaScript запускается на странице и на основе страницы и влияет только на саму страницу.

Head First: Вы говорите про JavaScript как про чужеродную сущность. Но разве это не часть вас?

Браузер: Да, JavaScript определенно является частью меня, но его можно воспринимать и как стороннюю сущность, так как доступ к клиенту (ко мне) он осуществляет посредством ограниченного интерфейса. То есть я не даю JavaScript неограниченного доступа ко всему. Это было бы неосторожно, ведь я не знаю, кто написал сценарий и попросил меня запустить его.

Head First: Понятно. Спасибо, что поделились с нами подробностями жизни клиента.

Browser: Рад, что смог вам помочь.

Зависимость от размера экрана

Только Ален успел обрадоваться новому диапазону эмоций объекта iRock, как компанию захлестнула новая волна пользовательских жалоб. Оказывается, размер объекта не очень стабилен. Некоторые пользователи жалуются на «синдром схлопывания», в то время как другие испытали страх перед «гигантским камнем». Ален доверяет только вам, так что пришло время заработать еще немного денег снова привести iRock в порядок.

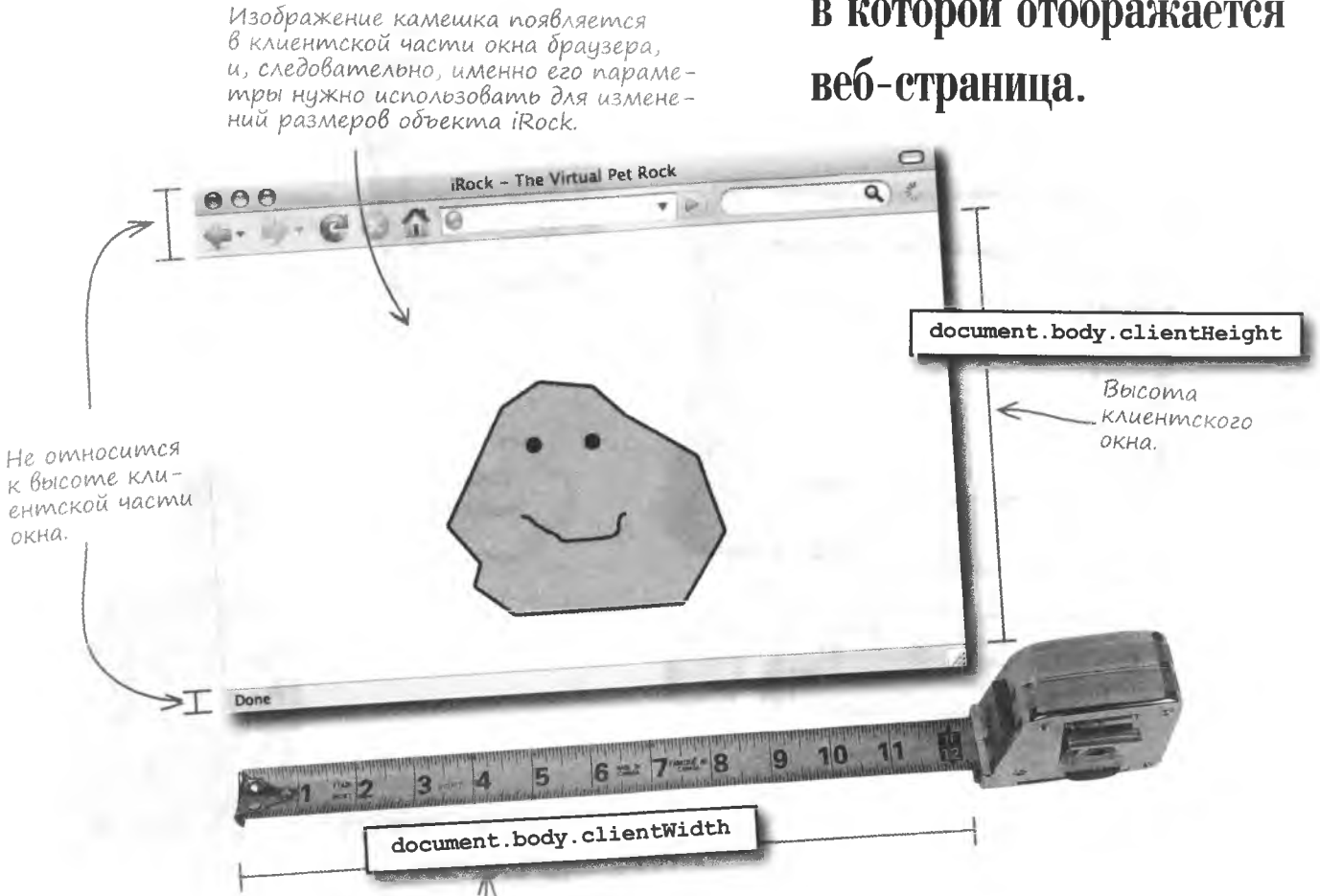
МОЗГОВОЙ ШТУРМ

Почему в разных браузерах камешек имеет разный размер?

Ширина окна браузера

Возникшая проблема связана с тем, что размер объекта iRock не меняется вместе с размером окна браузера. Может показаться, что это хорошо, если не учитывать, что пользователь может заходить в Интернет как с мобильного устройства, так и с настольного компьютера с гигантским монитором. Значит, вам следует заранее определить ширину окна и соответственно отредактировать размер нашего камешка.

Клиентской называется та часть окна браузера, в которой отображается веб-страница.



Важно отличать ширину и высоту клиентской части от общей ширины и высоты окна браузера. Окном клиента считается только та часть, в которой отображается веб-страница. Строка заголовка, панели инструментов и строка состояния сюда уже не входят. Размер объекта iRock вычисляется, исходя из размеров клиентской части.

Задание ширины окна

Размер клиентского окна плотно связан с веб-страницей, доступ к которой в JavaScript осуществляется при помощи объекта `document`. Именно этот объект давал вам доступ к элементам страницы в методе `getElementById()`. Свойства `body.clientWidth` и `body.clientHeight` документа содержат информацию о ширине и высоте клиентского окна.

document

Объект document представляет собой веб-страницу.

```
<html>
  <head>
    <title>iRock - The Virtual Pet Rock</title>
    <script type="text/javascript">
      var userName;
      function greetUser() {
        alert('Hello, I am your pet rock.');
      }
      function touchRock() {
        if (userName) {
          alert("I like the attention, " + userName + ". Thank you.");
        }
        else {
          userName = prompt("What is your name?", "Enter your name here.");
          if (userName)
            alert("It is good to meet you, " + userName + ".");
        }
        document.getElementById("rockImg").src = "rock_happy.png";
        setTimeout("document.getElementById('rockImg').src = 'rock.png';",
          5 * 60 * 1000);
      }
    </script>
  </head>

  <body onload="greetUser();">
    <div style="margin-top:100px; text-align:center">
      <img id="rockImg" src="rock.png" alt="iRock" style="cursor:pointer"
        onclick="touchRock();" />
    </div>
  </body>
</html>
```

document.body

Тело документа представляет собой видимую часть страницы, включая ширину и высоту клиентского окна.

Часто Задаваемые Вопросы

В: И все-таки, чем различаются веб-клиент, браузер, окно клиента и окно браузера?

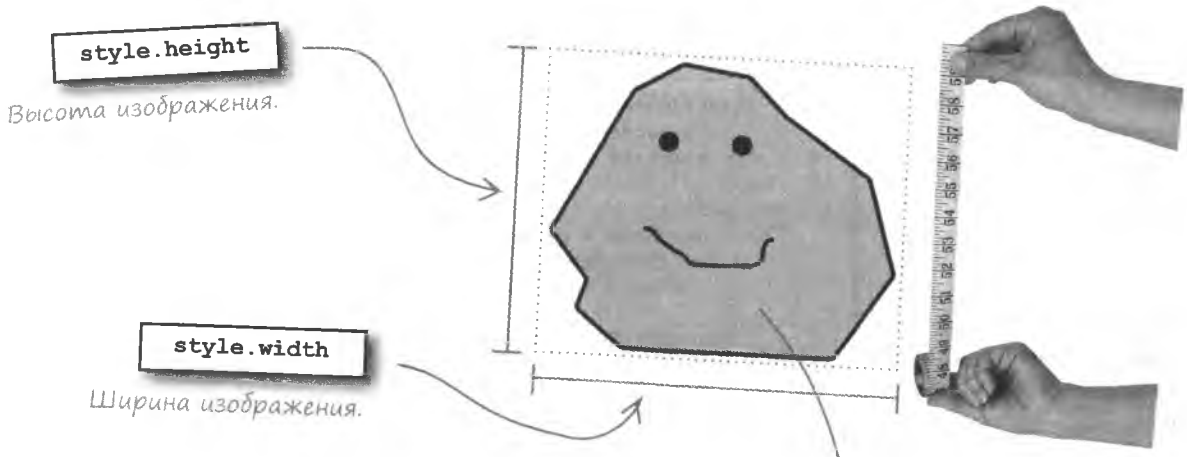
О: Да, запутаться тут легко. С глобальной точки зрения браузер является веб-клиентом, так как при обслуживании веб-страниц он находится на стороне клиента. Если же рассмотреть браузер, слово «клиент» уже будет относиться к области, в которой появляется веб-страница. Так что окном клиента называется область внутри браузера, в которую не входят строки заголовка, полосы прокрутки, панели инструментов и т. д.

В: Почему мы собираемся менять размер объекта iRock именно в зависимости от размера окна клиента?

О: Именно от размеров окна клиента зависит, какую часть пространства будет занимать отображаемый объект. Остальные элементы браузера, например дополнительные панели инструментов, крайне сложно учесть из-за их разнообразия при переходе от одной платформы к другой. К примеру, при одинаковом размере окна клиента размер браузера Safari в Mac будет отличаться от размера браузера Firefox в Windows.

Высота и ширина объекта iRock

Знание размеров клиентского окна не поможет вам без возможности менять размер изображения объекта iRock. К счастью, JavaScript вместе с CSS позволяют это делать. Свойства `width` и `height` изображения дают не только информацию о размере объекта, но и возможность его динамически менять.



Для каждого элемента страницы существует объект `style`, позволяющий узнать геометрические размеры этого элемента. Сначала вам потребуется доступ к самому элементу, то есть к картинке с камешком. Как вы уже знаете, для этого нужен метод `getElementById()` объекта `document`:

Код HTML картинки с камнем дает вам доступ к свойствам стиля изображения.

```
<img id="rockImg" src="rock.png" alt="iRock" ... />
```

```
document.getElementById("rockImg").style.height
```

Этот код позволяет узнать высоту изображения камешка.

Для изменения размеров объекта iRock достаточно поменять значения соответствующих свойств. Точнее, вы можете ограничиться редактированием только одного свойства. Второе изменится автоматически, чтобы сохранить пропорции изображения:

Сделайте высоту изображения равной 100 пикселам.

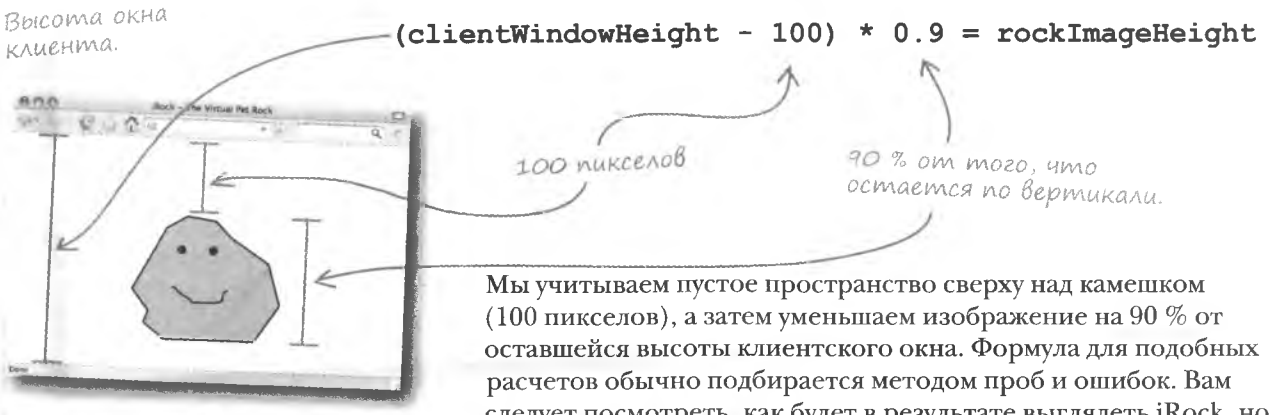
```
document.getElementById("rockImg").style.height = "100px";
```

Ширину в явном виде указывать уже не нужно... данный параметр изменится автоматически в соответствии с пропорциями изображения.

iRock должен соответствовать странице

Пока что размер изображения камешка никак не связан с размером окна клиента, в то время как он должен меняться пропорционально ему. Пусть его размер составляет определенный процент от размера окна.

Какое из измерений следует выбрать для привязки? Так как по вертикали браузеры имеют большие ограничения, имеет смысл выбирать размер камня в зависимости от высоты видимой области.



Мы учитываем пустое пространство сверху над камешком (100 пикселов), а затем уменьшаем изображение на 90 % от оставшейся высоты клиентского окна. Формула для подобных расчетов обычно подбирается методом проб и ошибок. Вам следует посмотреть, как будет в результате выглядеть iRock, но сначала напишем код.

Возьми в руку карандаш

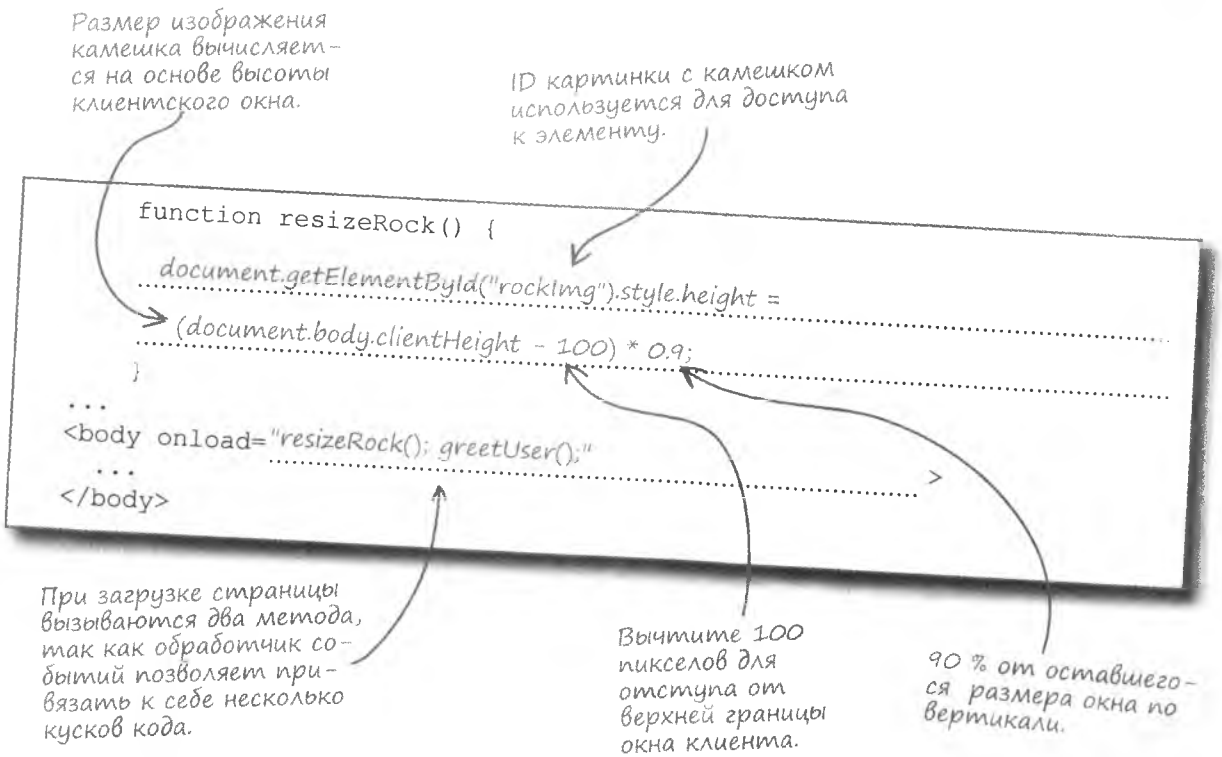
Напишите код для метода `resizeRock()`, устанавливающего размер камешка в зависимости от размеров клиентского окна. Также добавьте код к обработчику события onload, чтобы кроме метода `greetUser()` вызывался еще и метод `resizeRock()`.

```
function resizeRock() {
    .....
    .....
}
...
<body onload=
  ...                                    >
</body>
```

Возьми в руку карандаш

Решение

Вот как выглядит код метода `resizeRock()` и код обработчика события `onload`, вызывающего этот метод.

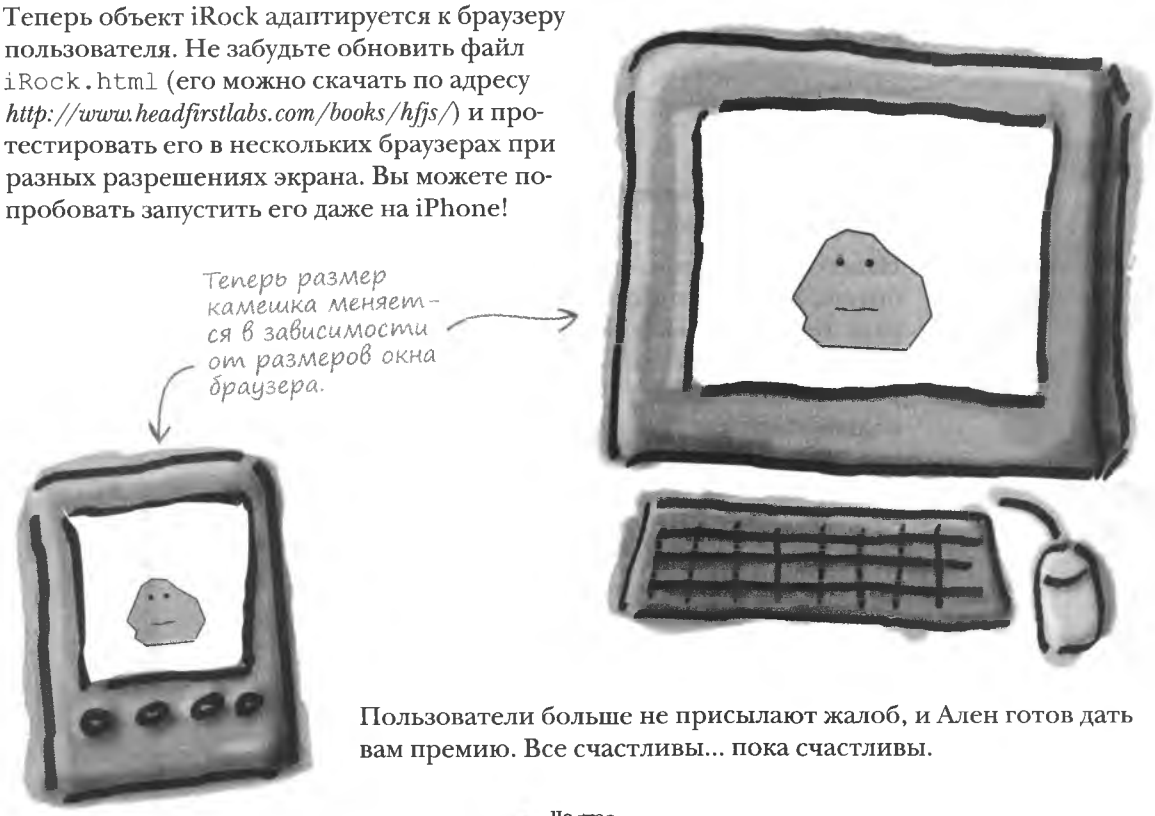


КЛЮЧЕВЫЕ МОМЕНТЫ

- Метод `setTimeout()` создает **однократный таймер**, который после завершения отсчета запускает код JavaScript.
- Таймер, срабатывающий через заданные промежутки времени, создается методом `setInterval()`.
- Время работы таймера указывается в миллисекундах, то есть в тысячных долях секунды.
- Объект `style` элементов веб-страницы задает их свойства, например, `width` и `height`.
- Окном клиента называется часть окна браузера, в котором отображается веб-страница.
- Информация о ширине и высоте окна клиента содержится в свойствах `body.clientWidth` и `body.clientHeight` объекта `document`.

iRock эволюционирует!

Теперь объект iRock адаптируется к браузеру пользователя. Не забудьте обновить файл `iRock.html` (его можно скачать по адресу *http://www.headfirstlabs.com/books/hfjs/*) и протестировать его в нескольких браузерах при разных разрешениях экрана. Вы можете попробовать запустить его даже на iPhone!



Пользователи больше не присылают жалоб, и Ален готов дать вам премию. Все счастливы... пока счастливы.

Часто Задаваемые Вопросы

В: Я так и не понял, зачем нужно было вычитать 100 пикселов.

О: Код HTML/CSS для объекта iRock помещает его изображение на 100 пикселов ниже верхней границы страницы, чтобы он не прилипал к верху окна. В вычислениях это смещение на 100 пикселов учитывается до момента нахождения (90 %) от высоты клиентского окна. Подобный отступ нужен только для того, чтобы поместить камешек на более удачное место в большинстве браузеров.

В: Свойства `width` и `height` CSS-стиля позволяют менять ширину и высоту любого объекта?

О: Практически так. Теперь вы можете представить, насколько мощным инструментом управления содержимым веб-страницы является JavaScript. В рассмотренном случае с его помощью вы смогли узнать размеры окна и на основе этих данных рассчитать размер изображения объекта.

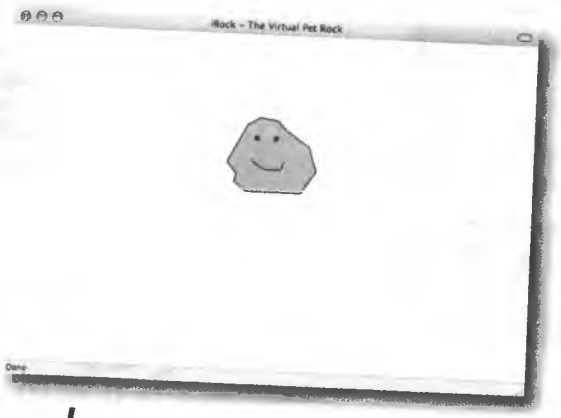
В: Почему не поменять размеры объекта iRock в коде JavaScript в заголовке страницы, не прибегая к событию `onload`?

О: Дело в том, что содержимое страницы не загружается до появления события `onload`. Поэтому если ваш код JavaScript имеет доступ к элементам страницы, как в случае с кодом объекта iRock, вы не сможете его запустить, пока не произойдет событие `onload`.

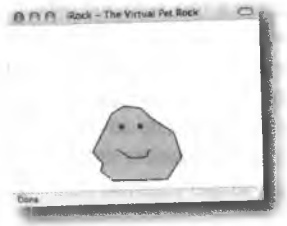
А что происходит при изменении размеров окна браузера? Изображение сохраняет свой размер?

Нет, размер камешка не является динамическим.

Некоторые пользователи могут поменять размер окна браузера, а объект iRock при этом останется тем же самым. Ведь его размер определяется при первой загрузке, в момент события `onload`. После этого на вид объекта уже ничего не влияет. То есть мы вернулись к нашей исходной проблеме:

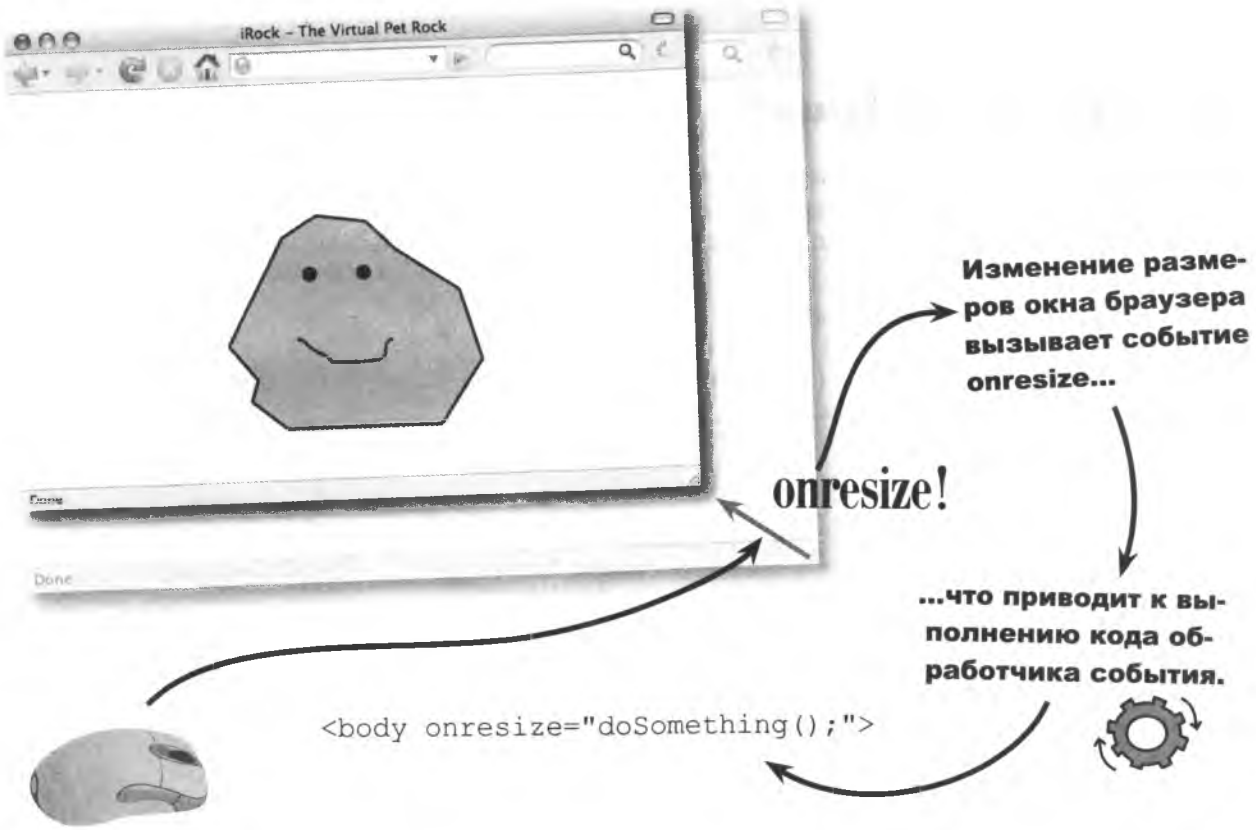


Камешек сохраняет свой размер при изменении размеров окна браузера.



Событие onresize

Для того чтобы изображение камешка могло менять свой размер при изменении размеров браузера, сценарию нужно дать понять, что произошло такое изменение. Для этой цели служит событие `onresize`.



Чтобы ответить на событие `onresize`, используйте код JavaScript для атрибута `onresize` тега `<body>`.

Возьми в руку карандаш

Одно из этих событий отличается от двух других. Какое именно и почему?

onload **onresize** **onclick**

.....

Возьми в руку карандаш

Решение

Какое из этих событий отличается от двух других и почему?

onload **onresize** **onclick**

События onresize и onclick инициируются пользователем, а onload — нет.

Событие onresize для камешка

Пришло время создать метод, меняющий размер изображения камешка. Чтобы это происходило в ответ на изменение размеров браузера, метод `resizeRock()` должен вызываться событием `onresize`:

Возникает при первой загрузке страницы.

Возникает при изменении размеров окна браузера.

```
<body onload="resizeRock(); greetUser();" onresize="resizeRock();">
```

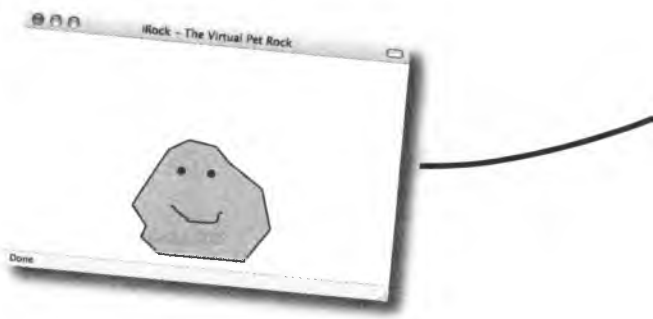
Метод resizeRock() вызывается при первой загрузке страницы, задавая начальные размеры нашего изображения.

Ответом на событие может стать выполнение нескольких фрагментов кода.

Теперь метод resizeRock() вызывается еще и при любом изменении размеров окна браузера.

Теперь размер изображения объекта iRock автоматически меняется при изменении пользователем размеров окна браузера.

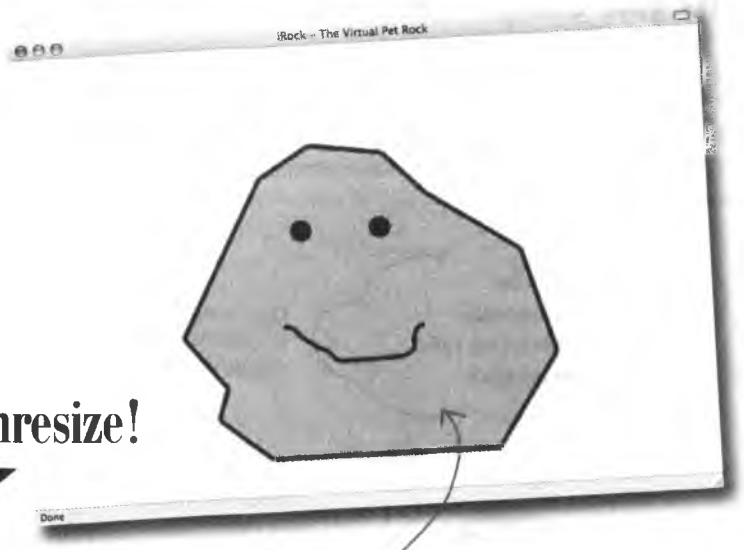
Событие onresize позволяет распознать изменение размеров окна браузера и отреагировать на них.



Будьте осторожны!

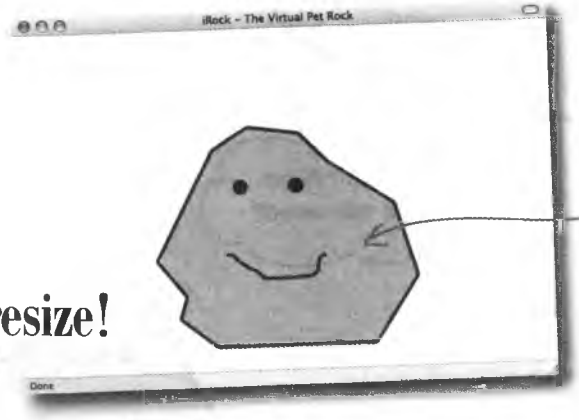
Будьте аккуратны, меняя размер изображения при помощи JavaScript.

Особенно если вы увеличиваете маленькую картинку. Ведь при этом ее качество может ухудшиться.



onresize!

JavaScript засекает изменение клиента и динамически меняет содержимое веб-страницы в соответствии с этим изменением.



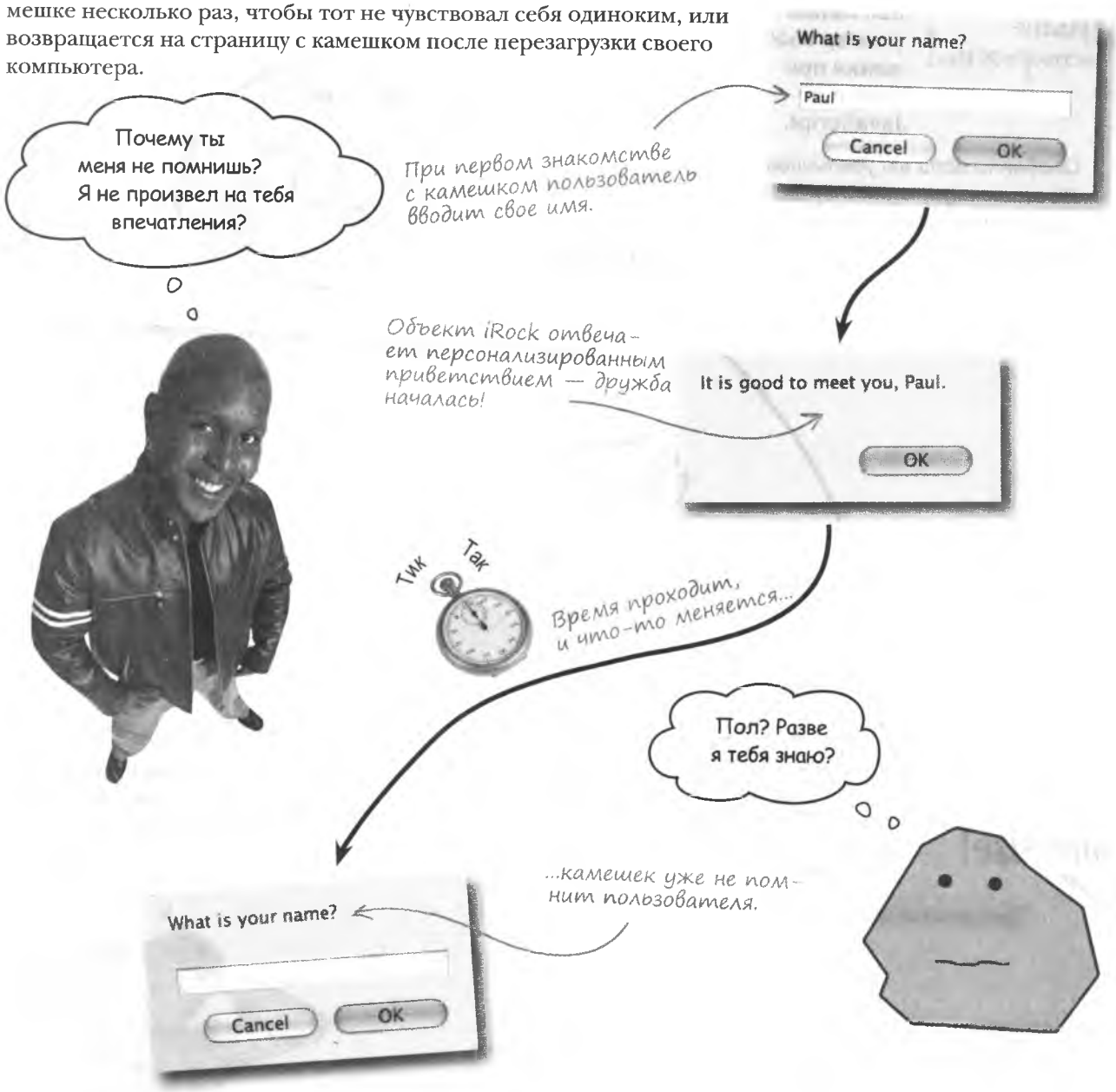
onresize!

Пользователи будут в восторге от этого. А какие еще будут идеи?

Ален чувствует любовь со стороны пользователей, ведь теперь объект iRock невосприимчив к изменениям размеров браузеров. iRock не только подстраивается под размер клиентского окна на первом этапе, но и динамически реагирует на любые его изменения.

Мы уже встречались?

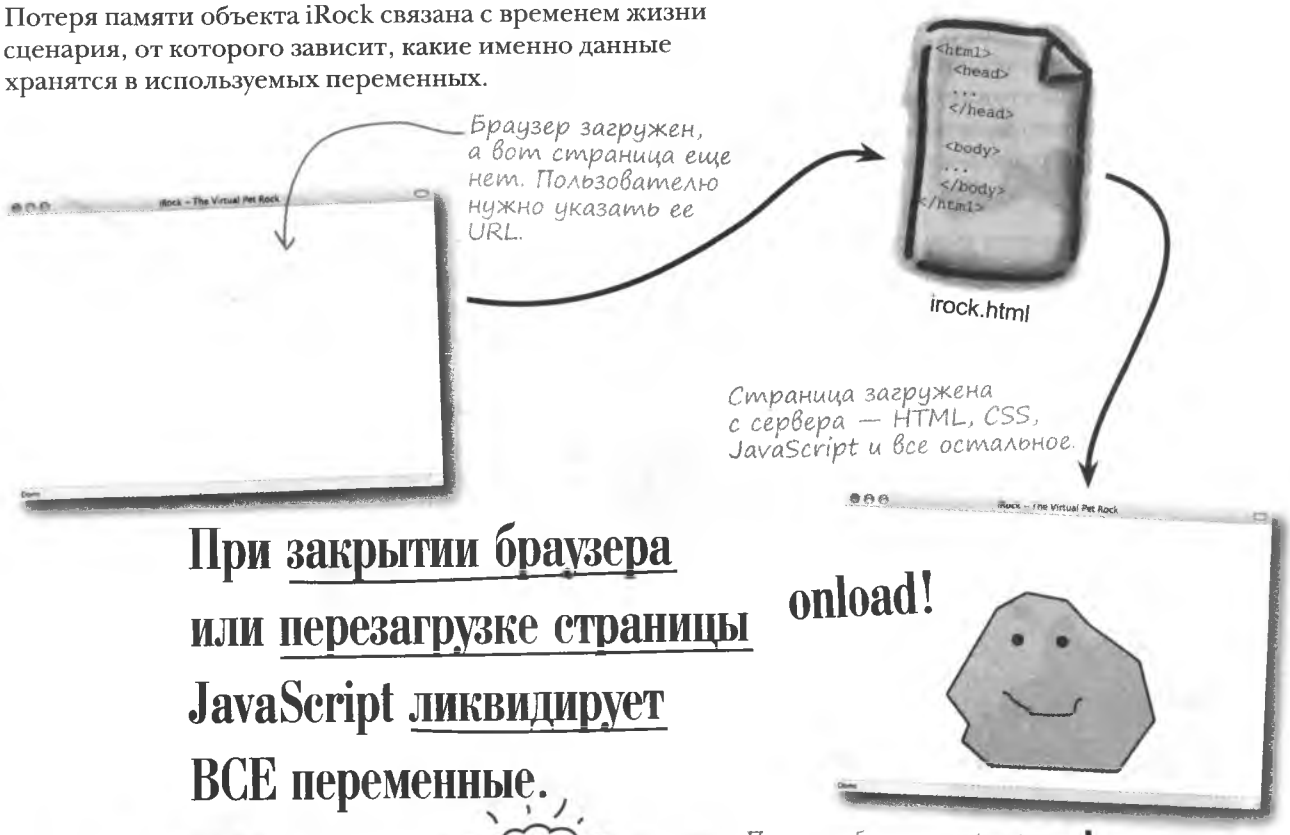
Проблемы с размером объекта iRock отошли в прошлое... теперь решим, как быть в ситуациях, когда пользователь щелкает на камешке несколько раз, чтобы тот не чувствовал себя одиноким, или возвращается на страницу с камешком после перезагрузки своего компьютера.



Хотя iRock и встречался со своим хозяином некоторое время назад, он забыл его имя...

Время жизни сценария

Потеря памяти объекта iRock связана с временем жизни сценария, от которого зависит, какие именно данные хранятся в используемых переменных.



При закрытии браузера или перезагрузке страницы JavaScript ликвидирует ВСЕ переменные.

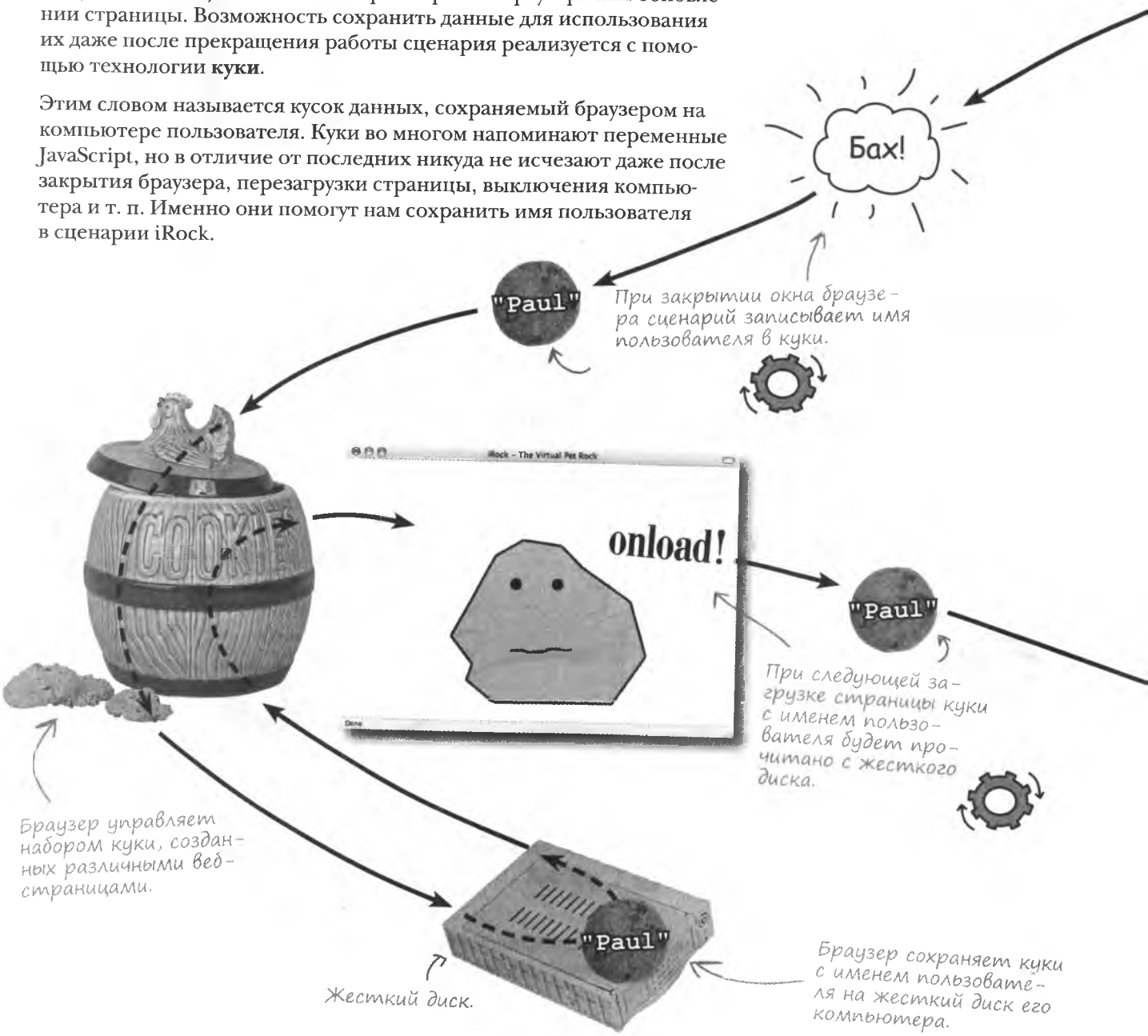
МОЗГОВОЙ ШТУРМ

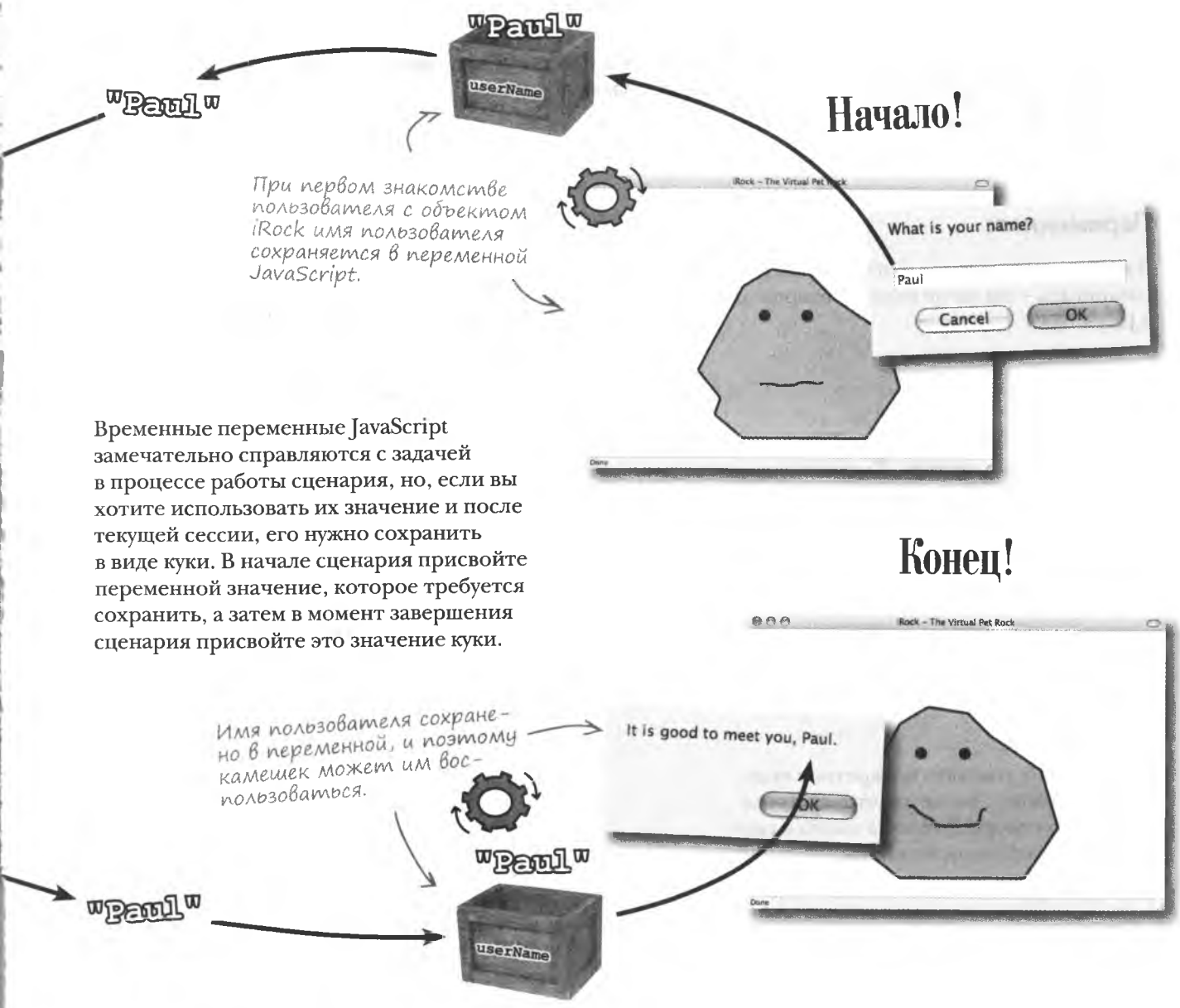
Как бы вы решили проблему с объектом iRock, забывающим имя пользователя?

Продление времени жизни сценария

Проблема, которую мы имеем с объектом iRock, называется **постоянство**. Точнее, его отсутствие. Иногда нам требуются данные, которые никуда не исчезают. К сожалению, переменные в JavaScript живут недолго и уничтожаются при закрытии браузера или обновлении страницы. Возможность сохранить данные для использования их даже после прекращения работы сценария реализуется с помощью технологии **куки**.

Этим словом называется кусок данных, сохраняемый браузером на компьютере пользователя. Куки во многом напоминают переменные JavaScript, но в отличие от последних никуда не исчезают даже после закрытия браузера, перезагрузки страницы, выключения компьютера и т. п. Именно они помогут нам сохранить имя пользователя в сценарии iRock.





Временные переменные JavaScript замечательно справляются с задачей в процессе работы сценария, но, если вы хотите использовать их значение и после текущей сессии, его нужно сохранить в виде куки. В начале сценария присвойте переменной значение, которое требуется сохранить, а затем в момент завершения сценария присвойте это значение куки.

Возьми в руку карандаш

Как вы думаете, какие еще данные с веб-страницы будет полезно сохранить при помощи куки.

.....

Беседа у камина

Переменная и куки обсуждают важность длительного хранения данных.

Переменная:

Я вообще не понимаю, о чем с тобой разговаривать — ты же не имеешь отношения к JavaScript.

Я вижу, куда ты клонишь. Ты считаешь меня менее подходящей для хранения данных, так как они удаляются при каждом закрытии браузера или перезагрузке страницы. Но я при этом очень доступна, в отличие от некоторых...

Допустим. Но разве ты не живешь в тесном соседстве с множеством других куки?

Говорят, что отыскать конкретное куки крайне сложно... вы же хранитесь в виде огромного списка. Вот что я имела в виду, говоря о твоей недоступности.

Куки:

Ты почти права. Я выполняю свою работу без помощи JavaScript, но при этом я помогаю сохранять данные сценариев надолго. А как ты, вероятно, знаешь, JavaScript не позволяет этого делать.

Я недоступно? Я всегда под рукой, в браузере, и могу быть вызвано в любой момент.

Да... и что?

Ну да, мы хранимся в виде списка, но все куки имеют уникальные имена, поэтому отыскать нужное не так уж сложно. Достаточно понимать, как разбить список на части и найти в нем конкретное имя.

Возьми в руку карандаш

Решение

Вот какие еще данные веб-страниц имеет смысл сохранять при помощи куки.

ID пользователя, содержимое корзины покупок, место жительства, язык.

Переменная:

А я вот не участвую ни в каких списках. Достаточно назвать мое имя... я и тут!

При всех своих достоинствах постоянство не решает каждодневных проблем. Если подумать, то далеко не все данные нужно сохранять надолго. Более эффективно хранить их некоторое время, а потом, завершив работу, удалять. Вот тут на сцену выхожу я – временная среда хранения для данных сценария.

Вот только вспомни, каким образом все эти товары изначально оказались в корзине? Ведь именно я храню временную информацию о предлагаемых товарах. Я важна так же, как и ты... а может быть даже больше. Даже несмотря на стремление быстро забывать данные, с которыми работаю.

Думаю, ты право. Мы решаем разные задачи и делить нам нечего. Хотя должна заметить, что я предпочитаю легкий доступ ко мне возможности перманентного хранения данных.

Какой разговор?

Куки:

Я поняло вашу точку зрения. Зато, когда во мне сохраняют информацию, я ее действительно запоминаю. Не важно, был ли закрыт браузер или обновлена страница. Я вечно... до тех пор, пока пользователь не решит почистить куки. Но это уже другой вопрос.

Я все равно убеждено, что ты недооцениваешь важность долговременного хранения данных. Разве не восхитительно, когда после долгого времени пользователь заходит в интернет-магазин и видит, что все выбранные им ранее товары все еще находятся в его корзине? И все это благодаря мне.

Мне начинает казаться, что мы дополняем друг друга. А ведь я всегда считала тебя заклятым врагом.

А я испытываю удовольствие от мысли, что, как только страница будет перевернута, вы забудете весь наш разговор.

Ну, что я говорило?

Для сохранения небольших фрагментов информации, например имени пользователя, сервер не нужен.

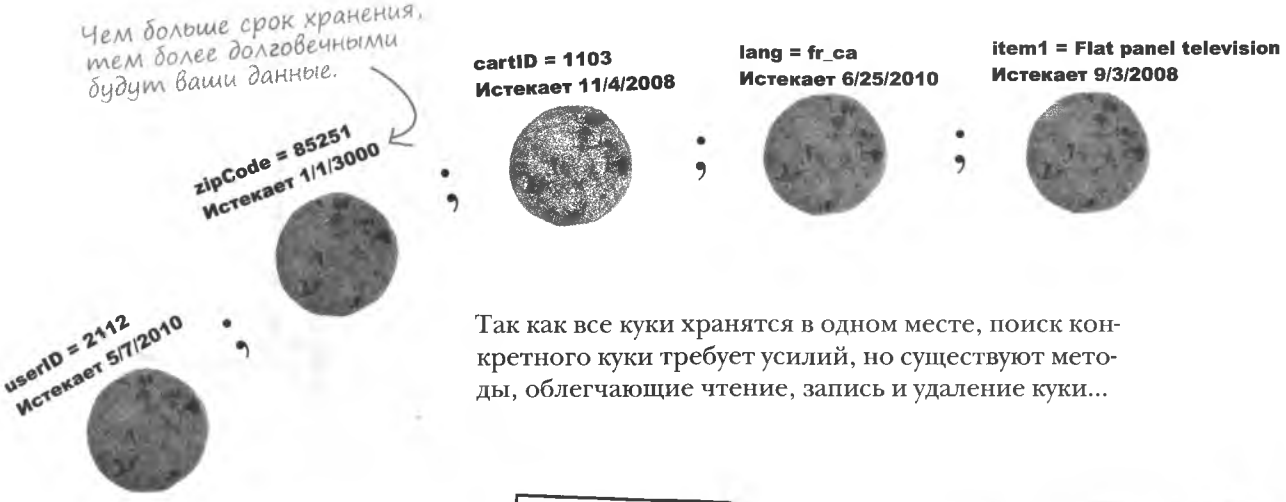
Разумеется, сервер может использоваться как долговременное хранилище данных, но скоро он окажется переполнен маленькими фрагментами информации. Сохранение данных на сервере требует работы программистов и специальной среды, например базы данных. Вам не кажется, что это слишком большая работа для сохранения, например, имени пользователя в сценарии iRock?

Куки сохраняют данные на стороне клиента, никак не затрагивая сервер. При этом пользователь всегда имеет возможность удалить куки со своего компьютера, если он решил, что эта информация ему больше не требуется. Такое простое удаление данных с сервера невозможно.

Свойства куки

Куки сохраняют маленькие фрагменты данных под уникальными именами, почти как переменные. Но, в отличие от переменных, у куки есть срок хранения. После его достижения куки уничтожаются. Так что куки не являются вечными, они просто живут намного **дольше** переменных. Можно создать куки, не имеющие срока хранения, но в этом случае они будут, как и переменные JavaScript, стираться после закрытия браузера.

Куки сохраняются на компьютере пользователя в виде длинной строки текста, связанной с сайтом (или доменом). Друг от друга они отделяются точкой с запятой (;). Именно этот разделитель дает возможность найти в списке конкретный куки.



Так как все куки хранятся в одном месте, поиск конкретного куки требует усилий, но существуют методы, облегчающие чтение, запись и удаление куки...

```
readCookie()
writeCookie()
eraseCookie()
```

Если вы не понимаете смысл написанного, ничего страшного. По мере чтения книги все постепенно станет ясно.

Вот код трех вспомогательных методов, предназначенных для чтения, записи и удаления куки. Иногда правильнее — заставить работать других. Поэтому возьмите этот рецепт (его можно скачать по адресу *http://www.headfirstlabs.com/books/hfjs/*) и используйте его для работы с куки.

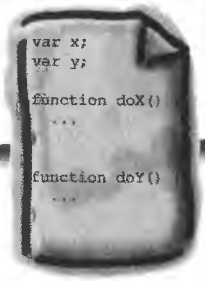
```
function writeCookie(name, value, days) {
  // По умолчанию куки являются временными, не имея срока хранения
  var expires = "";

  // Указав число дней, вы сделаете куки постоянными
  if (days) {
    var date = new Date();
    date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
    expires = "; expires=" + date.toGMTString();
  }

  // Присвоим куки имя, значение и срок хранения
  document.cookie = name + "=" + value + expires + "; path=/";
}

function readCookie(name) {
  // Найдем конкретный куки и вернем его значение
  var searchName = name + "=";
  var cookies = document.cookie.split(';');
  for(var i=0; i < cookies.length; i++) {
    var c = cookies[i];
    while (c.charAt(0) == ' ')
      c = c.substring(1, c.length);
    if (c.indexOf(searchName) == 0)
      return c.substring(searchName.length, c.length);
  }
  return null;
}

function eraseCookie(name) {
  // Удалим выбранный куки
  writeCookie(name, "", -1);
}
```

Срок хранения записывается в виде числа дней, которые должны существовать куки.

Срок хранения вычисляется преобразованием числа дней в миллисекунды и добавлением полученного числа к текущему времени.

Разделитель точка с запятой делит список куки на отдельные экземпляры.

Для удаления куки достаточно записать его с отсутствующим значением и истекшим сроком хранения (-1 день).



cookie.js

Создайте пустой файл с именем cookie.js и добавьте в него этот код.

Разрешение .js указывает, что файл содержит только код JavaScript.

Код JavaScript ВНЕ Веб-страницы

Сохраненный в отдельный файл код JavaScript требуется импортировать на страницу, для которой он предназначен. Соответственно, файл `cookie.js` нужно **импортировать** на страницу *iRock.html*. Это реализуется в теге `<script>`:

Не забудьте закрыть его тегом </script>.

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

Для JavaScript значение атрибута type всегда равно text/javascript.

Имя файла с кодом сценария обычно заканчивается на .js.

Для импорта на страницу кода JavaScript из файла используйте тег `<script>`. Многократно используемые сценарии всегда имеет смысл помещать в отдельный файл и потом просто импортировать в веб-страницу.

СДЕЛАЙТЕ ЭТО! Добавьте строчку в файл iRock.html и убедитесь, что файл cookie.js находится в той же папке.

```
<html>
  <head>
    <title>iRock - The Virtual Pet Rock</title>
    <script type="text/javascript" src="cookie.js"></script>
    <script type="text/javascript">
      var userName;

      function resizeRock() {
        document.getElementById("rockImg").style.height =
          (document.body.clientHeight - 100) * 0.9;
      }

      function greetUser() {
```

Импортированный код сценария помещается на страницу при ее загрузке.

Упражнение

Как вы считаете, почему многократно используемый код имеет смысл помещать в отдельный файл?

.....

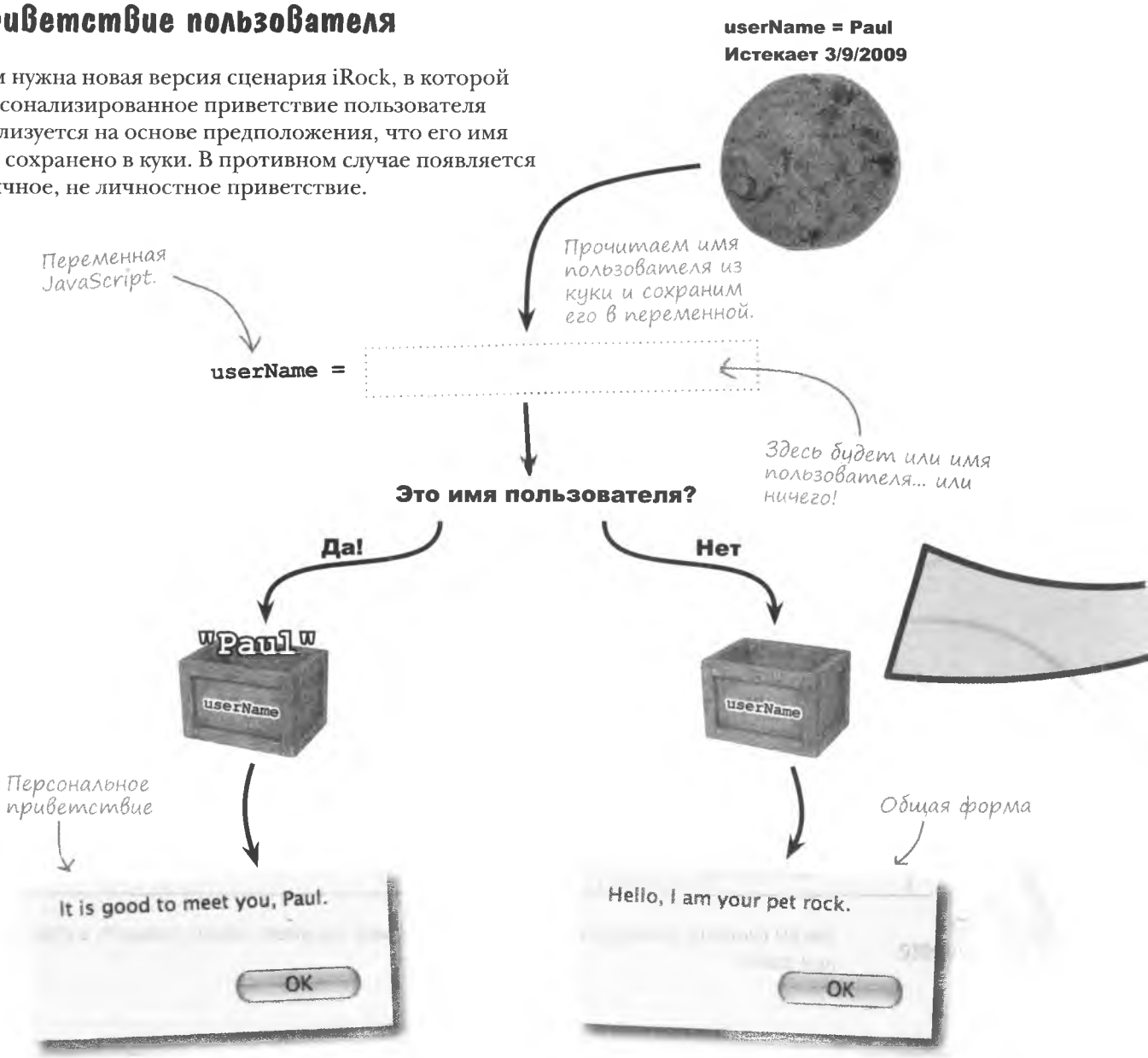
Упражнение
Решение

Почему код многократного использования имеет смысл поместить в отдельный файл?

Так как при необходимости его можно будет легко отредактировать.

Приветствие пользователя

Нам нужна новая версия сценария iRock, в которой персонализированное приветствие пользователя реализуется на основе предположения, что его имя уже сохранено в куки. В противном случае появляется обычное, не личностное приветствие.



Подробно про код

Метод greetUser() отвечает за приветствие пользователя при первой загрузке страницы.

Куки, хранящее информацию об имени пользователя, должно иметь значимое имя. Это избавит вас от путаницы, если в будущем вы решите добавить другие куки в данный сценарий.

```
function greetUser() {
  userName = readCookie("irock_username");
  if (userName)
    alert("Hello " + userName + ", I missed you.");
  else
    alert('Hello, I am your pet rock.');
}
```

Это не сложение... это соединение строк.

Имя пользователя читается из куки и хранится в переменной userName.

Имя пользователя отсутствует, потому что не существует нужного куки. Значит, покажем приветствие в общей форме.

При наличии куки с именем пользователя покажем персонализированное приветствие.

Метод greetUser() на основе куки

Теперь в методе `greetUser()` работают дуэтом и переменная, и куки. Имя пользователя читается из куки и сохраняется в переменной. Но на тот факт, что имя сохранено в куки, полагаться нельзя, в конце концов, сценарий может запускаться в первый раз, значит, пользователь еще не указывал своих личных данных. Именно поэтому в коде проверяется возможность для переменной получить имя из куки, и на основе этой проверки выбирается нужный тип приветствия.

Создание куки

Использовать куки для нашего объекта iRock — отличная идея, но сначала их требуется создать. Запись куки осуществим в методе `touchRock()`, который вызывается при щелчке пользователя на камешке. Как вы помните, этот метод предлагает пользователю ввести его имя — теперь он будет еще и записывать куки после ввода этой информации.

Подробно про код

```
function touchRock() {
  if (userName) {
    alert("I like the attention, " + userName + ". Thank you.");
  }
  else {
    userName = prompt("What is your name?", "Enter your name here.");
    if (userName) {
      alert("It is good to meet you, " + userName + ".");
      writeCookie("irock_username", userName, 5 * 365);
    }
  }
  document.getElementById("rockImg").src = "rock_happy.png";
  setTimeout("document.getElementById('rockImg').src = 'rock.png';",
    5 * 60 * 1000);
}
```

Сначала проверим, указал ли пользователь свое имя.

Если имя указано, поблагодарим пользователя за внимание.

Если имя не указано, его нужно узнать.

Убедитесь, что имя введено.

Имя имеется, поэтому поприветствуйте пользователя и запишите его данные в куки.

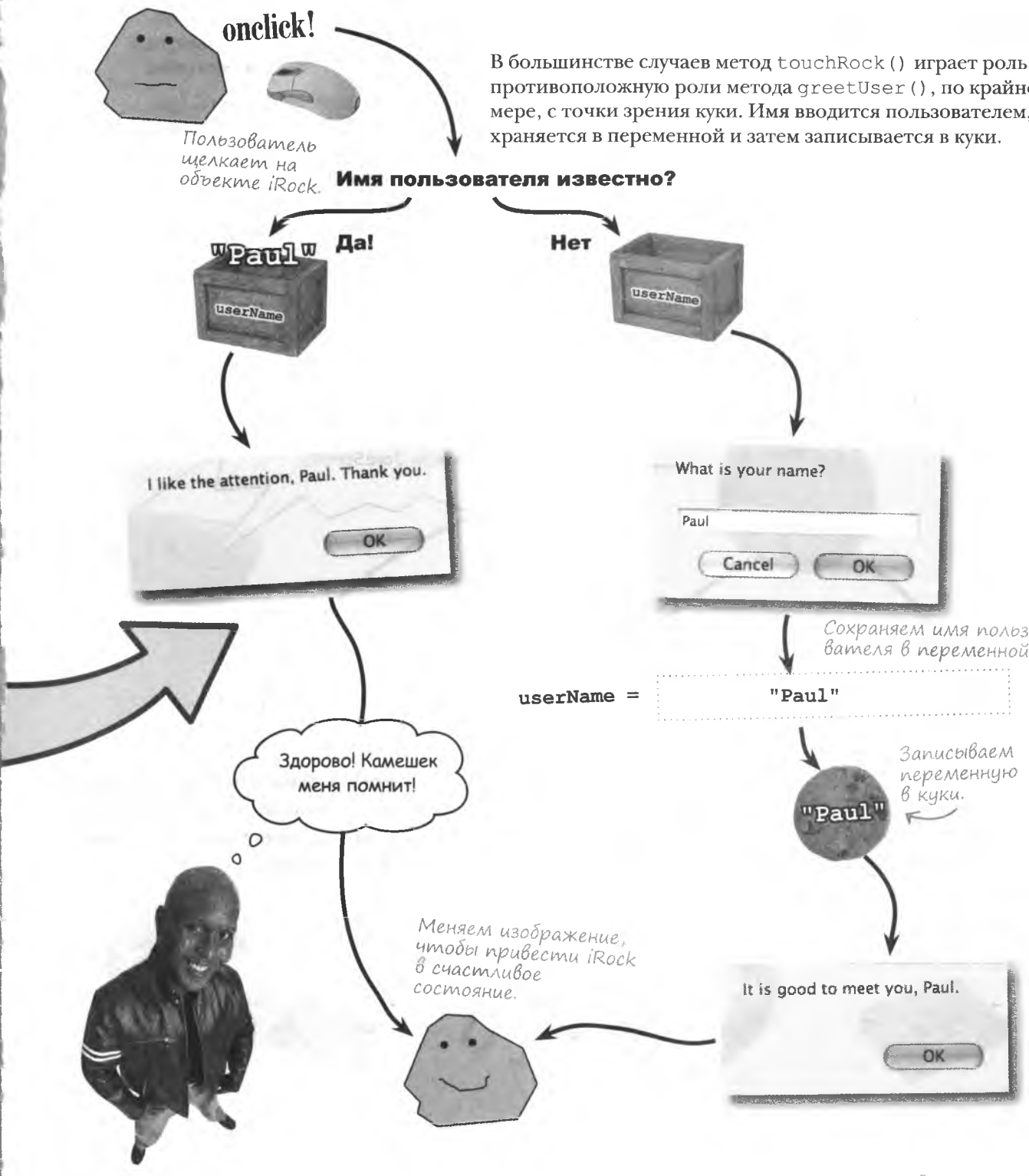
Именно это имя используется при чтении куки. Куки не используют стильВерблюда, они больше напоминают идентификаторы HTML.

Имя пользователя записывается из переменной.

Сохраним куки с именем пользователя на 5 лет.

Этот метод вызывается при щелчке на камешке.

```
<img id="rockImg" src="rock.png" onclick="touchRock();" />
```



В большинстве случаев метод `touchRock()` играет роль, противоположную роли метода `greetUser()`, по крайней мере, с точки зрения куки. Имя вводится пользователем, сохраняется в переменной и затем записывается в куки.

Влияние на безопасность

Большинство пользователей объекта iRock в восторге от такого лекарства против склероза, как куки, но некоторые из них озабочены вопросом безопасности. Это правильный вопрос, так как в куки часто сохраняют персональные данные, хотя и **не ставят** под угрозу безопасность... по меньшей мере в плане доступа к хранящимся на вашем компьютере уязвимым данным. Но сами по себе куки не являются безопасным способом хранения информации, поэтому сохранять с их помощью **уязвимые** данные не стоит.

То, что вы можете, не означает, что вы *должны*.

Хотя в куки можно сохранить что угодно, они являются не слишком безопасным способом хранения данных. Поэтому уязвимые данные, например пароль пользователя, таким способом сохранять не стоит.

Куки — это всего лишь фрагменты текстовой информации, сохраняемой к вам на компьютер.

Хотя куки и хранятся на жестком диске, они не имеют доступа к остальной хранящейся там информации.

Не являясь исполняемыми программами, куки не могут стать источником вирусов или червей.

Куки могут сохранять персональные данные, но только после того, как пользователь сознательно указал их на веб-странице.

Часто Задаваемые Вопросы

В: То есть куки всегда сохраняются на жесткий диск компьютера?

О: Нет. Просто большинство браузеров сохраняет куки именно на жесткий диск. Но при этом есть и браузеры, которые не имеют туда доступа. К примеру, некоторые мобильные устройства используют для этого специальную память. При этом с точки зрения браузера (и сценария) куки помнят присвоенные им значения вне зависимости от того, куда они были сохранены.

В: Как понять, что куки имеют уникальные имена?

О: Уникальность имени нужна для куки только в пределах рассматриваемой страницы, ведь они сохраняются со ссылкой на нее. Таким образом, эффективной частью имени куки является название страницы, что гарантирует его уникальность.

В: Куки сохраняются для всех браузеров?

О: Нет. Каждый браузер имеет свою, уникальную базу куки. То есть куки, созданные в Internet Explorer, не будут видимы в Firefox или Opera.

В: Имея куки, зачем вообще сохранять данные на сервер?

О: Ну, во-первых, куки позволяют сохранять только относительно небольшие (менее 4 Кбайт) фрагменты текста. Это одно из их самых больших ограничений. Кроме того, куки недостаточно эффективны, и вряд ли вы захотите постоянно записывать и читать оттуда данные. Для подобных целей обычно используются базы данных, которые находятся именно на серверах. Несмотря на то что куки прекрасно подходят для сохранения небольших фрагментов информации, которые необязательно помещать в базу на сервере, они не помогут вам в случаях с другими видами данных. Также их не стоит применять для сохранения уязвимых данных, так как это небезопасно.

В: Можно ли создавать действительно вечные куки?

О: Нет. Нравится вам это или нет, все куки имеют срок действия. Они предназначены для достаточно длительного хранения данных, но никто не предполагал использовать их в качестве вечного хранилища. Куки хранят данные дни, недели, месяцы. Для более длительного хранения данные стоит поместить на сервер. Нет, куки тоже потенциально могут хранить данные годами, но пользователи обновляют компьютеры, переустанавливают браузеры и т. п.

В: Какие недостатки есть у хранения кода JavaScript во внешнем файле?

О: Недостатков нет. Просто следует помнить, что целью сохранения сценария во внешний файл является доступ к нему из нескольких источников. Если ваш сценарий появляется всего на одной странице, нет никакого смысла сохранять его во внешний файл. Ну разве что только в случаях, когда код выглядит запутанным и вы хотите упростить его, распределив сценарии по отдельным файлам.

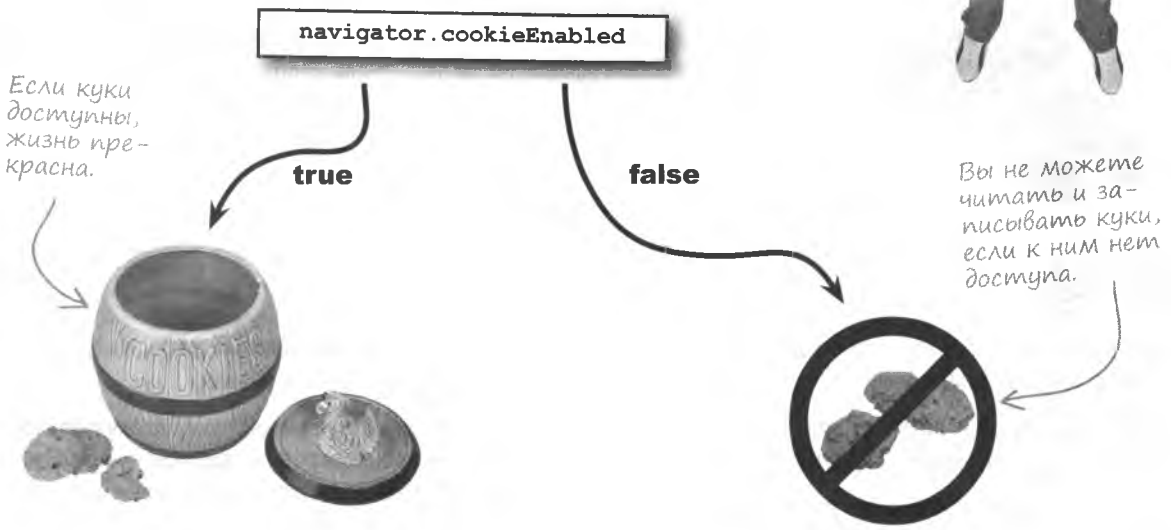
КЛЮЧЕВЫЕ МОМЕНТЫ

- **Куки** — это фрагменты **текстовых данных**, которые браузер сохраняет на компьютер пользователя.
- Куки позволяют сценариям **сохранять данные**, которые могут потребоваться **в следующей сессии**.
- Все куки имеют **срок хранения**, после истечения которого они уничтожаются браузером.
- Поместив сценарий во **внешний файл**, вы сделаете его доступным для других страниц.
- Куки **не имеют доступа к жесткому диску пользователя** и не **распространяют вирусы**, они просто сохраняют введенные пользователем на страницу персональные данные.

Мир без куки

По причинам безопасности или из-за ограничений браузеров некоторые пользователи объекта iRock не могут с ним работать, так как у них запрещены куки. Это большая проблема, так как iRock был создан в предположении, что куки поддерживаются везде. По крайней мере, нужно дать понять пользователям, что они не имеют доступа ко всем функциям объекта iRock.

К счастью, браузеры могут проверять доступность куки. Свойство `cookieEnabled` принадлежит объекту `navigator`, предоставляющему JavaScript информацию о браузере.



Возьми в руку карандаш

Напишите недостающий код для проверки доступности куки в методах `greetUser()` и `touchRock()`. В метод `touchRock()` также добавьте код, сообщающий пользователю о недоступности куки.

```
function greetUser() {

  .....
    userName = readCookie("irock_username");
  if (userName)
    alert("Hello " + userName + ", I missed you.");
  else
    alert('Hello, I am your pet rock.');
}

function touchRock() {
  if (userName) {
    alert("I like the attention, " + userName + ". Thank you.");
  }
  else {
    userName = prompt("What is your name?", "Enter your name here.");
    if (userName) {
      alert("It is good to meet you, " + userName + ".");

      .....
        writeCookie("irock_username", userName, 5 * 365);
      else

      .....
    }
  }
  document.getElementById("rockImg").src = "rock_happy.png";
  setTimeout("document.getElementById('rockImg').src = 'rock.png';",
    5 * 60 * 1000);
}
```

Возьми в руку карандаш

Решение

Вот как должен выглядеть код, проверяющий поддержку куки для методов `greetUser()` и `touchRock()`, а также код для второго метода, дающий пользователю понять, что куки недоступны.

Если куки поддерживаются, прочитайте имя пользователя из куки iRock.

```
function greetUser() {

  if (navigator.cookieEnabled) {
    userName = readCookie("irock_username");
  if (userName)
    alert("Hello " + userName + ", I missed you.");
  else
    alert('Hello, I am your pet rock.');
}

function touchRock() {
  if (userName) {
    alert("I like the attention, " + userName + ". Thank you.");
  }
  else {
    userName = prompt("What is your name?", "Enter your name here.");
    if (userName) {
      alert("It is good to meet you, " + userName + ".");

      if (navigator.cookieEnabled) {
        writeCookie("irock_username", userName, 5 * 365);
      else

        alert("Sorry. Cookies aren't supported/enabled in your browser. I won't
        remember you later.");
    }
  }
  document.getElementById("rockImg").src = "rock_happy.png";
  setTimeout("document.getElementById('rockImg').src = 'rock.png';",
    5 * 60 * 1000);
}
```

Если куки поддерживаются, запишите куки с именем пользователя.

Дайте пользователю знать, что отсутствие куки ограничит функциональность iRock.

Отредактируйте методы на странице iRock.html, как указано выше, и протестируйте полученный результат.

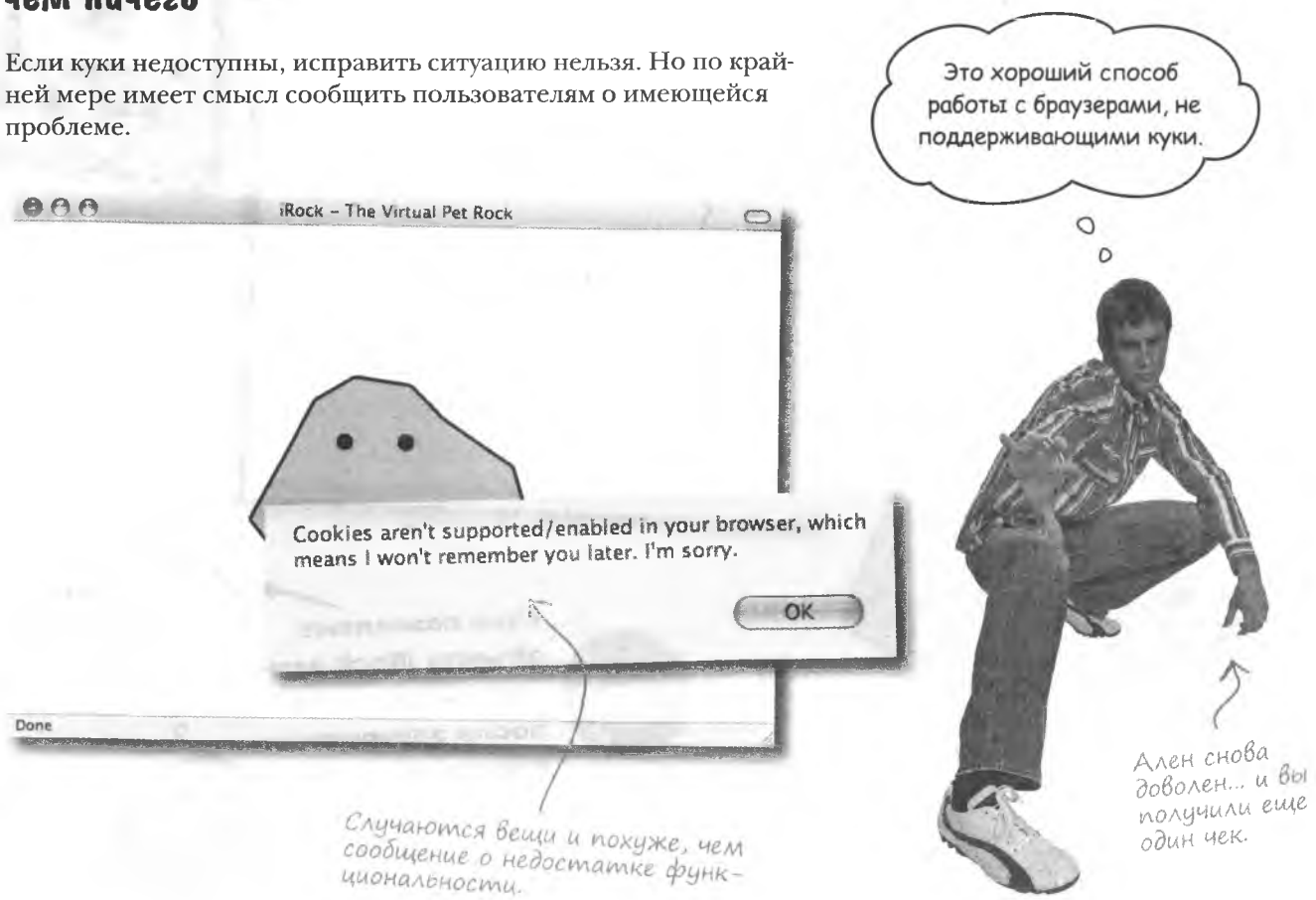
Часто Задаваемые Вопросы

В: Зависит ли поддержка куки от типа или версии браузера?

О: Распознавание типа и версии браузера в данном случае ведет к непредсказуемым результатам. К сожалению, верить тому, что сообщают о себе браузеры, нельзя. Поэтому, чтобы проверить, поддерживаются ли куки, используйте свойство `navigator.cookieEnabled`.

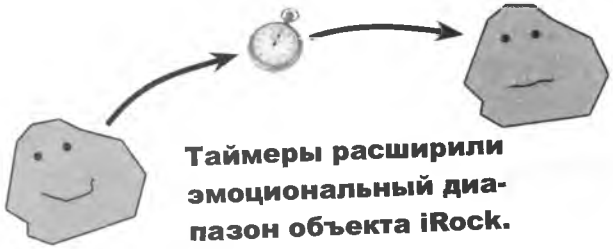
Разговор с пользователями... это лучше, чем ничего

Если куки недоступны, исправить ситуацию нельзя. Но по крайней мере имеет смысл сообщить пользователям о имеющейся проблеме.



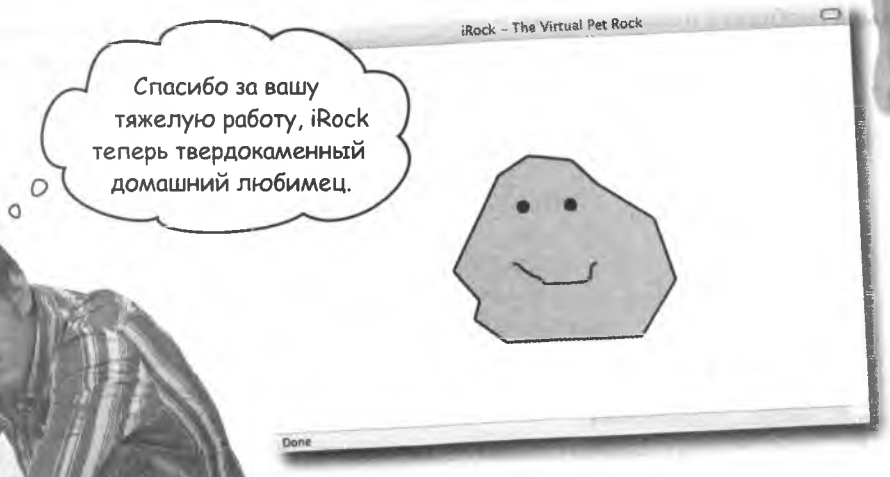
iRock — король JavaScript

Вы приложили множество усилий, совершенствуя код сценария и внося в него все необходимые для успеха объекта iRock изменения. Благодаря отзывам пользователей iRock приобрел бóльшую эмоциональность, возможность менять свой размер и даже улучшил память!



Таймеры расширили эмоциональный диапазон объекта iRock.

Параметры браузера и свойства CSS дают объекту iRock возможность подстраиваться под среду.



Спасибо за вашу тяжелую работу, iRock теперь твердокаменный домашний любимец.

```
userName = "Paul"
```

Куки позволяют объекту iRock помнить данные даже после завершения сценария.

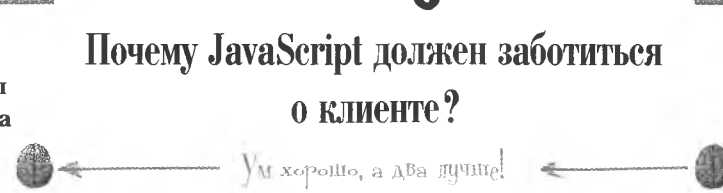
Рад встрече с тобой, Пол.

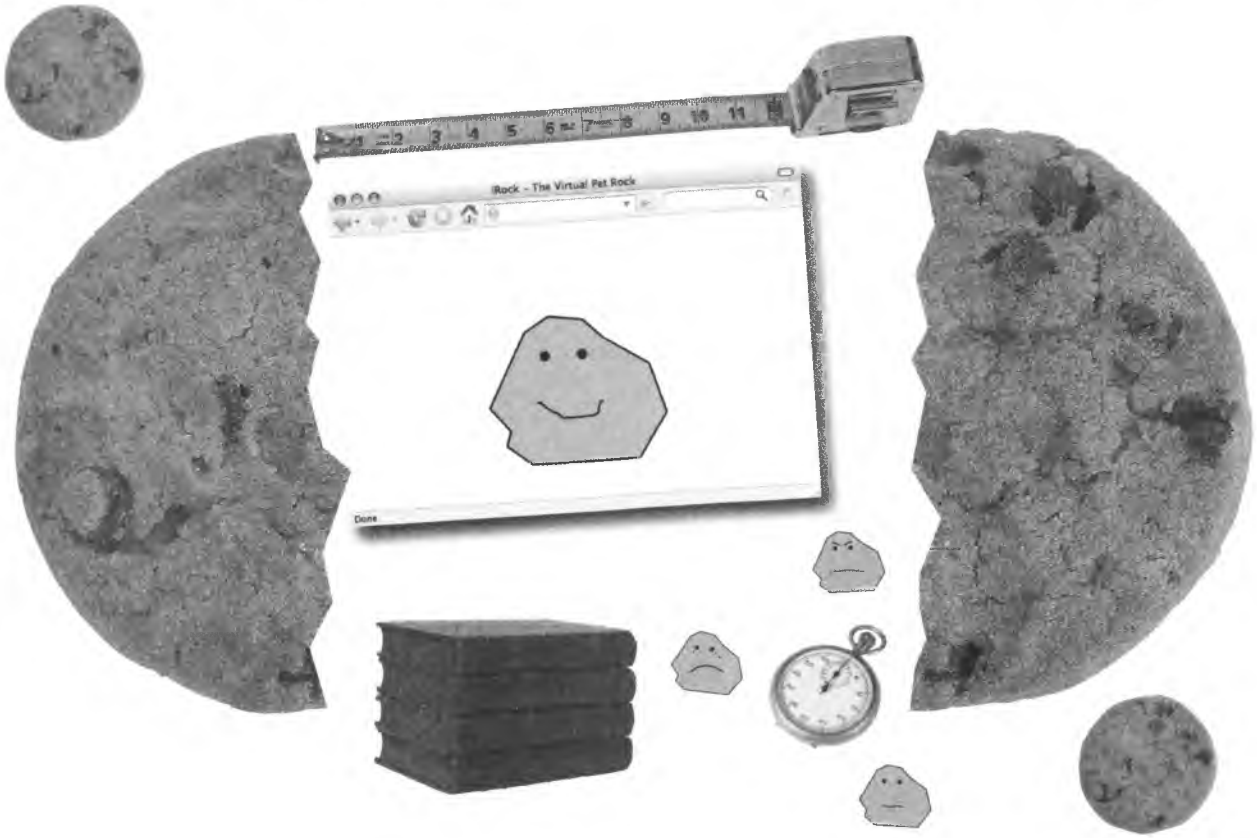
Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Почему JavaScript должен заботиться о клиенте?

Ум хорошо, а два лучше!





Клиент — это место запуска кода JavaScript, то есть JavaScript связан с браузером. Это хорошая новость, потому что в результате серверу не приходится сохранять куки!

4 Принятие решений

Если на дороге развилка...

Жизнь неотделима от принятия решений. Стоять или идти, пойти на сделку с негодяем или пойти в суд... Результата невозможно добиться без выбора. То же самое происходит в JavaScript — **вам приходится выбирать между различными вариантами сценария**. ***Приходится то и дело принимать решения***. Стоит ли поверить данным, введенным пользователем, и отправить его охотиться на львов? Или же проверить еще раз, может быть, он всего лишь пытался заказать билет до Львова? Выбор за вами!

Счастливчик, спускайся ко мне!

В сегодняшнем эпизоде захватывающего шоу «Заключим сделку?» мы выберем участника, которому улыбнется удача...

Не сомневаемся, что вы уже подпрыгиваете на стуле в предвкушении демонстрации Эриком умения заключать сделки. Но вот какой вопрос: почему ведущий шоу выбрал именно Эрика?

Выбор — это принятие решения

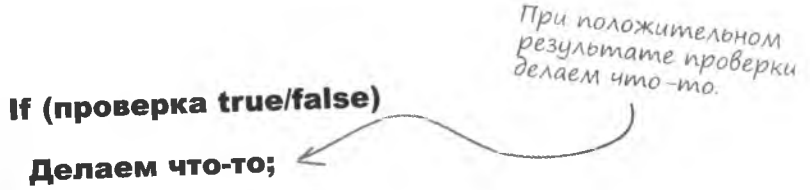
Все просто. Имя Эрика было написано на карточке! Обратите внимание на то, что факт принятия ведущим решения на основе записанных на карточках имен вы приняли как должное. Ведь он человек, а люди умеют обрабатывать информацию и принимать решения. А вот если бы он был сценарием...

Написанное на карточке имя стало причиной, по которой был выбран именно Эрик.

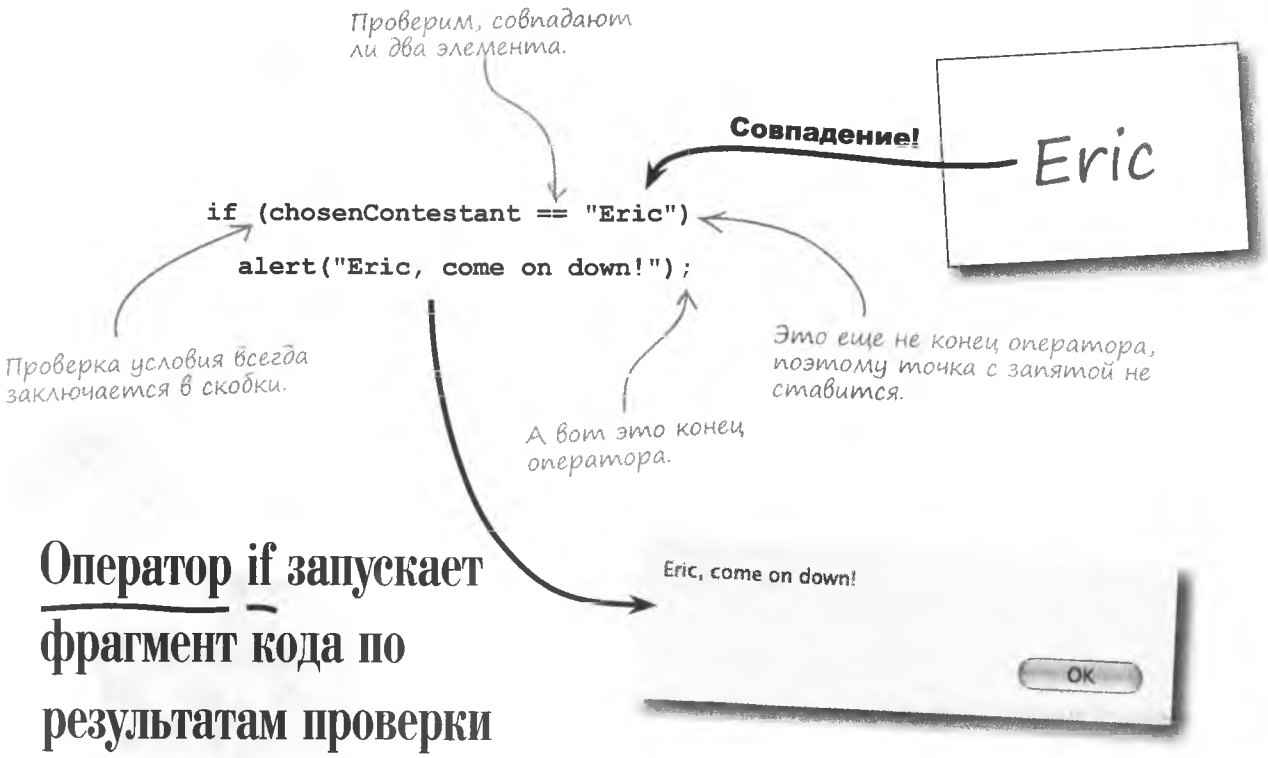
Вот ведь какой вопрос: каким образом сценарий может использовать фрагменты информации в качестве основы для действий? Узнать, чье имя появится на выбранной карточке, — только половина дела. А ведь нужно еще и оценить прочитанное и выбрать участника с аналогичным именем.

«если» так, то сделай что-нибудь

На самом деле JavaScript умеет обрабатывать информацию и принимать решения и делает это, например, при помощи оператора `if`. Этот оператор запускает код JavaScript по результатам проверки какого-либо условия.



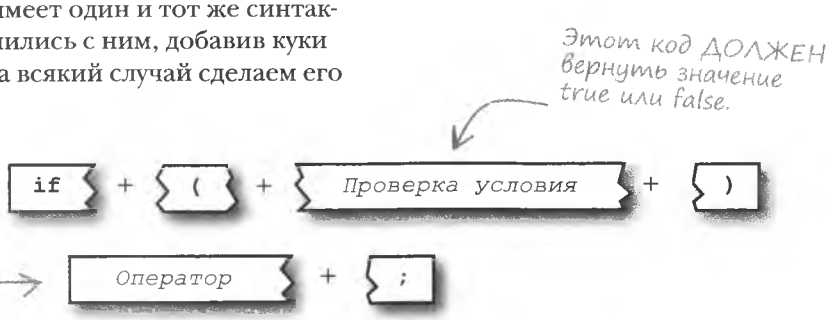
Если взглянуть на пример с игровым шоу с точки зрения оператора `if`, вы получите следующий код:



Оператор if запускает фрагмент кода по результатам проверки условия.

Оператор if

Оператор `if` всегда имеет один и тот же синтаксис. Вы уже познакомились с ним, добавив куки к объекту iRock, но на всякий случай сделаем его подробный анализ:



Отступы облегчают чтение кода. Написанный с отступом оператор является частью оператора "if."

Вот что следует знать об операторе `if`. Во-первых, запустить можно только один фрагмент кода, который пишется с отступом сразу под проверкой условия. Отступ не является обязательным требованием, но он позволяет определить, что вторая строчка также относится к оператору `if`. Вот основные принципы написания этого оператора:

1. Проверяемое условие заключается в скобки.
2. Отступ на пару пробелов во второй строчке.
3. Код, запускаемый в случае соблюдения условия.

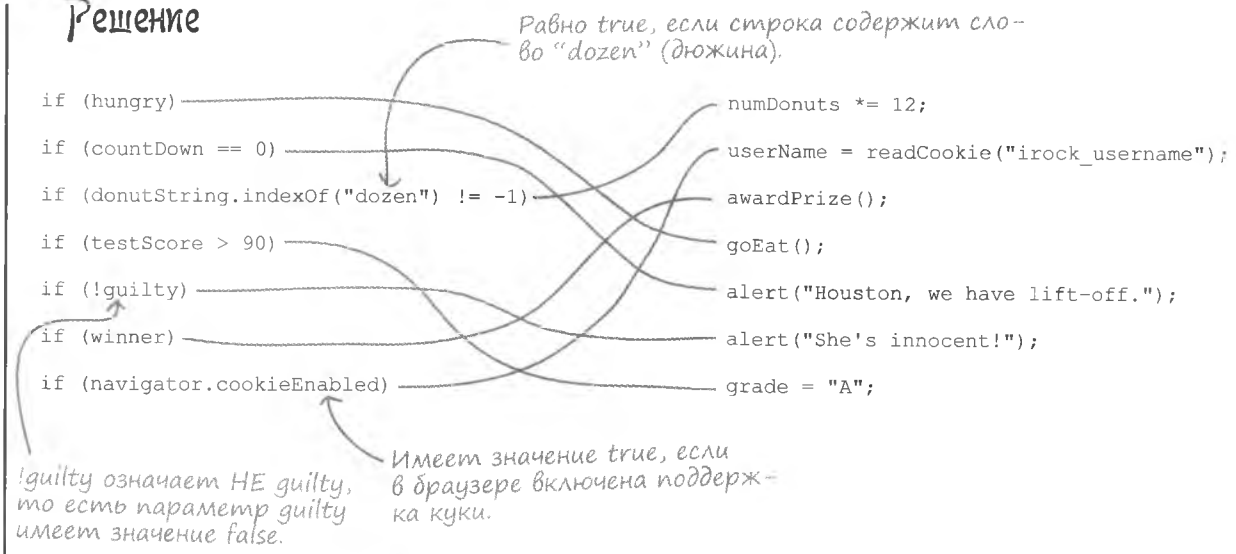
Упражнение Укажите, какие действия должны следовать за каждым из операторов `if`.

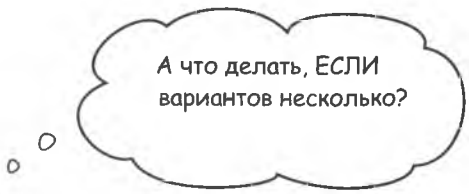
```
if (hungry)
if (countDown == 0)
if (donutString.indexOf("dozen") != -1)
if (testScore > 90)
if (!guilty)
if (winner)
if (navigator.cookieEnabled)
```

```
numDonuts *= 12;
userName = readCookie("irock_username");
awardPrize();
goEat();
alert("Houston, we have lift-off.");
alert("She's innocent!");
grade = "A";
```

Упражнение
Решение

Вот какие действия должны следовать за каждым из операторов `if`.





Делайте это... или вот то.

Стоит вам подумать, что тема раскрыта, как появляется нечто выходящее из ряда вон. Хотя в ситуации, когда приходится выбирать из нескольких вариантов, нет ничего экстраординарного... Шоколадное или ванильное, без кофеина или обычный, кажется, что множественные варианты сводятся к предложению: вам одного ***или*** другого. Именно поэтому оператор `if` умеет не только принимать решения, но и проделывать одну из двух возможных операций.

Когда вариантов два

Оператор `if` умеет делать выбор из двух предложенных вариантов. Вернемся к нашему шоу. Эрик пытается принять решение, какую из двух предоставленных ему возможностей предпочесть.

Первый вариант — чемодан A...

...а второй — чемодан B.

```
if (проверка условия)
  делай что-то;
else
  делай что-то еще;
```

В жизни обычно предоставляется выбор из более чем одного варианта. Оператор `if` дает нам (и Эрику) возможность выбрать чемодан A или чемодан B. Вот как это выглядит в JavaScript:

```
if (chosenCase == "A")
  openCase("A");
```

?

И что теперь? Каким образом написать "if" chosenCase не равен "A"?

Вы можете сделать множественный выбор

В случае множественного выбора оператор `if` превращается в оператор `if/else`, дающий возможность в случае несоблюдения заданного условия запустить другой фрагмент кода. Фактически вы говорите: «если условие соблюдено, запустим этот фрагмент кода, а если нет — вот этот фрагмент».

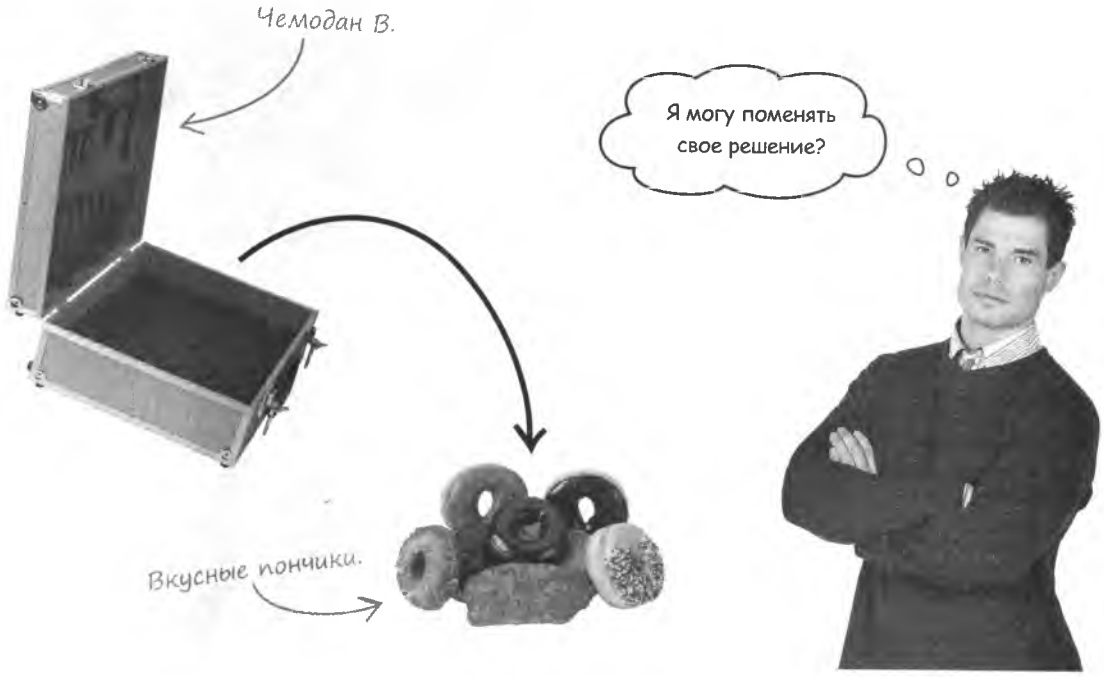
```
if (chosenCase == "A")
  openCase("A");
else
  openCase("B");
```

True

False

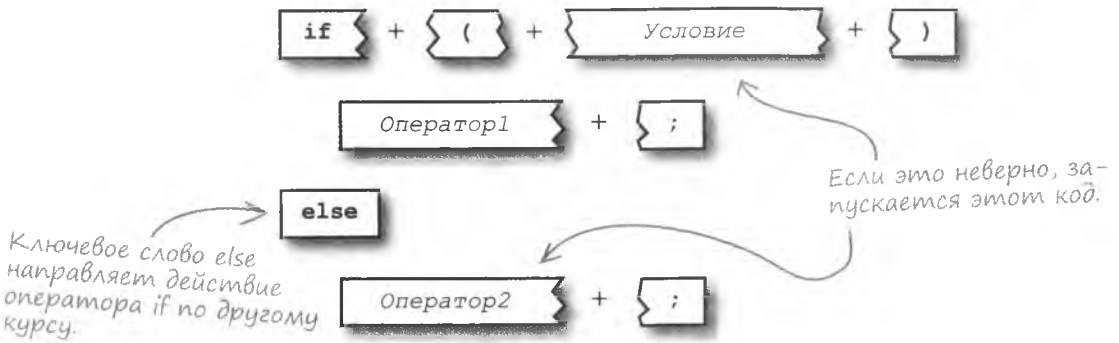
Оператор if/else предоставляет вам два варианта развития событий. Один — на случай выполнения условия, а другой — на случай его невыполнения.

Эрик выбирает чемодан B, то есть переменная `chosenCase` будет иметь значение «B». Так как заданное условие не выполняется, управление переходит к коду, стоящему после оператора `else`. К несчастью для Эрика, выбранный им чемодан наполнен пончиками, а вовсе не пачками денег, как он надеялся.



Ключевое слово else

Синтаксис оператора `if/else` практически не отличается от синтаксиса уже известного вам оператора `if`. Просто добавляется ключевое слово `else` с еще одним фрагментом кода, который запускается при несоблюдении условия:



Вот каким образом можно добавить второй вариант действия к оператору `if`:

1. Ключевое слово `else` после первого оператора.
2. Отступ на пару пробелов для следующей строчки.
3. Код, запускаемый при несоблюдении условия.

Часто Задаваемые Вопросы

В: Почему после скобок в операторе `if` отсутствует точка с запятой?

О: В JavaScript принято ставить точку с запятой после каждого оператора, и `if` не исключение. Однако оператор `if` — это не просто запись `if (`*`проверка условия`*`)`, это еще и код, который выполняется в случае положительного результата проверки. *И вот этот* код венчает точка с запятой. Так что оператор `if` заканчивается так, как нужно, если учесть, из каких фрагментов он состоит.

В: Что происходит, если условие не выполняется, а ключевое слово `else` отсутствует?

О: Ничего. В этом случае по результатам проверки условия не предпринимается никаких действий.

В: Можно ли использовать несколько ключевых слов `else` для выбора из большего числа вариантов?

О: Да. В структуру `if/else` можно добавить дополнительные варианты, но это вовсе не сводится к написанию еще одного ключевого слова `else`. Конструкцию `if/else` приходится вкладывать целиком, что легко может привести к путанице. Поэтому в JavaScript для такой ситуации предусмотрена другая структура — оператор `switch/case`, с которым мы также познакомимся в этой главе.

Пишем приключения

Элли пишет интерактивную историю «Приключения нарисованного человечка». На каждом этапе приходится принимать решения, и она надеется создать сценарий на JavaScript, который позволит поместить ее творение в Интернет для пользователей.

Пользователю представлена интерактивная история.

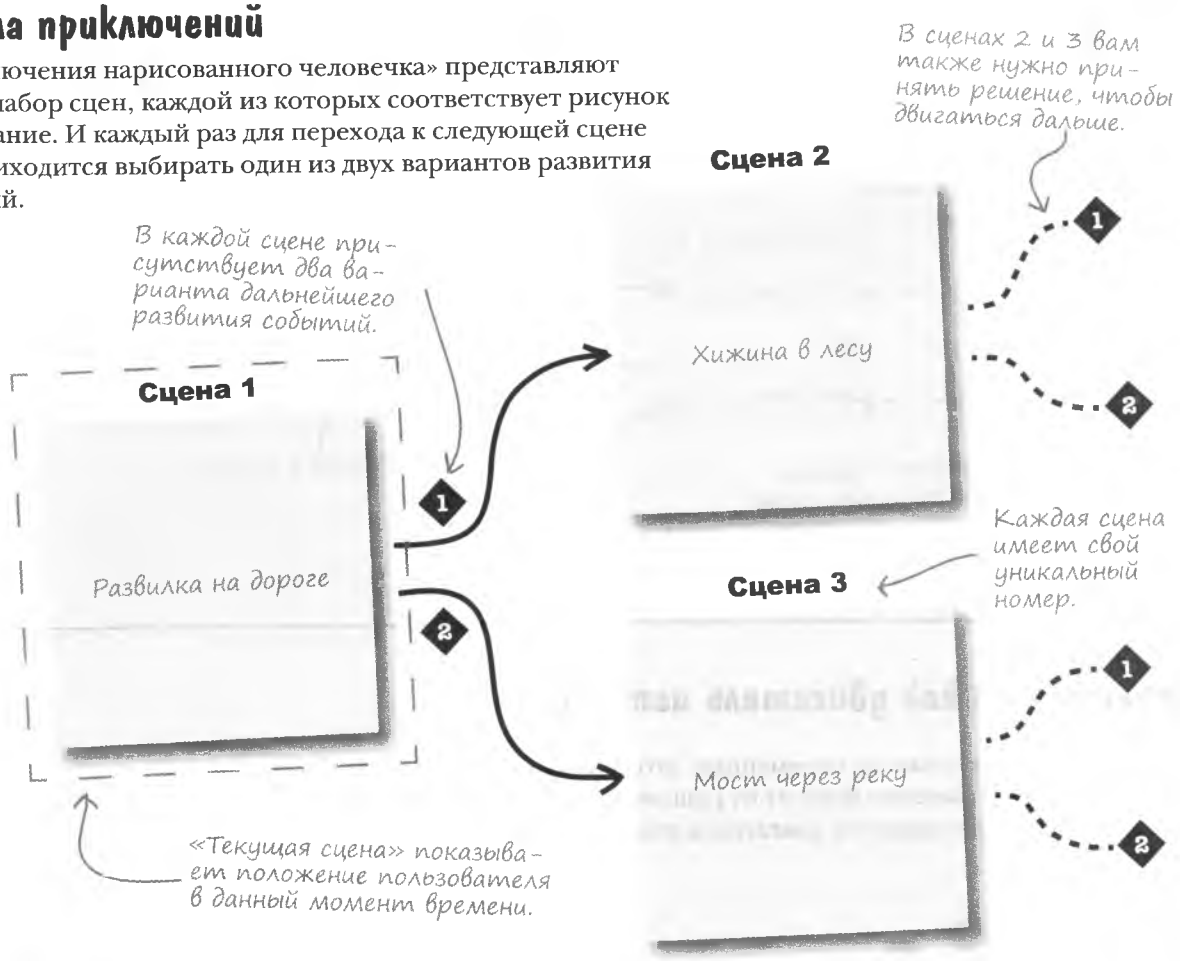
Я надеюсь, что JavaScript сделает мою историю по-настоящему интерактивной.

Элли мечтает о различных поворотах сюжета, но пока не знает, как их воплотить в реальность.

Элли хочет, чтобы пользователи на каждом этапе ее истории вынуждены были делать выбор. Вы можете сразу скачать все необходимые файлы по адресу *http://www.headfirstlabs.com/books/hfjs/*.

Схема приключений

«Приключения нарисованного человечка» представляют собой набор сцен, каждой из которых соответствует рисунок и описание. И каждый раз для перехода к следующей сцене вам приходится выбирать один из двух вариантов развития событий.



Возьми в руку карандаш

Напишите код принятия решений для первых трех сцен с оператором `if/else`. Подсказка: переменная `decision` хранит информацию о выборе пользователя, в то время как переменная `curScene` содержит сцену, к которой происходит переход.

.....

.....

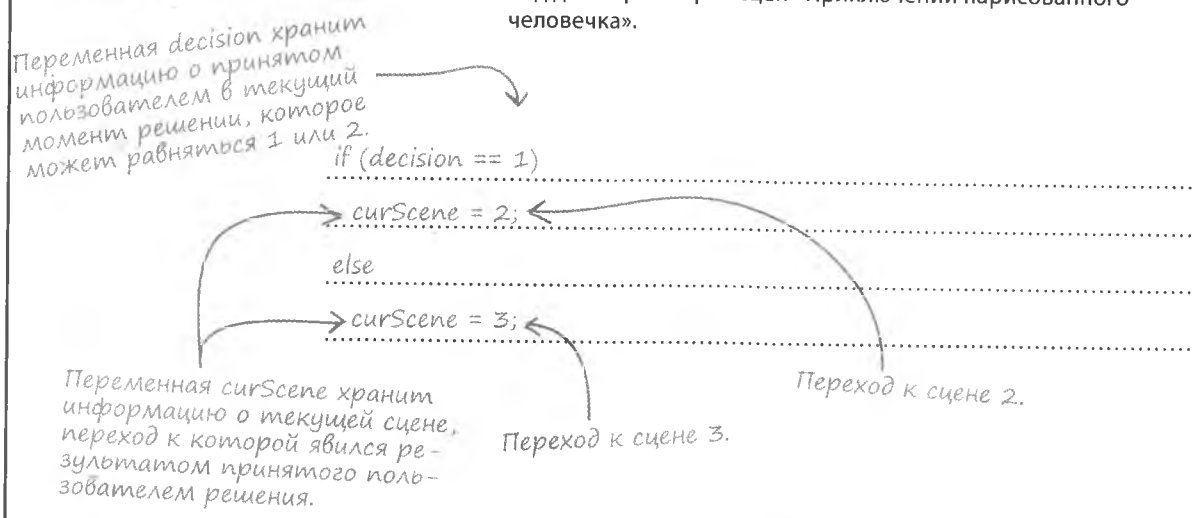
.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит написанный на основе оператора `if/else` код для первых трех сцен «Приключений нарисованного человечка».

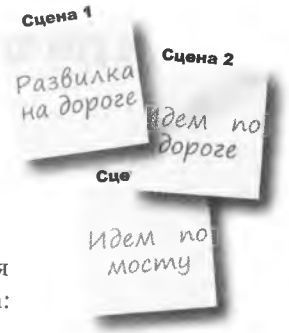


Переменные как двигатель истории

Посмотрим внимательно на переменные, которые фигурируют в нашей истории. Их значение зависит от решения, принятого пользователем, и именно оно является двигателем нашей истории.

decision

Нужно выбрать между значениями 1 и 2. Это определит, в какой сцене пользователь окажется на следующем этапе.



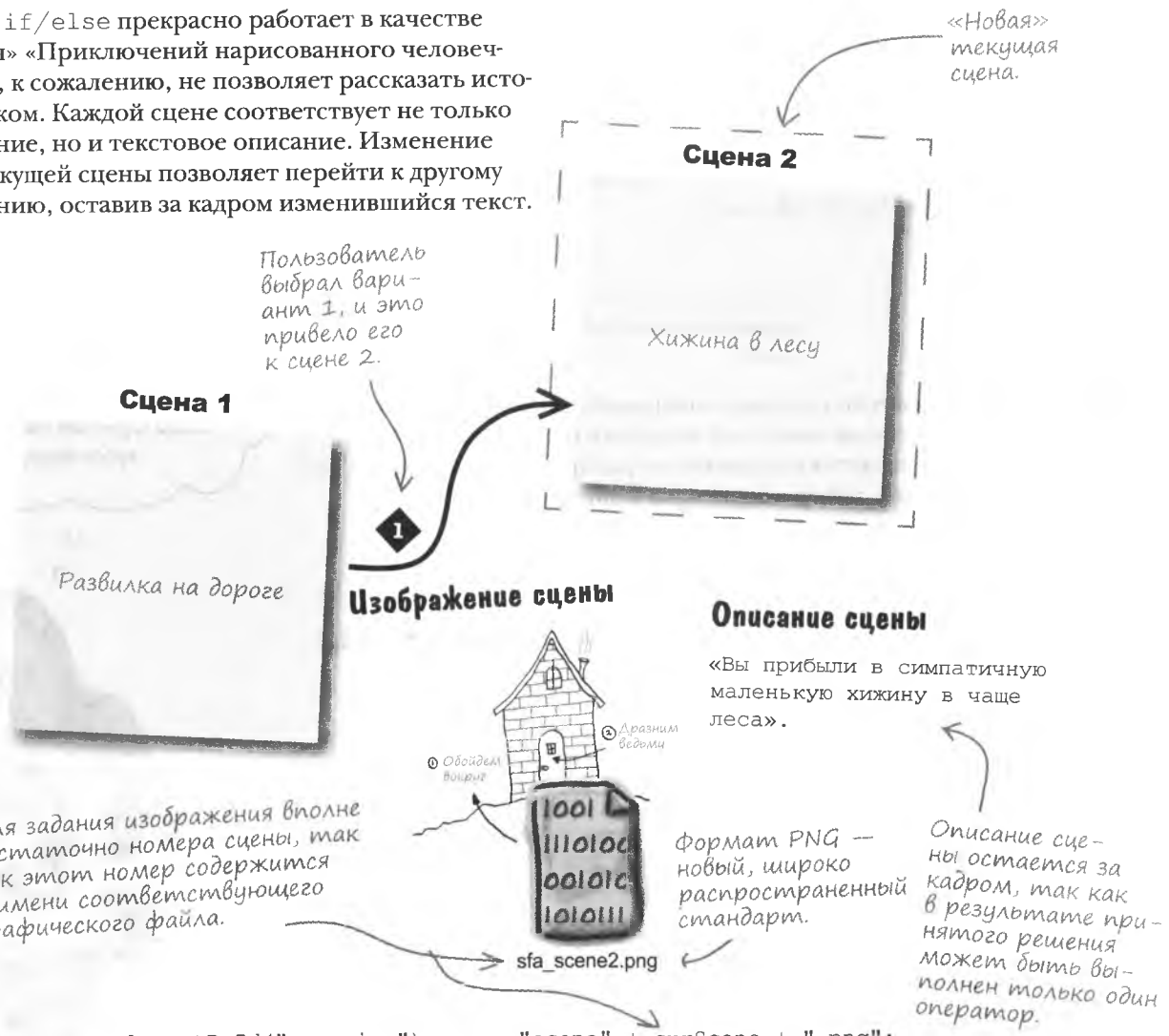
curScene

Сцена, в которой находится пользователь. Ее номер определяется переменной `decision`: сцена 1, сцена 2 и т. д.

Переменные `decision` и `curScene` работают совместно, сохраняя вариант, выбранный пользователем, и затем на его основе осуществляя переход к следующей сцене. Процедура повторяется от сцены к сцене до конца истории. Спасибо за это оператору `if/else`.

Недостающие части истории

Оператор `if/else` прекрасно работает в качестве «двигателя» «Приключений нарисованного человечка», но он, к сожалению, не позволяет рассказать историю целиком. Каждой сцене соответствует не только изображение, но и текстовое описание. Изменение номера текущей сцены позволяет перейти к другому изображению, оставив за кадром изменившийся текст.



Так как после каждой части оператора `if/else` можно запустить только один фрагмент кода, количество вариантов развития событий ограничено. Другими словами, вы не можете отобразить новый текст и новое изображение одновременно.

МОЗГОВОЙ ШТУРМ

Как бы вы реализовали выполнение набора операций в ответ на принятие решения?

Совмещение усилий

Элли нужно, чтобы на каждой ветви оператора `if/else` выполнялось несколько фрагментов кода. При переходе к следующей сцене она хочет менять не только изображение, но и сопроводительный текст:

Изменение картинки при переходе к выбранной сцене.

```
document.getElementById("sceneimg").src = "scene" + curScene + ".png";
alert(message);
```

Отображение описания выбранной пользователем новой сцены.

Вам нужно произвести несколько операций, несмотря на то что JavaScript в этом случае позволяет запускать всего один фрагмент кода. Решением является составной оператор. Для его создания достаточно заключить набор операторов в фигурные скобки ({}).

Ага! Составной оператор позволяет превратить набор фрагментов в один кусок кода.

```
doThis();
doThat();
doSomethingElse();
```

= 3 оператора

Количество открывающих и закрывающих скобок должно совпадать.

```
{
  doThis();
  doThat();
  doSomethingElse();
}
```

= 1 оператор

Составной оператор позволяет создать конструкцию `if/else`, реализуюшую в одной ветке несколько операций:

```
if (chosenDoor == "A") {
  prize = "donuts";
  alert("Вы выиграли коробку с пончиками!");
}
else {
  prize = "pet rock";
  alert("Вы выиграли камешек!");
}
```

Выполняйте набор действий в каждой ветке оператора if/else.

Часто Задаваемые Вопросы

В: Каким образом переменные влияют на развитие сюжета в «Приключениях»?

О: В каждый момент переменная `curScene` содержит номер текущей сцены. При этом демонстрируются соответствующее изображение и описание, а пользователю предлагается указать, к какой из следующих сцен он предпочитает перейти. Выбор пользователя — 1 или 2 — сохраняется в переменной `decision`. Вместе с переменной `curScene` она определяет следующую сцену. На основе значения этой переменной выбирается новая картинка, и во всплывающем окне появляется новое описание.

В: Зачем соединять несколько операторов в один?

О: В JavaScript часто требуется всего один оператор. В качестве аналогии вспомним про авиакомпании, которые позволяют взять с собой только два места багажа. Ничто не мешает вам перевозить произвольное количество вещей, главное — упаковать их таким образом, чтобы они занимали всего два места. Также устроены и составные операторы. Это своего рода контейнер, **воспринимаемый** сценарием как один оператор.

В: Почему в конце составного оператора нет точки с запятой?

О: Точка с запятой обозначает конец обычного, одиночного оператора. Соответственно, она завершает каждый составной оператор из формирующих операторов.

В: Относится ли метод к составным операторам?

О: Разумеется. Вы уже могли заметить, что код методов заключен в фигурные скобки. Поэтому на данном этапе методы можно воспринимать как большие составные операторы, которым вы передаете данные и от которых получаете результат.

Возьми в руку карандаш

Перепишите код для первого фрагмента `if/else`, переводящего героя «Приключений» к следующей сцене. На этот раз воспользуйтесь составным оператором, который задает и номер сцены, и ее текстовое описание.

Возьми в руку карандаш

Решение

Вот как должен выглядеть новый вариант кода для первого оператора `if/else`, направляющего героя «Приключений» к следующей сцене.

```
if (decision == 1) {
  curScene = 2;
  message = "You have arrived at a cute little house in the woods.";
}
else {
  curScene = 3;
  message = "You are standing on the bridge overlooking a peaceful
  stream.";
}
```

Номер новой сцены определяет решение пользователя.

Описание выбирается в соответствии с номером сцены.

Начните составной оператор с фигурной открывающей скобки.

Все содержимое составного оператора пишем с отступом.

В конце составного оператора ставим закрывающую фигурную скобку.

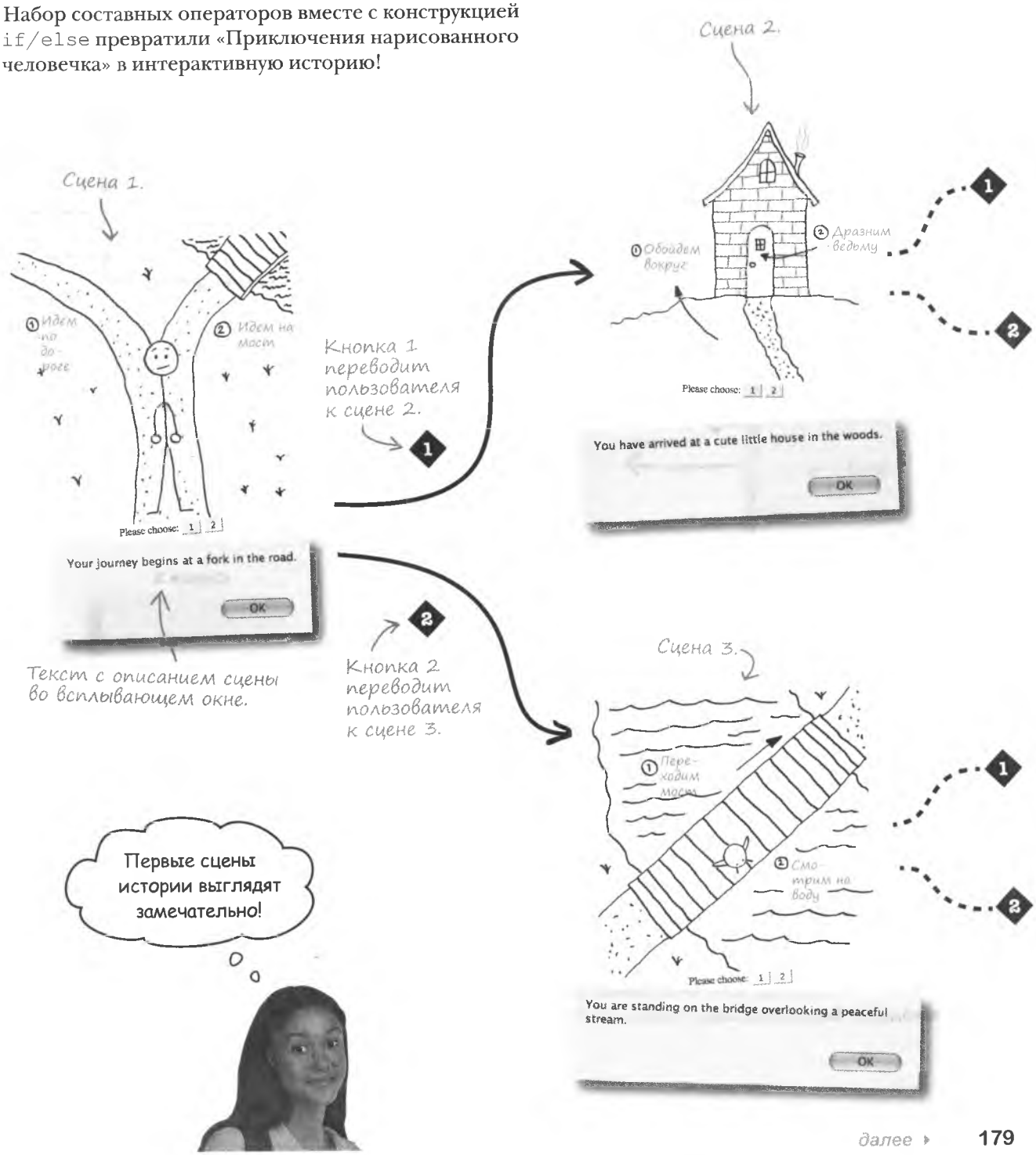
Для сцены 3 задано другое описание.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Оператор `if` запускает фрагмент кода JavaScript при соблюдении заданного условия.
- Результатом проверки условия в операторе `if` является значение `true` или `false`.
- Оператор `if/else` позволяет по результатам проверки условия запустить два фрагмента кода JavaScript.
- Составной оператор позволяет запустить набор фрагментов кода JavaScript вместо одного.
- Для создания составного оператора достаточно заключить набор операторов в фигурные скобки (`{}`).
- Составные операторы позволяют осуществлять набор действий в конструкциях `if` и `if/else`.

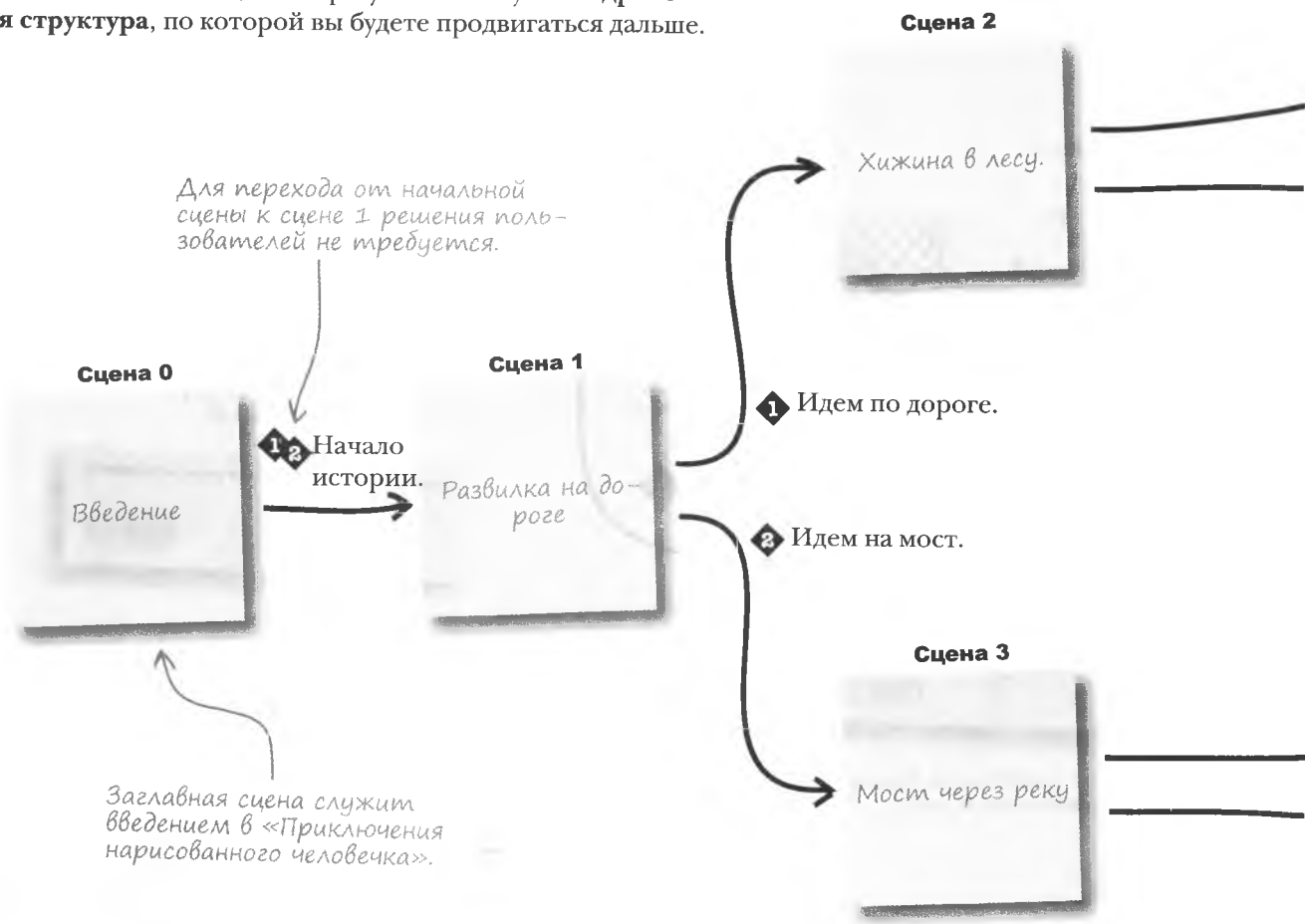
Приключения начинаются

Набор составных операторов вместе с конструкцией `if/else` превратили «Приключения нарисованного человечка» в интерактивную историю!

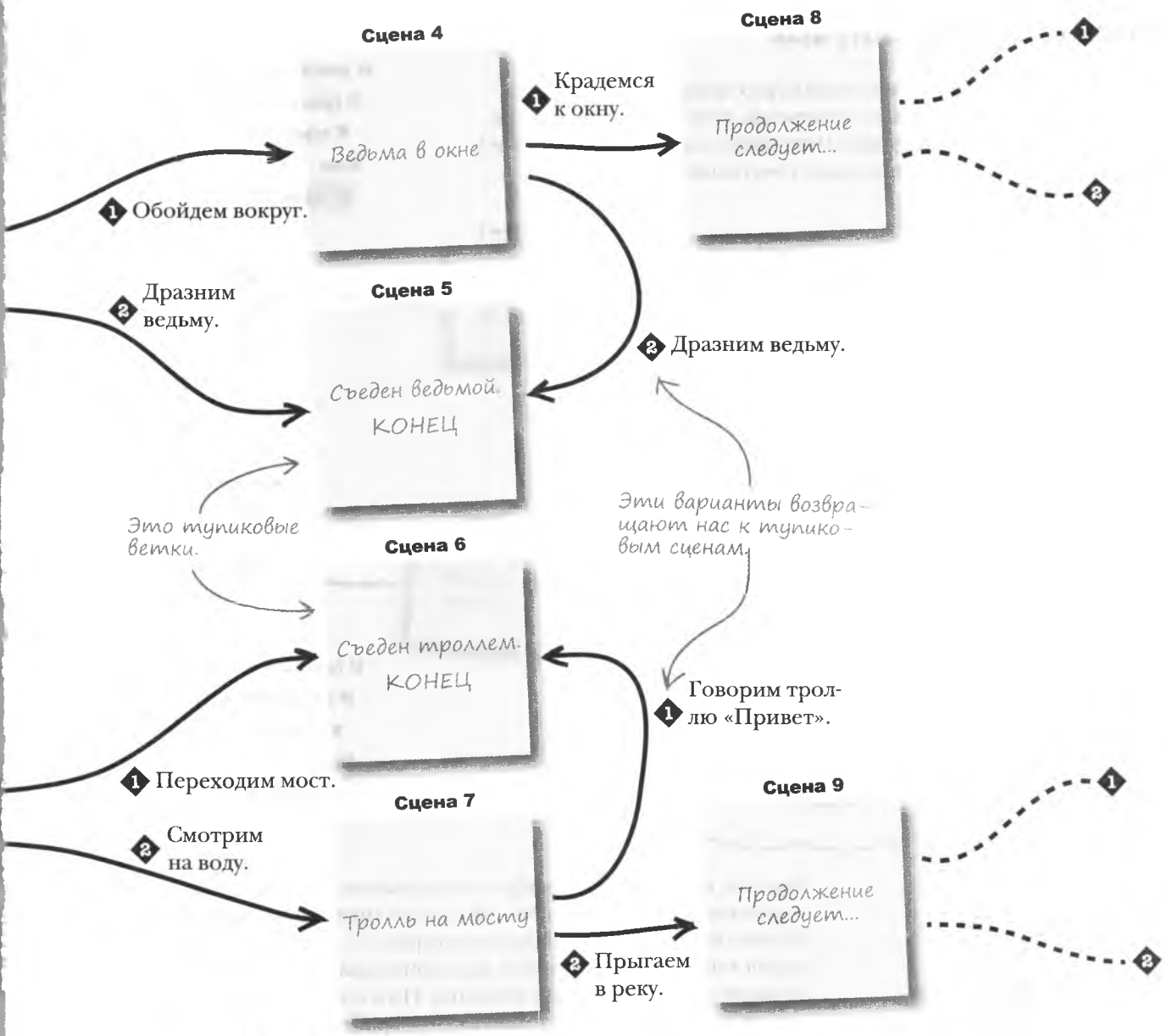


Остальные приключения

Единственное решение пользователя вряд ли позволит создать интересную историю. Но Элли планирует включить в повествование дополнительные сцены. В результате получится **древовидная структура**, по которой вы будете продвигаться дальше.



Кроме новых сцен, обеспечивающих дополнительные повороты сюжета, Элли создала вводную сцену, которая будет появляться перед сценой 1. Она уникальна тем, что не требует принятия решения для перехода. Все материалы, необходимые для этого урока, можно скачать по адресу *http://www.headfirstlabs.com/books/hfjs/*.

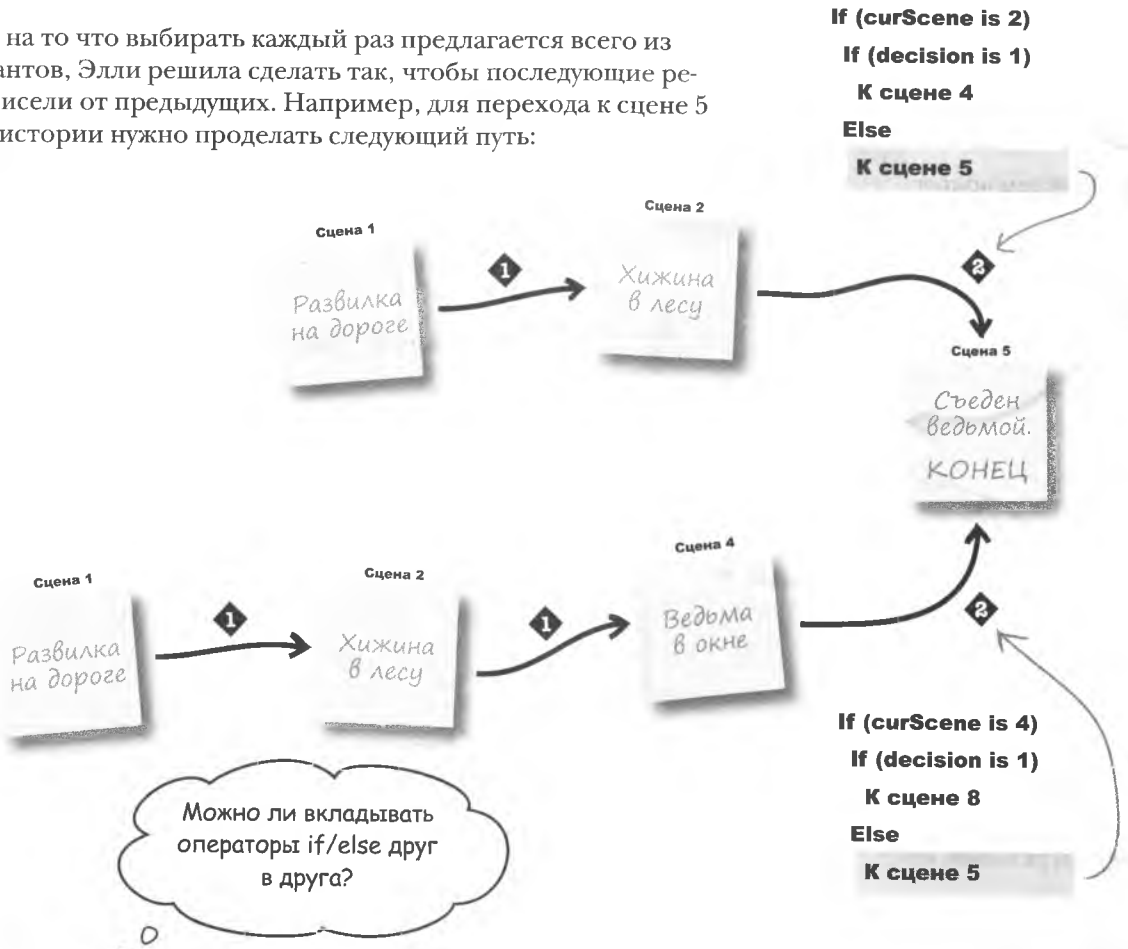


МОЗГОВОЙ ШТУРМ

Как будет выглядеть это дерево решений, написанное при помощи операторов `if/else`?

Запись при помощи if/else

Несмотря на то что выбирать каждый раз предлагается всего из двух вариантов, Элли решила сделать так, чтобы последующие решения зависели от предыдущих. Например, для перехода к сцене 5 от начала истории нужно проделать следующий путь:



Знания, какой вариант выбрал пользователь, недостаточно для перехода к следующей сцене. Это всего лишь один из факторов. Можно использовать набор операторов `if/else`, сначала проверяя текущую сцену, а затем предпринимая действие на основе принятого пользователем решения. Но в этом случае вы получите набор вложенных операторов `if`, что крайне неудобно.

Но ведь в реальном мире мы принимаем решения поэтапно. Вы ведь отвечали на вопрос: «Вы будете заказывать картофель фри?» Его обычно не задают, если вы заказали салат. Другими словами, чтобы получить такой вопрос, нужно перед этим дать определенный ответ, например: «Я буду чизбургер». То есть более поздние вопросы (картофель фри?) зависят от ответа на вопрос предыдущий (чизбургер или салат?).

Вложенный оператор if

JavaScript позволяет вкладывать операторы `if` друг в друга. Другими словами, ничто не запрещает вам задать новый вопрос, получив ответ на предыдущий. Такая конструкция называется вложенным оператором `if`.

Вложенный оператор if рассматривается только в том случае, когда внешний оператор возвращает значение true.

```
if (order == "cheeseburger") {
  if (wantFries)
    order += " fries";
}
else if (order == "salad") {
  if (wantFruit)
    order += " fruit";
}
```

Упрощенный вариант записи order = order + .. на выходе вы получите салат и фрукты или чизбургер и картофель фри.

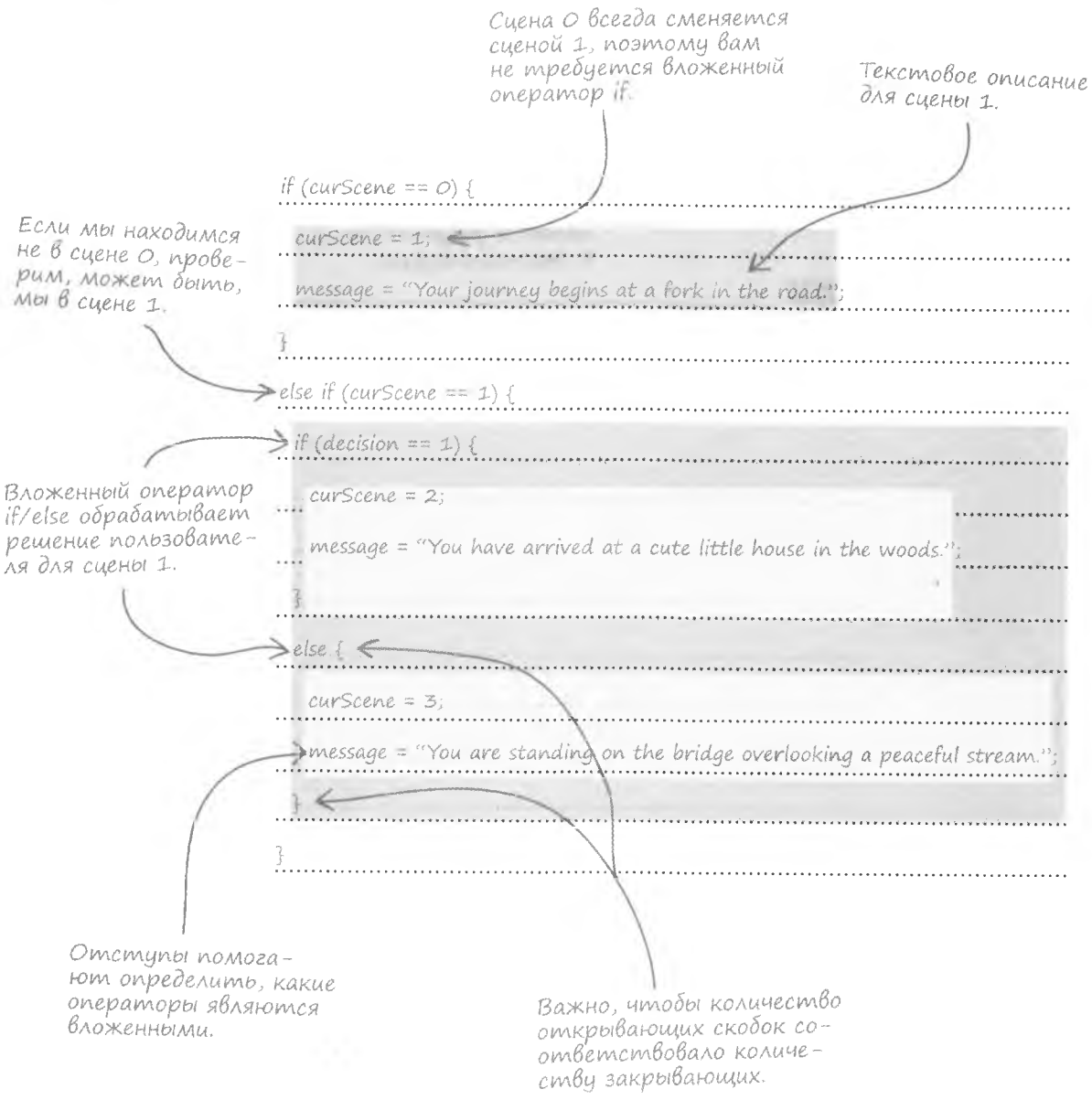
Возьми в руку карандаш

Напишите код для сцены 0 и сцены 1 «Приключений нарисованного человечка». При необходимости используйте вложенные операторы `if` и `if/else`.

Возьми в руку карандаш

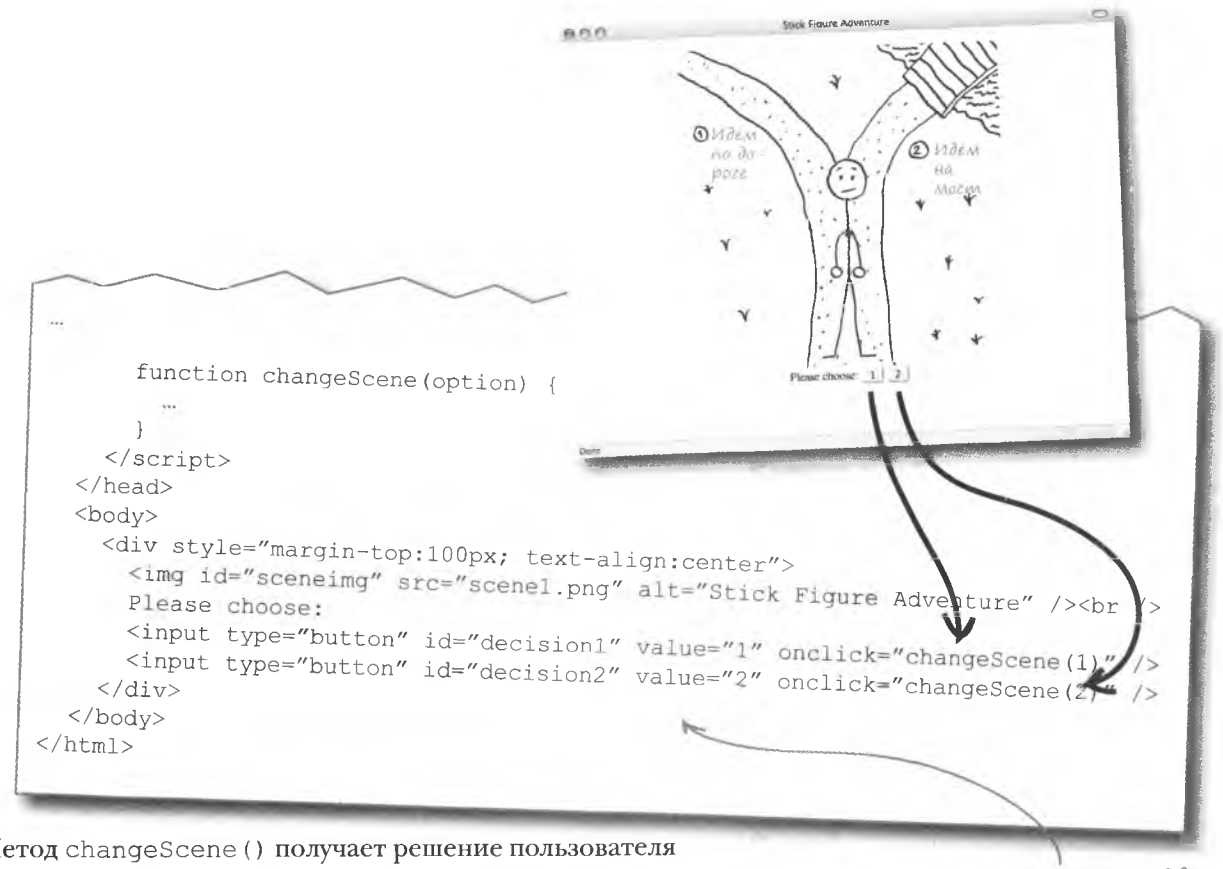
Решение

Вот как выглядит код принятия решений для сцены 0 и сцены 1 «Приключений нарисованного человечка».



Управление при помощи методов

Пользователь переходит от одной сцены к другой при помощи пары кнопок («1» и «2»). Щелчок на любой из кнопок вызывает метод `changeScene()`, отвечающий за переход к следующей сцене.



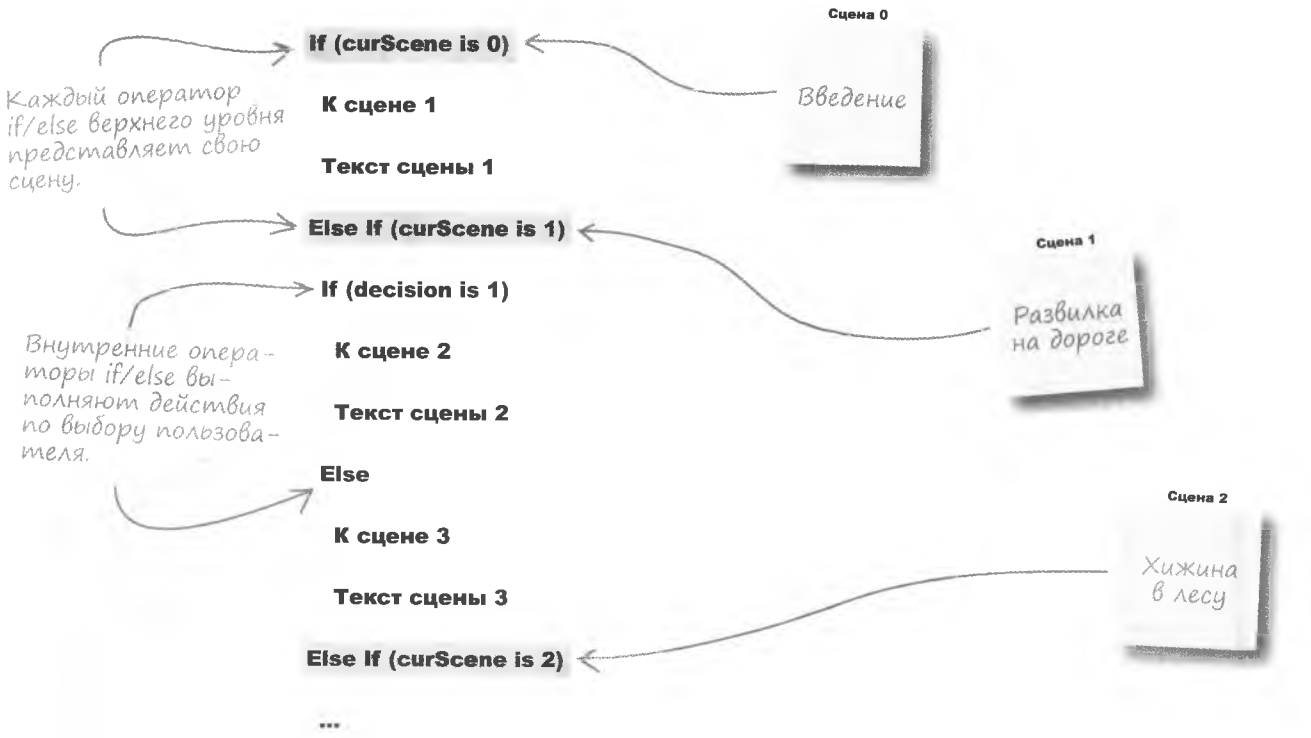
Метод `changeScene()` получает решение пользователя («1» или «2») в качестве аргумента. Этой информации достаточно для перехода к следующей сцене. Метод `changeScene()` делает следующее:

На веб-странице находится две кнопки, определяющие, к какой сцене пользователь перейдет на следующем этапе.

1. Присваивает переменной `curScene` номер новой сцены.
2. Присваивает переменной `message` новое описание.
3. Меняет изображение на основе значения переменной `curScene` и выводит новое описание.

Псевдокод

Элли примерно представляет, как написать на JavaScript код метода `changeScene()`, реализующий дерево решений нашей истории. Но из-за большого количества решений в коде легко запутаться. Поэтому имеет смысл сначала написать псевдокод, который проще воспринимается, а уже на его основе мы займемся программированием. Такой подход позволяет избежать путаницы и ошибок.



Часто Задаваемые Вопросы

В: Псевдокод похож на код JavaScript. Зачем он нужен?

О: Он позволяет упростить процесс преобразования сложного логического дерева в код JavaScript и минимизирует риск ошибок. Псевдокод имеет меньший уровень детализации, поэтому вы можете сфокусировать внимание на логике происходящего и переходе от одной сцены к другой. Привыкнув работать с псевдокодом, вы сможете практически на автомате переводить его в обычный код JavaScript.

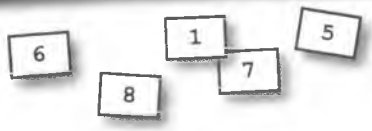
В: Требуются ли фигурные скобки для вложенных операторов `if`?

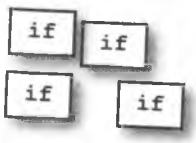
О: Нет. Если вы вкладываете всего один оператор `if`, фигурные скобки проще игнорировать, так как *технически* вам не нужен составной оператор. А вот при сложном вложении операторов `if` скобки могут пригодиться, так как они структурируют код и делают его более читабельным.

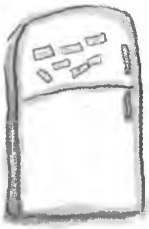
Магниты JavaScript

В методе `changeScene()` не хватает нескольких фрагментов. Воспользуйтесь магнитами, чтобы получить код, воспроизводящий диаграмму на странице 180. Помните, что показаны далеко не все варианты развития истории — некоторые сцены намеренно были оставлены за кадром.

```
function changeScene(option) {
  var message = "";

  .....(curScene == 0) {
    curScene =.....;
    message = "Your journey begins at a fork in the road.";
  }
  ...
  .....(curScene == 3) {
    .....(option == 1) {
      curScene = .....;
      message = "Sorry, a troll lives on the other side of the bridge and you " +
        "just became his lunch.";
    }
    ..... {
      curScene = .....;
      message = "Your stare is interrupted by the arrival of a huge troll.";
    }
  }
  .....(curScene == 4) {
    if (option == 1) {
      curScene = ..... ;
    }
    ..... {
      curScene = .....;
      message = "Sorry, you became part of the witch's stew.";
    }
  }
  ...

  document.getElementById("sceneimg").src = "scene" + curScene + ".png";
  alert(message);
}
```







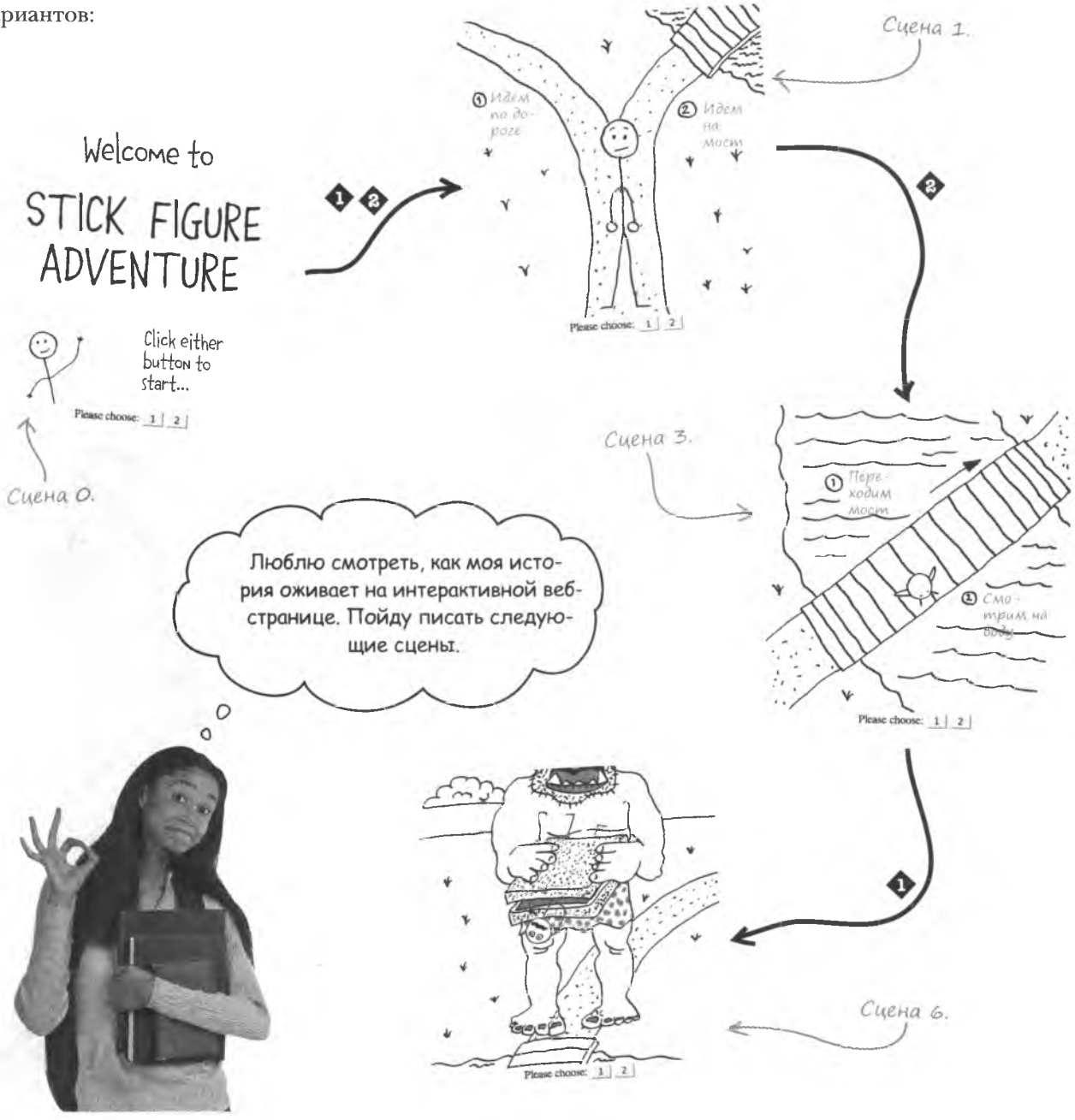
Решение задачи с магнитами

Вот каким образом требовалось дополнить метод `changeScene()`, чтобы получить код, воспроизводящий диаграмму со страницы 180.

```
function changeScene(option) {
  var message = "";

  if (curScene == 0) {
    curScene = 1;
    message = "Your journey begins at a fork in the road.";
  }
  ...
  else if (curScene == 3) {
    if (option == 1) {
      curScene = 6;
      message = "Sorry, a troll lives on the other side of the bridge and you " +
        "just became his lunch.";
    }
    else {
      curScene = 7;
      message = "Your stare is interrupted by the arrival of a huge troll.";
    }
  }
  else if (curScene == 4) {
    if (option == 1) {
      curScene = 8;
    }
    else {
      curScene = 5;
      message = "Sorry, you became part of the witch's stew.";
    }
  }
  ...

  document.getElementById("sceneimg").src = "scene" + curScene + ".png";
  alert(message);
}
```

Приключения продолжаются

Теперь сценарий «Приключений нарисованного человечка» отображает все дерево решений, позволяя направлять события по различным веткам. Вот один из вариантов:



Проблемы нарисованного человечка

К сожалению, Элли уже столкнулась с проблемой. Попросив своих друзей протестировать страницу с «Приключениями», она узнала о появлении пустого окна. «Окно-призрак» возникает при переходе от одной сцены к другой. То есть проблема каким-то образом связана с возвращением к сцене 0.

Оказалось, что две сцены действительно возвращают нас к сцене 0. Их номера 5 и 6 , и они соответствуют тупиковым веткам истории. В принципе имеет смысл, после того как история столь бесславно завершилась, вернуться в начало. Именно поэтому Элли написала код метода `changeScene()` таким образом, чтобы после этих сцен происходил переход к сцене 0:

```
else if (curScene == 5) {
  curScene = 0;
}
else if (curScene == 6) {
  curScene = 0;
}
```

Сцены 5 и 6 меняют значение переменной curScene, но ничего не делают с переменной message.

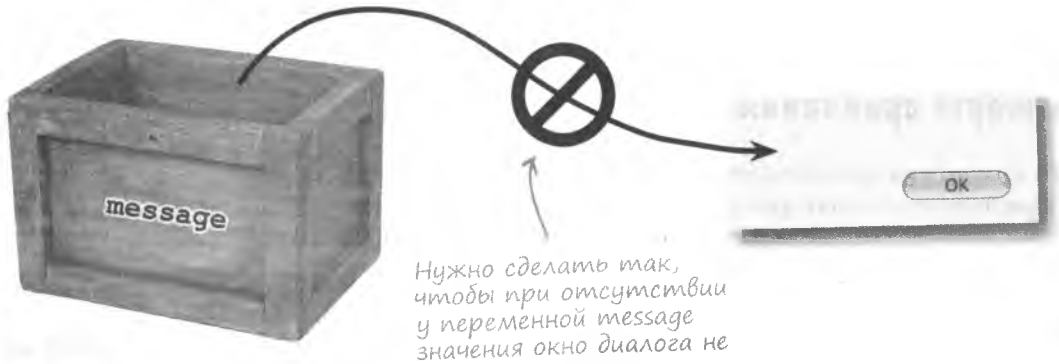
Кажется, что такой простой код не должен был преподнести нам никаких сюрпризов, но взглянем на последнюю строчку метода `changeScene()`, отвечающего за изменение изображения и описания при переходе к следующей сцене.

```
document.getElementById("sceneimg").src = "scene" + curScene + ".png";
alert(message);
```

Отображает текст с описанием сцены, хранящийся в переменной message.

!= m-c-c-c, мне нечего тебе сказать...

Проблема в том, что код «Приключений» всегда вызывает окно с описанием сцены, даже если описывать нечего, как, например, в сцене 0. Каким же образом проверить, содержит ли переменная `message` какой-нибудь текст?



Нужно сделать так, чтобы при отсутствии у переменной message значения окно диалога не появлялось.

Перед вызовом окна нужно проверить, не содержит ли переменная `message` пустой строки (`""`). Можно поставить и другое условие. Вызывать окно диалога, если переменная `message` не равна пустой строке. Это выглядит как решение проблемы задом наперед, но в данном случае мы рассуждаем в терминах проверки условия.

В то время как оператор (==) проверяет равенство двух элементов, оператор (!=) проверяет их неравенство.

Условие соблюдается для всех сцен, кроме сцены 6.

```
if (curScene != 6)
  alert("Thankfully, you haven't been eaten by the troll.");
```

Возьми в руку карандаш

Как будет выглядеть код, вызывающий окно диалога в случае, если переменная `message` содержит текстовые данные?

.....

.....

Возьми в руку карандаш

Решение

Вот как будет выглядеть код, вызывающий окно диалога в случае, если переменная `message` содержит текстовые данные.

Условие соблюдается, если переменная message содержит непустую строку.

```
if (message != "")
  alert(message);
```

Операторы сравнения

До этого момента мы рассматривали проверку условий, созданных только при помощи операторов равенства и неравенства. А ведь наши возможности этим не ограничиваются. Рассмотрим и другие операторы сравнения.

Равенство

```
x == y
```

Значение true, если x РАВЕН y.

Неравенство

```
x != y
```

Значение true, если x НЕ РАВЕН y.

Меньше чем

```
x < y
```

Значение true, если x МЕНЬШЕ, ЧЕМ y.

Больше чем

```
x > y
```

Значение true, если x БОЛЬШЕ, ЧЕМ y.

Отрицание

```
!x
```

Значение false, если x имеет значение true, и наоборот.

Меньше или равно

```
x <= y
```

Значение true, если x МЕНЬШЕ ИЛИ РАВЕН y.

Больше или равно

```
x >= y
```

Значение true, если x БОЛЬШЕ ИЛИ РАВЕН y.

Операторы сравнения JavaScript используются для построения выражений, которые затем комбинируются в единое значение. Это значение принадлежит к логическому типу (`true/false`), что делает его незаменимым в качестве проверяемого условия в операторах `if/else`.

Будьте осторожны!

= и == вовсе не одно и то же.

Для проверки равенства двух значений следует использовать оператор == и ни в коем случае не оператор =. Последний осуществляет присвоение значения.

Часто Задаваемые Вопросы

В: Почему оператор отрицания использует всего одно значение?

О: Этот оператор реализует очень простую задачу. Он меняет значение операнда на противоположное. В результате `true` превращается в `false`, и наоборот.

В: Я видел оператор отрицания рядом со значением, которое не имеет отношения к сравнению. Как он работает?

О: В таких случаях все зависит от того, каким образом JavaScript определяет «правдивость» значения. Используя не имеющие отношения к сравнению значения в ситуации, где сравнение ожидается, мы интерпретируем любое значение, отличное от `null`, `0` или `" "`, как `true`. Другими словами, как `true` рассматривается само наличие данных. Поэтому, когда вы видите оператор отрицания с несравниваемым значением, `null`, `0` и `" "` дают в результате `true`, а все остальные значения — `false`.

В: А что такое `null`?

О: `null` — это специальное значение, указывающее на отсутствие данных. Его смысл вы лучше поймете при рассмотрении объектов, которым будут посвящены главы 9 и 10.

Упражнение

Этот код выводит сообщение «I love Stick Figure Adventure!» Какие значения должны иметь переменные `a`, `b`, `c` и `d`, чтобы получился именно такой результат?

```
var quote = "";

if (a != 10)
  quote += "Some guy";
else
  quote += "I";
if (b == (a * 3)) {
  if (c < (b / 6))
    quote += " don't care for";
  else if (c >= (b / 5))
    quote += " can't remember";
  else
    quote += " love";
}
else {
  quote += " really hates";
}
if (!d) {
  quote += " Stick Figure";
}
else {
  quote += " Rock, Paper, Scissors";
}

alert(quote + " Adventure!");
```

a = ...

b = ...

c = ...

d = ...

Упражнение Решение

Вот какие значения должны иметь переменные a, b, c и d, чтобы во всплывающем окне появилось сообщение «I love Stick Figure Adventure!»

```
var quote = "";

if (a != 10)
  quote += "Some guy";
else
  quote += "I";
if (b == (a * 3)) {
  if (c < (b / 6))
    quote += " don't care for";
  else if (c >= (b / 5))
    quote += " can't remember";
  else
    quote += " love";
}
else {
  quote += " really hates";
}
if (!d) {
  quote += " Stick Figure";
}
else {
  quote += " Rock, Paper, Scissors";
}

alert(quote + " Adventure!");
```

a должно равняться 10.

b должно равняться 10 x 3.

c должно равняться 5.

a = 10

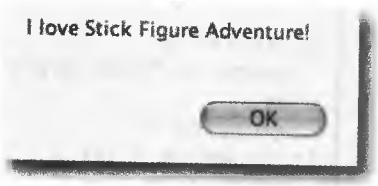
b = 30

c = 5

d = false

Переменная d должна иметь значение false, чтобы выражение !d имело значение true.

Вот сообщение, которое нам требовалось.

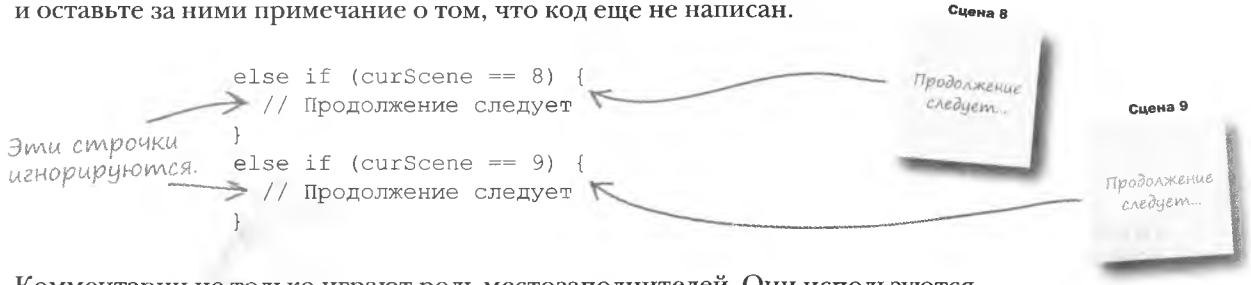


Комментарии

«Приключения» являются хорошим примером сценария с незавершенными фрагментами кода, ведь история еще не дописана. К примеру, сцены 8 и 9 помечены словосочетанием «Продолжение следует». Незаконченные области кода имеет смысл помечать комментариями, чтобы не забыть заполнить их позднее. Комментарии в JavaScript никоим образом не влияют на запуск и работу сценария.

Комментарии начинаются с //

Начинаясь с //, комментарии продолжаются до конца строки. Для создания местозаполнителя поставьте две косые черты и оставьте за ними примечание о том, что код еще не написан.




```
else if (curScene == 8) {
  // Продолжение следует
}
else if (curScene == 9) {
  // Продолжение следует
}
```

Комментарии не только играют роль местозаполнителей. Они используются в качестве примечаний, делающих код более понятным. То, что вы сейчас помните предназначение того или иного фрагмента кода, не означает, что вы будете помнить это и через год. Кроме того, ваш код может попасть в чужие руки, и именно примечания позволят быстро понять, что для чего предназначено.

Комментарий объясняет инициализацию переменной.

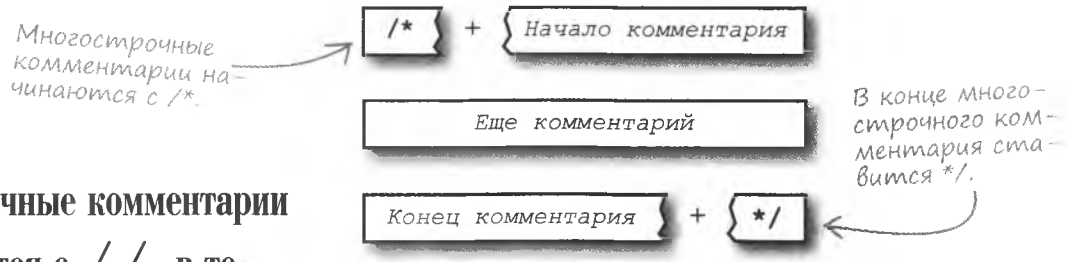
```
// Выбираем в качестве текущей сцену 0 (Введение)
var curScene = 0;
```

Инициализация переменной `curScene` в «Приключениях» стала понятней благодаря комментарию. Аналогичным комментарием можно снабдить инициализацию переменной `message`.

```
// Удаляем описание сцены
var message = "";
```

И снова комментарий поясняет назначение фрагмента кода.

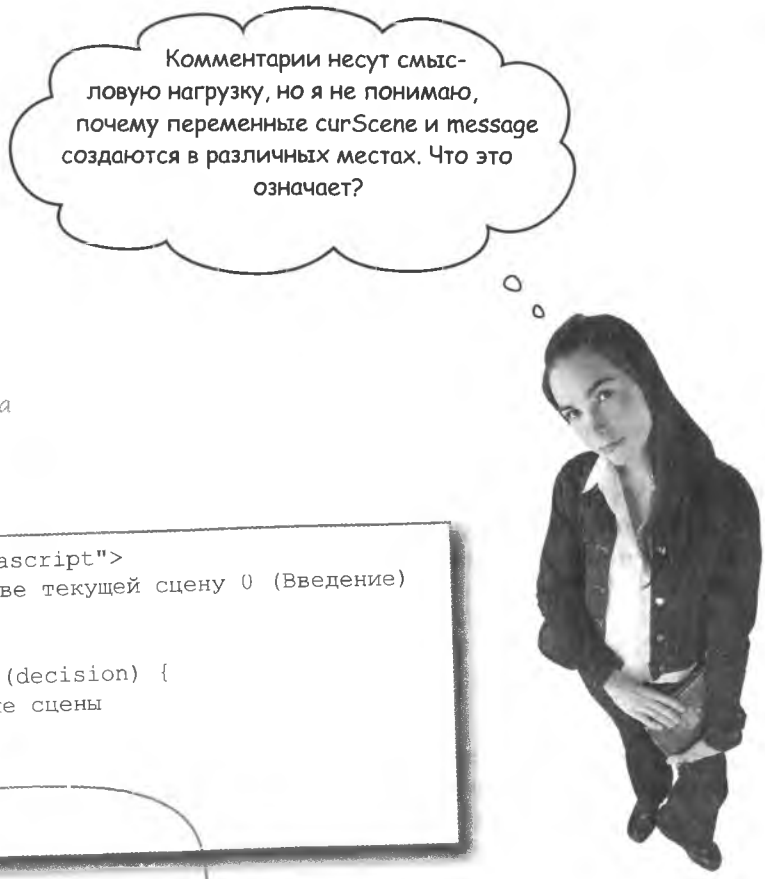
Существует возможность создавать комментарии, занимающие несколько строк.



Однострочные комментарии начинаются с //, в то время как многострочные комментарии заключены между /* и */.

Комментарий может иметь произвольную длину и занимать сколько угодно строк. Достаточно поставить в начале /*, а в конце */.

```
/* Эти три строчки кода представляют собой один
   большой комментарий. Серьезно, я не шучу.
   Кроме шуток, это все еще один комментарий. */
```

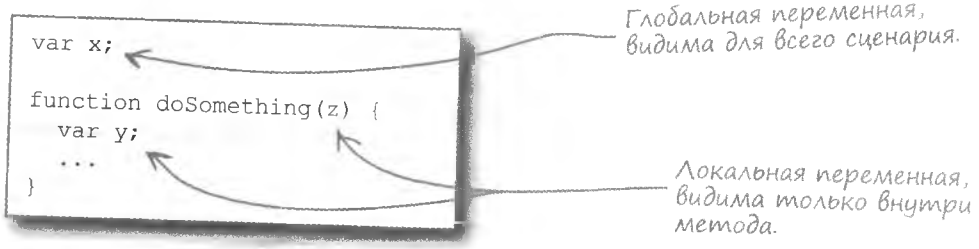


Место создания переменных

Как при покупке недвижимости, в JavaScript большое значение имеет местоположение. В «Приключениях» учитывается место создания переменных. Не случайно переменная `curScene` появилась вне метода `changeScene()`, в то время как переменная `message` — внутри него. Все дело в **области видимости**, управляющей жизненным циклом переменной и определяющей, какой код имеет к ней доступ.

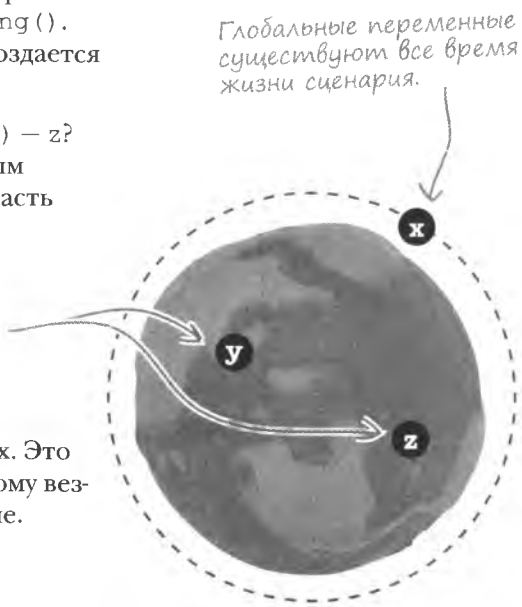
Область видимости

В JavaScript область видимости определяет способ доступа к данным. Некоторые данные видимы для всего сценария, в то время как видимость других ограничена фрагментом кода, например методом. Рассмотрим переменные, живущие в различных областях:



В данном коде x называется **глобальной** переменной, так как она была создана вне метода или любого другого фрагмента кода и, следовательно, видима для всего сценария. Более того, x «живет», пока работает сценарий. А вот y — это **локальная** переменная, видимость которой ограничена методом `doSomething()`. Она существует только во время работы этого метода — создается при его запуске и уничтожается после завершения.

А что можно сказать про аргумент метода `doSomething()` — z? Аргументы методов аналогичны уже инициализированным локальным переменным. Так что z имеет ту же самую область видимости, что и y, и доступна только внутри метода.



По возможности следует ограничивать видимость данных. Это помогает избежать их случайного редактирования. Поэтому везде, где только можно, используйте локальные переменные.

МОЗГОВОЙ ШТУРМ

Как использовать локальные и глобальные переменные в коде «Приключений нарисованного человечка»?

Проверим область видимости

Теперь, вооруженные сведением об области видимости, взглянем еще раз на переменные нашего сценария. Это позволит лучше понять, почему переменные следует создавать в разных местах.

Значение переменной message каждый раз сбрасывается методом changeScene(), поэтому сделаем ее локальной.

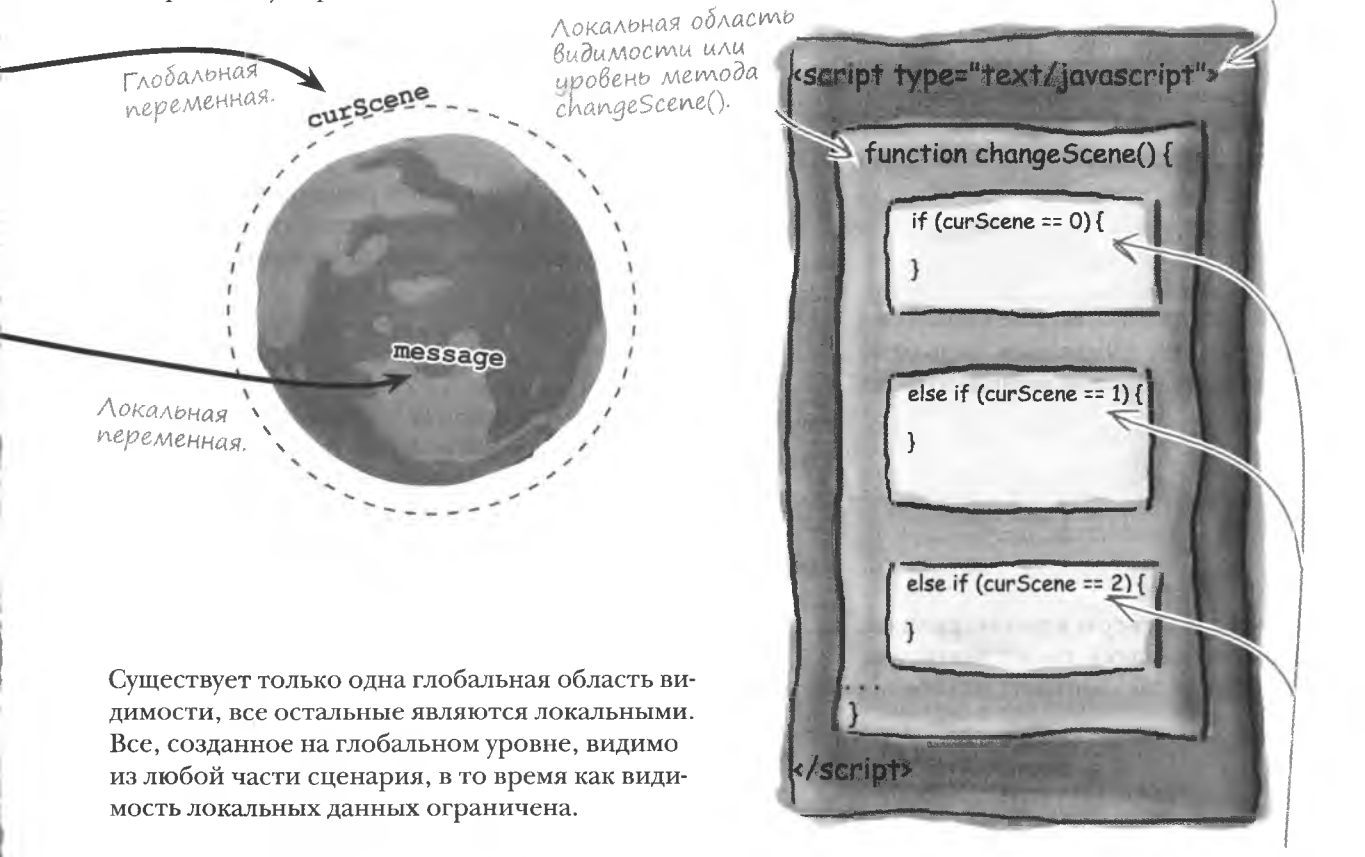
Значение переменной curScene задается между вызовами метода changeScene(), значит, это глобальная переменная.

```
<script type="text/javascript">
  // Выбираем в качестве текущей сцену 0 (Введение)
  var curScene = 0;

  function changeScene(decision) {
    // Удаляем описание сцены
    var message = "";

    if (curScene == 0) {
      curScene = 1;
      message = "Your journey begins at a fork in the road.";
    }
    else if (curScene == 1) {
      if (decision == 1) {
        curScene = 2
        message = "You have arrived at a cute little...";
      }
      else {
        curScene = 3;
        message = "You are standing on the bridge...";
      }
    }
    else if (curScene == 2) {
      ...
  }
</script>
```

Вопрос в том, нужно ли сохранять значение переменной вне области видимости метода `changeScene()`. Для переменной `message`, значение которой сбрасывается каждый раз перед началом работы метода, этого не требуется. А вот переменная `curScene` используется для проверки условия в нескольких операторах `if/else`, поэтому ее значение должно сохраняться вне зависимости от вызова метода. **В итоге получаем, что переменную `message` имеет смысл сделать локальной, в то время как переменная `curScene` в нашем примере должна быть глобальной.**

Где живут мои данные?

Если понятие области видимости все еще остается для вас неясным, различные части сценария можно представить в виде контейнеров, в которых живут наши данные. К примеру, сценарий «Приключений» имеет несколько областей видимости, которым могут принадлежать данные.



Существует только одна глобальная область видимости, все остальные являются локальными. Все, созданное на глобальном уровне, видимо из любой части сценария, в то время как видимость локальных данных ограничена.

Часто Задаваемые Вопросы

В: В переменной какого типа сохранить мои данные: локальной или глобальной?

О: Это зависит от способа, которым вы собираетесь использовать данные. Хотя есть и обобщенное правило. Все переменные нужно стараться делать локальными, превращая в глобальные только те, которые иначе не работают.

Беседа у камина

Локальная и глобальная переменные обсуждают важность расположения данных.

Локальная переменная:

Фокусироваться нужно только на том, что происходит вокруг тебя. Я понятия не имею, что происходит у соседей, и очень этим довольна.

Звучит соблазнительно, но мне нравится безопасность моего положения. Ведь меня не может достать никто из внешнего мира.

Честно говоря, я не верю в реинкарнацию, а для хранения данных я так же важна, как и ты. Я просто не даю смотреть на себя всем и каждому.

Зато, когда сценарию нужно, чтобы некоторая информация была недоступна вне определенного фрагмента кода, я незаменима.

Именно поэтому люди считают, что обе мы одинаково полезны.

Глобальная переменная:

Дружище, тебе следует расширить свой кругозор. Выйди из своего кокона и исследуй остальные части нашей вселенной.

Может быть и так, но понимаешь ли ты, что твоя маленькая жизнь бессмысленна в общей схеме сценария? Ты рождаешься и умираешь каждый раз, когда возникает твой маленький мирок, в то время как я тут надолго. Пока жив сценарий, жива и я.

Это да. Не могу не признать, что раз или два меня использовали неправильно, но моим преимуществом является способность сохранять значение при всех обстоятельствах. Как только сценарию требуется фрагмент данных, помнящий свое значение и доступный отовсюду, зовут меня.

Звучит заманчиво, но я все-таки предпочитаю приватности доступность и постоянство.

Часто Задаваемые Вопросы

В: Что получится, если в комментарий написать код JavaScript?

О: Ничего! Комментарии игнорируются интерпретатором JavaScript, поэтому, что бы вы туда ни поместили, он этого просто не заметит. Именно поэтому комментарии порой используются в качестве временного хранения кода при попытках отследить проблему или попробовать другой подход к решению задачи.

В: Может ли строчка кода JavaScript заканчиваться комментарием?

О: Да. В этом случае код все равно запускается, так как он не является частью комментария. Плюс комментарий вовсе не обязан занимать целую строчку — он начинается с `//` и заканчивается с концом строки. Поэтом если за `//` находится фрагмент кода, он будет работать.

В: Почему в конце комментариев не ставится точка с запятой?

О: Комментарии не являются операторами JavaScript. Они всего лишь предоставляют дополнительную информацию, как сноски в книге. Вам следует помнить, что интерпретатор JavaScript комментарии игнорирует — они предназначены для людей, а не для программы.

В: Что означает «уровень сценария» при создании глобальных переменных?

О: Это самый верхний уровень, расположенный сразу в теге `<script>`. Уровень сценария располагается вне методов и других фрагментов кода, и все, на этом уровне созданное, считается глобальным. То есть оно существует все время жизни сценария и доступно из любой его части.

В: Переменная, созданная внутри составного оператора, является глобальной или локальной?

О: Составной оператор создает новую область видимости, поэтому все переменные, возникшие внутри него, являются локальными. Их можно рассматривать как временные, так как они возникают и исчезают при каждом вызове составного оператора.

В: Материал, посвященный локальным и глобальным переменным, выглядит крайне сложным. Это на самом деле так?

О: Не совсем. Главное — запомнить, что локальные переменные идеальны для хранения временной информации, которую не нужно помнить вне метода или другого фрагмента кода. Если же данные требуются все время жизни сценария, нужно использовать глобальные переменные. Вы удивитесь, но большинство данных в сценариях обычно является временным, поэтому локальные переменные используются чаще глобальных.

Локальные переменные хранят временную информацию, в то время как глобальные сохраняются все время жизни сценария.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Комментарии**, в частности, напоминают о коде, который необходимо добавить позднее.
- Не бойтесь писать много комментариев, так как это делает код более понятным.
- Начало **однострочного комментария** отмечают двумя косыми чертами (`//`).
- **Многострочные комментарии** начинаются с `/*` и заканчиваются `*/`.
- Глобальные переменные создаются на уровне сценария и сохраняются **до конца сценария**.
- Локальные переменные создаются внутри фрагментов кода и не видны извне.
- Лучше использовать локальные переменные, так как доступ к ним больше контролируется.

Выбор из пяти

Помните нашего счастливчика Эрика? Кажется, он покончил с пончиками и перешел на следующий тур шоу «Заключим сделку?» И теперь перед ним стоит действительно сложная задача — нужно выбрать один из пяти вариантов.

МОЗГОВОЙ ШТУРМ

Каким образом вы написали бы на JavaScript сценарий выбора из пяти вариантов?

Выбор из пяти

Замечательная идея! Сам по себе оператор `if/else` позволяет выбрать один вариант из двух, но набор из таких операторов, вложенных друг в друга, позволяет выбирать из какого угодно количества вариантов.

```
if (chosenCase == "A")
  openCase("A");
else if (chosenCase == "B")
  openCase("B");
else if (chosenCase == "C")
  openCase("C");
else if (chosenCase == "D")
  openCase("D");
else if (chosenCase == "E")
  openCase("E");
```

Код работает, но, чтобы добраться до последнего чемодана, нужно выполнить проверку пяти условий, что неэффективно.

Переусложнение конструкции

Вложенные операторы `if/else` прекрасно работают... но они неэффективны в основном потому, что не предназначены для выбора из более чем двух вариантов. Достаточно посчитать, сколько раз придется осуществить проверку условий, прежде чем мы доберемся до последнего чемодана E. Чтобы его открыть, нам потребуется предварительная проверка еще четырех условий.


Разве не здорово было бы выбирать из множества вариантов без набора этих неэффективных операторов if/else?

Оператор switch/case

Для тех случаев, когда требуется сделать выбор из множества вариантов, в JavaScript существует специальный оператор. Он называется `switch/case`, попробуем с его помощью облегчить выбор Эрика:

Оператор switch/case позволяет эффективно выбирать из множества вариантов.

Вторая часть названия оператора никак не связана с именем переменной.

```
switch (chosenCase) {
  case "A":
    openCase("A");
    break;
  case "B":
    openCase("B");
    break;
  case "C":
    openCase("C");
    break;
  case "D":
    openCase("D");
    break;
  case "E":
    openCase("E");
    break;
}
```

Содержимое чемодана, выбранного Эриком, — это контролируемый кусок информации для оператора switch/case.

Код, определяющий, какие действия следует предпринять, располагается сразу после оператора case.

Оператор case представляет вам все возможные варианты.

Оператор break заканчивает каждую из возможных веток выбора, немедленно завершая работу оператора switch/case.

Оператор switch/case по структуре напоминает большой составной оператор.

Упражнение

Правда или ложь? Оператор `switch/case` обладает той же функциональностью, что и оператор `if/else`.

☐ **Правда** ☐ **Ложь**

Упражнение
Решение

Правда или ложь? Оператор `switch/case` обладает той же функциональностью, что и оператор `if/else`.

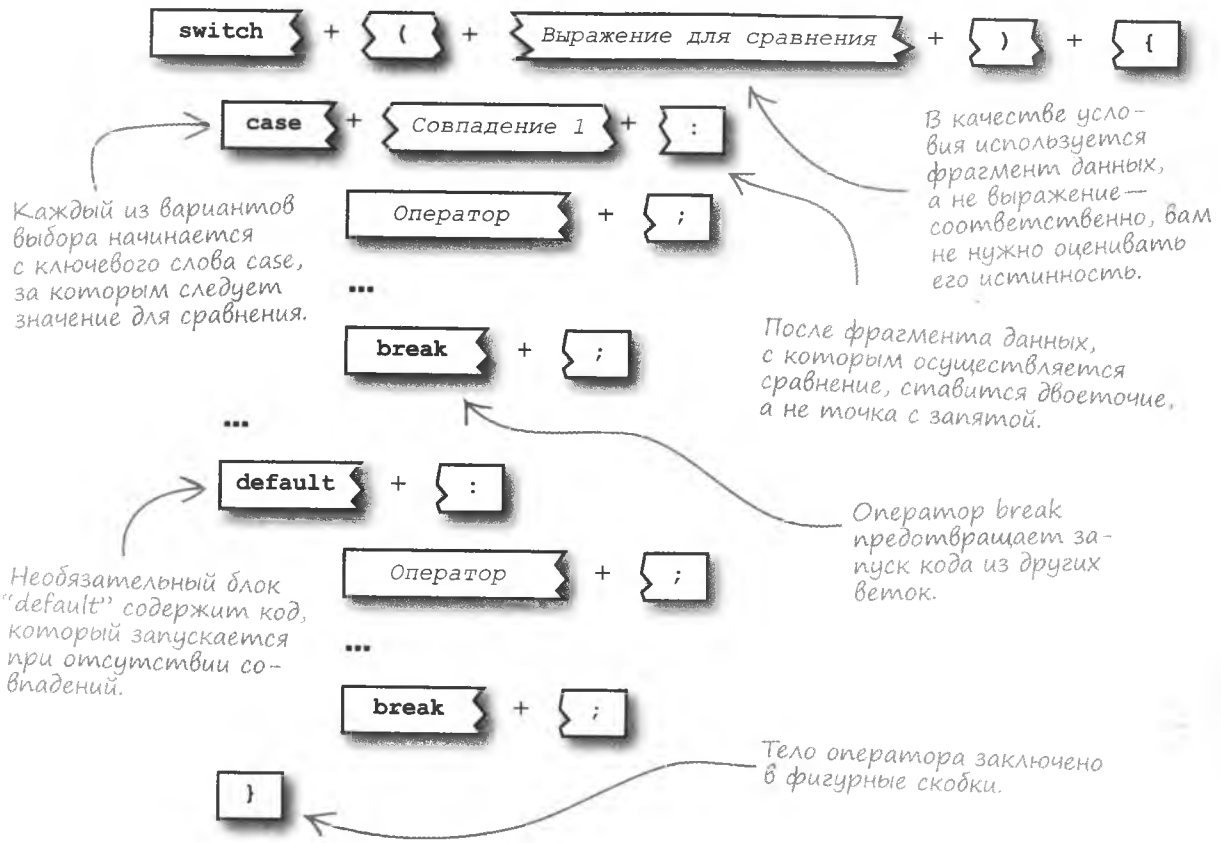
☐ **Правда**

☑ **Ложь**

У оператора `if/else` в качестве проверяемого условия может выступать выражение, в то время как у оператора `switch/case` это должен быть просто фрагмент данных.

Анализ оператора switch

Теперь, когда вы посмотрели на оператор `switch/case` в действии, посмотрим более внимательно на его синтаксис и функциональность.



Ваш собственный оператор Switch

Создать оператор `switch/case` сложнее, чем оператор `if/else`, зато он является более эффективным в случае множественного выбора. Следуйте правилам:

1. Условие заключите в скобки и откройте составной оператор (`{`).
2. Напишите `case` и условие, а потом двоеточие (`:`).
3. Напишите код, запускаемый в случае **совпадения**. Он может быть многострочным. Составной оператор тут не нужен.
4. Добавьте оператор `break` и точку с запятой (`;`).
5. По желанию создайте блок `default` для случая, когда совпадения отсутствуют.
6. Закройте составной оператор (`}`).

Интересно, а можно ли использовать оператор switch/case в «Приключениях»?

Часто Задаваемые Вопросы

В: То есть оператор `switch/case` не принимает решений на основании выражений вида `true/false`?

О: Да, в отличие от операторов `if` и `if/else`, `switch/case` принимает решение на основе тестовых данных. Именно это позволяет выбирать из множества вариантов.

В: То есть нам всего лишь нужно совпадение с тестовыми данными?

О: Да. В качестве тестовых данных используется переменная, со значением которой и идет сравнение.

В: Что произойдет при отсутствии ключевого слова `break`?

О: Оператор `break` является разделителем между блоками кода оператора `switch/case`. Без них все фрагменты кода были бы запущены просто в порядке очереди, что не имеет никакого смысла, так как выбор при этом не делается. В случае же совпадения запускается код после соответствующего оператора `case` вплоть до оператора `break`. После этого `switch/case` завершает свою работу.

Будьте осторожны!

Break для безопасности.

Этот оператор устраняет возможность случайного запуска ненужных фрагментов кода.

ОПЕРАТОР SWITCH О СЕБЕ

Интервью недели: Мастер на все руки

Head First: Спасибо, что согласились побеседовать с нами. Пожалуйста, опишите себя одним словом.

Switch: Разборчивый.

Head First: Уточните, пожалуйста.

Switch: Я предоставляю возможность выбирать между множеством вещей. Иногда достаточно выбрать между черным и белым, но иногда следует учитывать нюансы. Вот тут-то и появляюсь я.

Head First: Но говорят, что оператор If умеет делать то же самое, иногда с меньшим количеством кода.

Switch: Может быть и так. Можно и отрубить кусок дерева молотком, если бить по нему достаточно долго. Но лично я предпочитаю пилу. Пусть каждый занимается своим делом. Ничего не имею против оператора If, но этот инструмент предназначен для другой работы.

Head First: Вы говорили об эффективности. Каким образом она влияет на вашу работу?

Switch: Я принимаю решение на основе значения фрагмента данных. Произвожу сравнение и определяю, какой код запустить. Я не вычисляю значения выражений и не требую вложенности. Я всего лишь быстро принимаю решение.

Head First: Расскажите о вашем друге, операторе Break. Говорят, вы неразлучны?

Switch: Да, без оператора Break я не смог бы разделить фрагменты кода. Он дает мне понять, когда следует остановить выполнение выбранного блока, и я прекращаю работу, не трогая остальные блоки.

Head First: А что насчет оператора Case?

Switch: С оператором Case у меня очень близкие отношения, ведь именно он показывает мне варианты, с которыми следует сравнивать тестовые данные. Без него я не смог бы принимать решения.

Head First: То есть оператор Case предоставляет вам варианты, а вы на их основе определяете, что делать. А что происходит, когда совпадений нет?

Switch: Если на этот случай не добавлен отдельный код, то ничего. Но мой добрый друг Default дает возможность добавить такой код. Он и запускается при отсутствии совпадений.

Head First: Я и не знал. А как Default уживается с Case?

Switch: Прекрасно. Им нечего делить, каждый просто выполняет свою работу. Case обрабатывает совпадающие результаты, в то время как Default заботится о ситуации, когда совпадений не оказывается. Более того, мне кажется, что Case чувствует себя спокойней в присутствии Default, так как, если совпадений не обнаруживается, он начинает нервничать.

Head First: Понимаю. Что ж, наше время заканчивается. Хотите что-нибудь сказать напоследок?

Switch: Конечно. Помните, что нет ничего хуже нерешительности. Тюфяков никто не любит. Наличие слишком многих возможностей не означает, что нужно опускать руки. Звоните мне, и я помогу вам принять решение, которое лучше всего подходит для вашего сценария.

Возьми в руку карандаш

Перепишите первые две сцены «Приключений», заменив оператор `if/else` оператором `switch/case`.

```
...
if (curScene == 0) {
  curScene = 1;
  message = "Your journey begins at a fork in the road.";
}
else if (curScene == 1) {
  if (decision == 1) {
    curScene = 2
    message = "You have arrived at a cute little house in the woods.";
  }
  else {
    curScene = 3;
    message = "You are standing on the bridge overlooking a peaceful
stream.";
  }
}
...
```

Это исходная версия кода, в которой используется оператор if/else.

Возьми в руку карандаш

Решение

Вот как будут выглядеть первые две сцены «Приключений» после замены операторов `if/else` оператором `switch/case`.

```
...
if (curScene == 0) {
  curScene = 1;
  message = "Your journey begins at a fork in the road.";
}
else if (curScene == 1) {
  if (decision == 1) {
    curScene = 2
    message = "You have arrived at a cute little house in the woods.";
  }
  else {
    curScene = 3;
    message = "You are standing on the bridge overlooking a peaceful
stream.";
  }
}
...
```

Исходная версия кода.

Задает новый номер и описание сцены, как и в предыдущей версии.

```
switch (curScene) {
case 0:
  curScene = 1;
  message = "Your journey begins at a fork in the road.";
  break;
case 1:
  if (decision == 1) {
    curScene = 2
    message = "You have arrived at a cute little house in the woods.";
  }
  else {
    curScene = 3;
    message = "You are standing on the bridge overlooking a peaceful stream.";
  }
  break;
...
}
```

Оператор case предоставляет для сравнения номер сцены.

Внутри каждого оператора case имеет смысл воспользоваться оператором if/else для обработки решения пользователя о переходе к следующей сцене.

Для остальных сцен используется аналогичная структура.

Закроем оператор switch/case при помощи }.

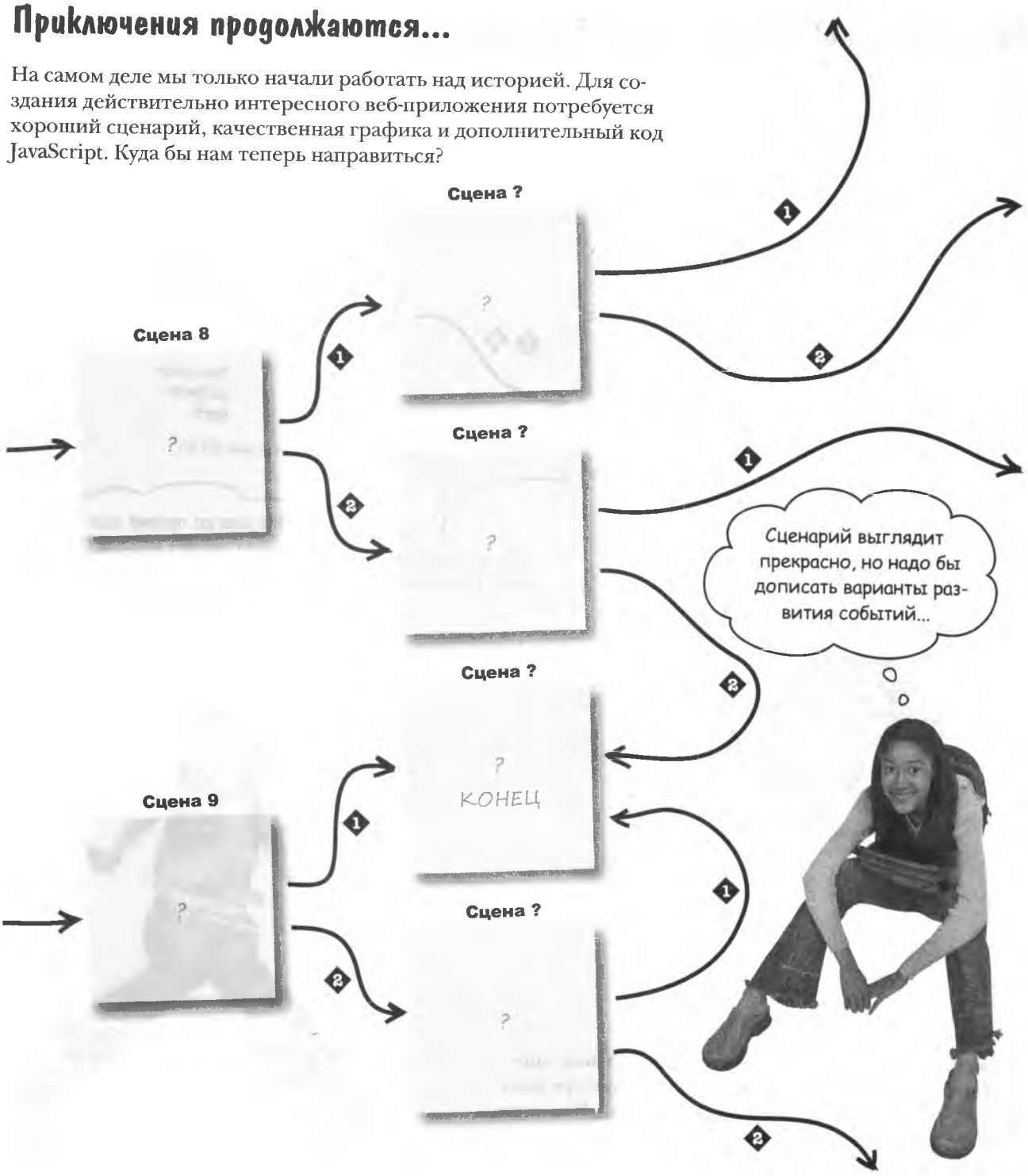
Тест-драйв нового варианта «Приключений»

Теперь, когда в основе принятия решения в «Приключениях» лежит совсем другая логика, Элли не терпится посмотреть на результат. Наверное, изменения будут заметны невооруженным глазом...

Удивлены, что ничего не изменилось! Но ведь оператор `switch/case` поменял только структуру кода наших «Приключений», а не внешний вид. Как видите, иногда изменения происходят за вашей спиной... в буквальном смысле слова!

Приключения продолжаются...

На самом деле мы только начали работать над историей. Для создания действительно интересного веб-приложения потребуется хороший сценарий, качественная графика и дополнительный код JavaScript. Куда бы нам теперь направиться?

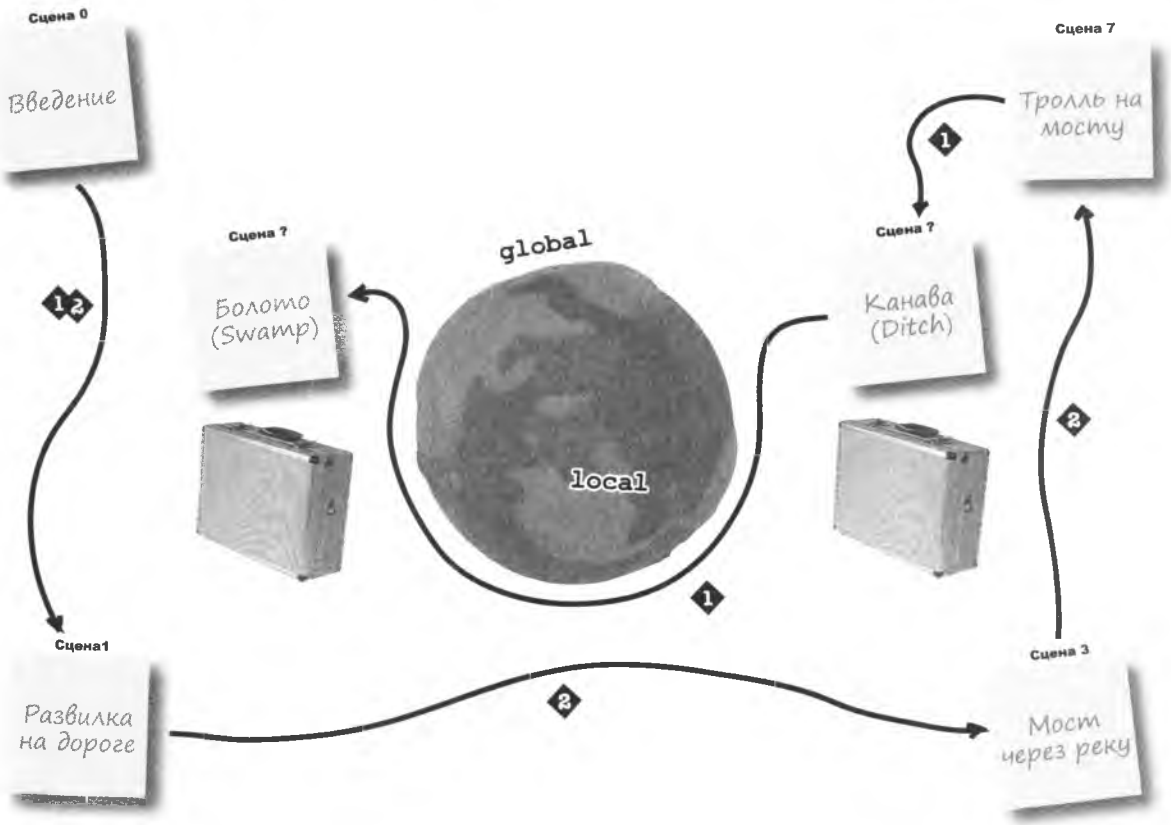


Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Когда не хватает if / else...

Ум хорошо, а два лучше!



Хотя оператор if / else крайне полезен, он имеет ограничения. Например, невозможно выбрать из множества вариантов. Если вы не верите, убедитесь сами.

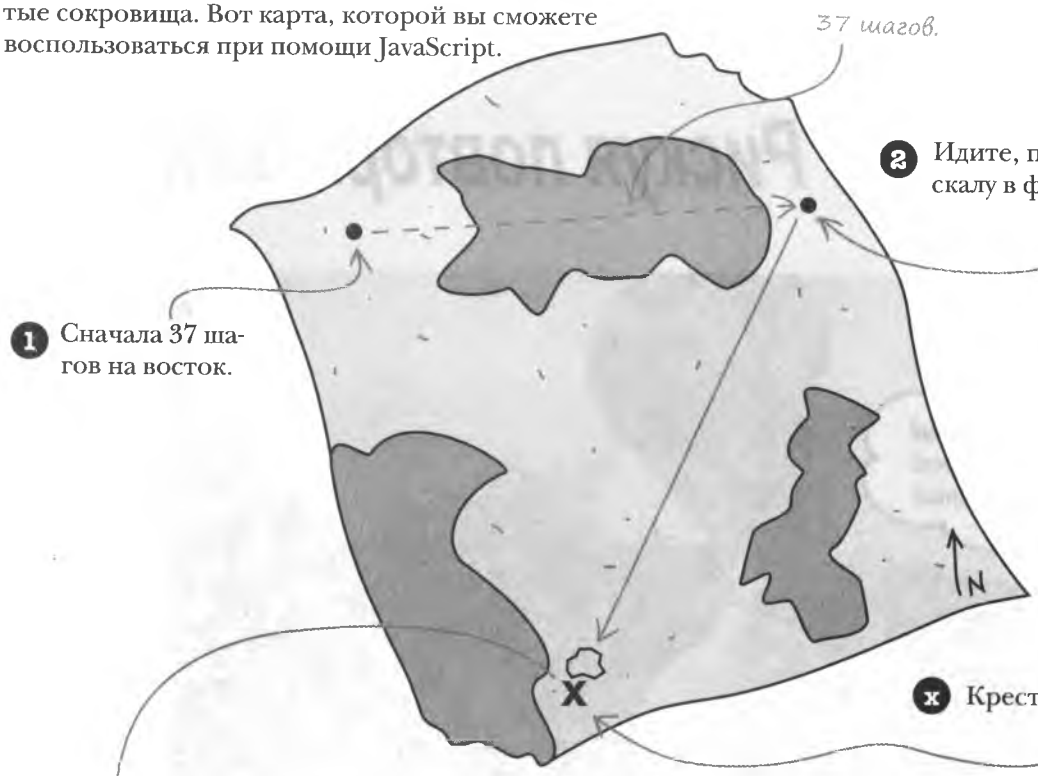
5 Циклы

Рискуя повториться

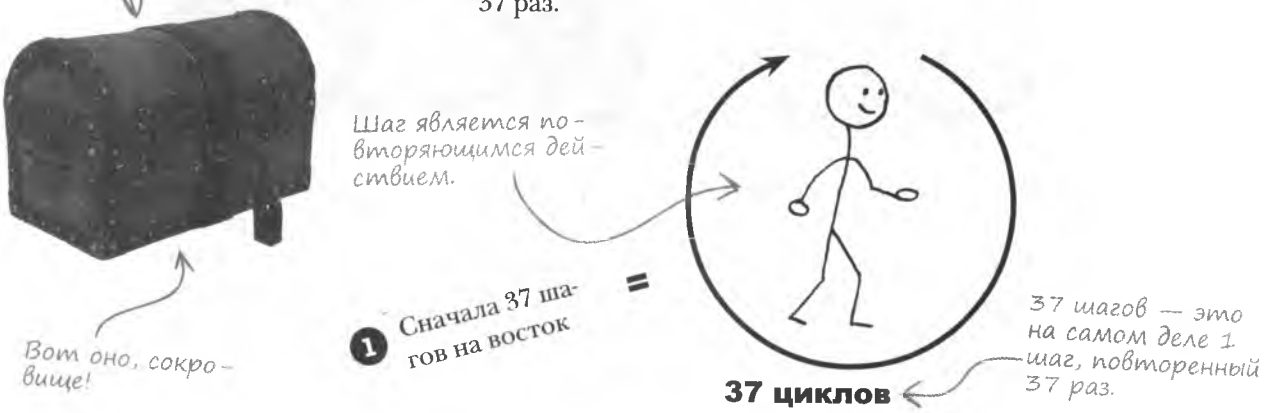
Говорят, что повторение — мать учения. Заниматься новыми и интересными делами здорово, но наши дни, как правило, состоят из рутины. Доведенное до автоматизма мытье рук, нервный тик, щелчок на кнопке Reply To All при получении любого дурацкого сообщения! Кажется, повторение не самая лучшая вещь в этом мире. А вот в мире JavaScript без него никак. Вы удивитесь, как часто бывают востребованы одни и те же фрагменты кода. Здесь вам на помощь приходят циклы. Без них пришлось бы снова и снова набирать один и тот же код.

Место помечено крестом

Самое соблазнительное в мире – это чужие зарытые сокровища. Вот карта, которой вы сможете воспользоваться при помощи JavaScript.



Сначала вам нужно повторить действие (шаг) указанное количество раз (37). Сделать 37 шагов – означает повторить один шаг 37 раз.



Осталось понять, каким образом в JavaScript реализуются повторения.

И снова дежавю... цикл for

Повторение в JavaScript реализуется при помощи циклов. Например, цикл `for` позволяет повторить фрагмент кода нужное количество раз. Он прекрасно подходит для подсчета количества повторяющихся действий.

Состоит этот цикл из четырех частей:

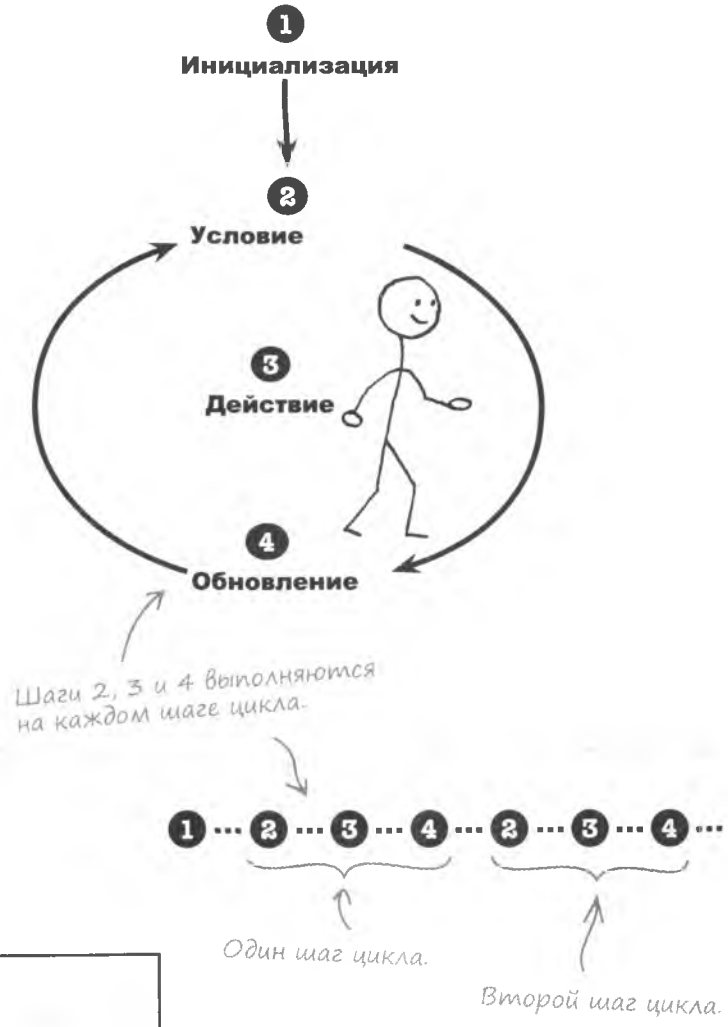
Цикл for позволяет повторить код нужное количество раз.

1. **Инициализация**
   Имеет место один раз в начале цикла `for`.

2. **Проверка условия**
   Условие показывает, следует ли продолжать выполнение цикла.

3. **Действие**
   Собственно код, который повторяется в процессе работы цикла.

4. **Обновление**
   В этой части обновляются значения всех переменных цикла.



МОЗГОВОЙ ШТУРМ

Каким образом четыре стадии цикла `for` помогут нам в поисках сокровищ?

Охота за сокровищами с циклом for

Цикл `for` подходит для работы с картой, так как включает в себя известное количество шагов. Его применение к первой части поиска будет выглядеть примерно так:

```
for (var x = 0; x < 37; x++)
  takeStep();
```

1 2 4 3

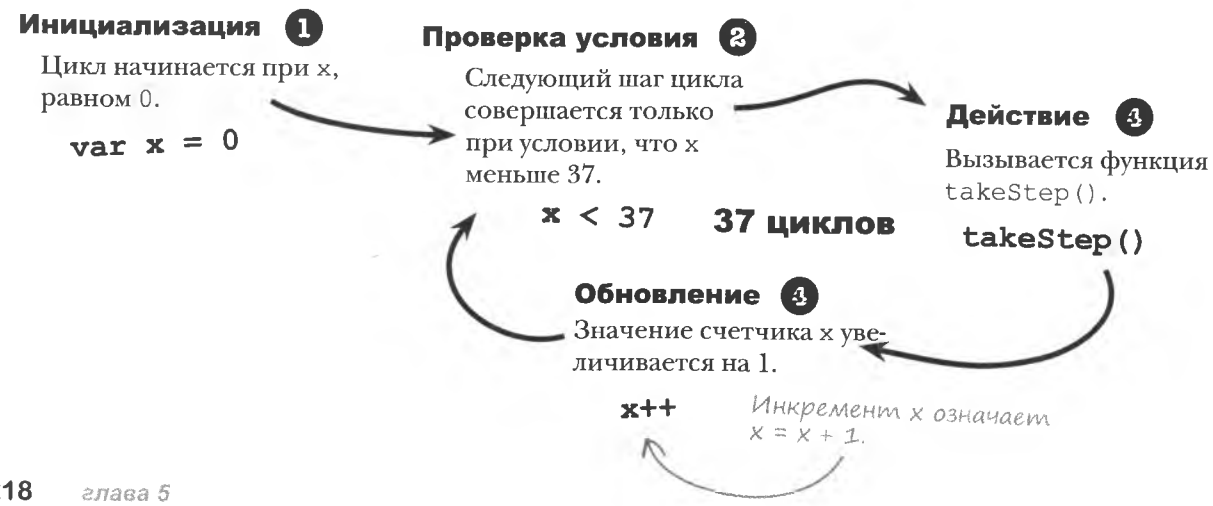
Инкремент x — это то же самое, что и x = x + 1.

В JavaScript отсчет работы цикла начинается с 0, хотя его можно начать и с 1.

Вот анализ кода цикла `for`:

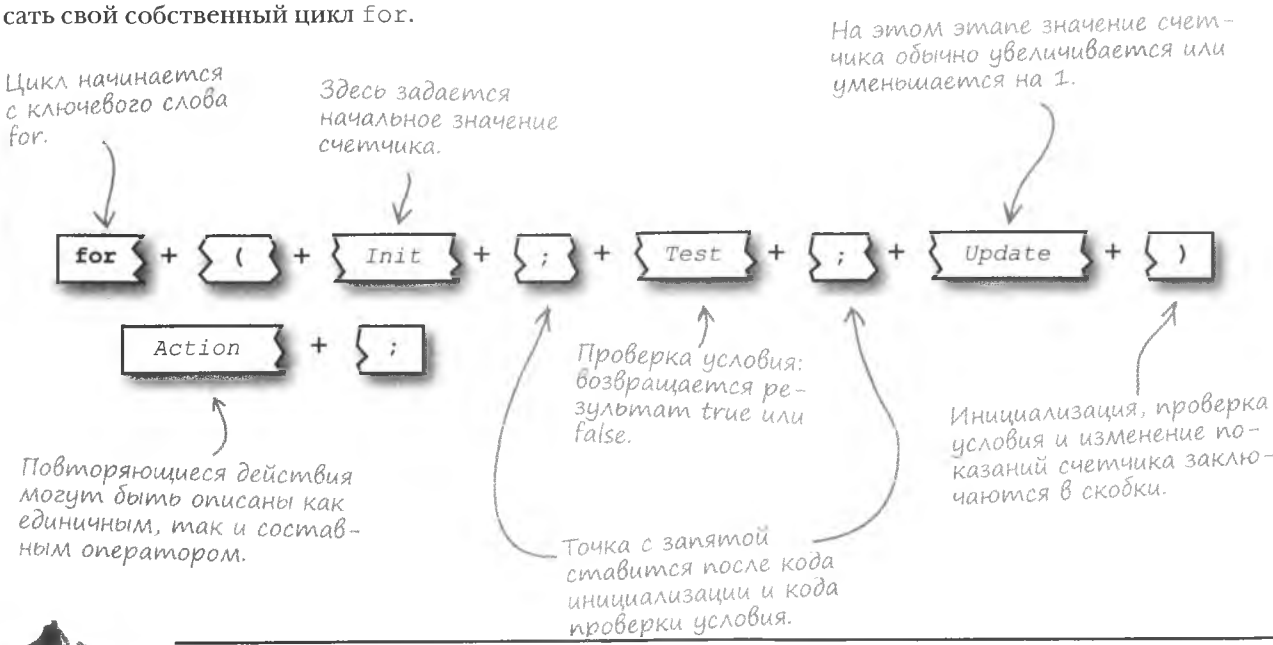
1. Переменной x присвоим начальное значение 0.
2. Проверим, меньше ли x, чем 37. Если да, перейдем к шагу 3 и продолжим цикл. Если нет, выйдем из цикла.
3. Запустим код цикла. В данном случае это функция `takeStep()`.
4. Увеличим значение x на 1 и вернемся ко второму шагу.

После 37 шагов x станет равен 37, и цикл завершится. Вот каким образом в JavaScript реализуются повторяющиеся действия.



Составные части цикла for

Каждый компонент цикла `for` должен находиться на предназначенном ему месте. Впрочем, существует множество возможностей написать свой собственный цикл `for`.



Упражнение

Завершите код, который сначала предлагает пользователю ввести число больше 0, а затем использует его в качестве начального значения счетчика цикла `for`, выполняющего обратный отсчет до начала фильма (4, 3, 2, 1, Roll film!). Перед началом обратного отсчета не забудьте удостовериться, что введенное значение действительно больше 0.

Сохраните введенное число в переменной count.

Пользователю предлагается ввести число.

```
var count = prompt("Enter a number greater than 0:", "10");
```

Упражнение Решение

Вот как выглядит код, который сначала предлагает пользователю ввести положительное число, а затем использует его в качестве начального значения счетчика цикла `for`, выполняющего обратный отсчет до начала фильма (4, 3, 2, 1, Roll film!).

Число сохраняется в переменной count.

Пользователю предлагается ввести число.

Обратный отсчет до 1.

Убедимся, что переменная count больше 0.

На каждом шаге цикла значение счетчика уменьшается на 1.

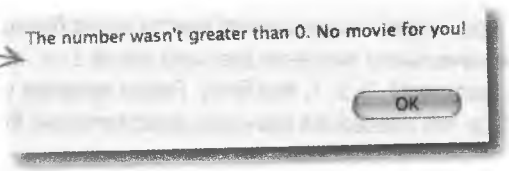
Присвоим счетчику цикла (x) значение переменной count.

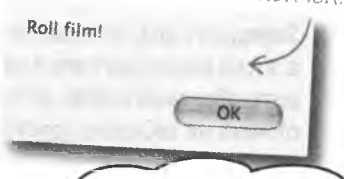
Текущее значение переменной count.

```
var count = prompt("Enter a number greater than 0:", "10");
if (count > 0) {
  for (var x = count; x > 0; x--)
    alert("Starting in..." + x);
  alert("Roll film!");
}
else
  alert("The number wasn't greater than 0. No movie for you!");
```

Некорректные данные.

Отсчет закончен!





Специальные места для мачо

Обратный отсчет до начала сеанса — это не единственный эпизод, в котором циклы могут быть использованы в кинотеатре. Может быть, вы слышали, что мачо требуют, чтобы между ними было пустое кресло. Поэтому Сет и Джейсон решили создать программу Mandango, осуществляющую поиск пустых кресел в кинотеатре.

Друзьям требуются группы из трех мест, чтобы между ними всегда оставалось свободное кресло. Но пока они еще не знают, как подойти к решению данной проблемы.

Проверка доступности мест

Парням нужно проверять каждый ряд на наличие расположенных подряд трех свободных кресел.

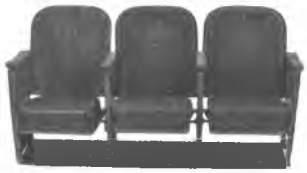
Свободно всего одно место.

Все три места заняты.

Плохо

Все варианты, отличные от трех стоящих подряд свободных кресел, не позволяют показать «мачизм».

Все три места свободны!



Классно!

Три свободных кресла, стоящих подряд, дают настоящему мачо нужное пространство.

Возьми в руку карандаш

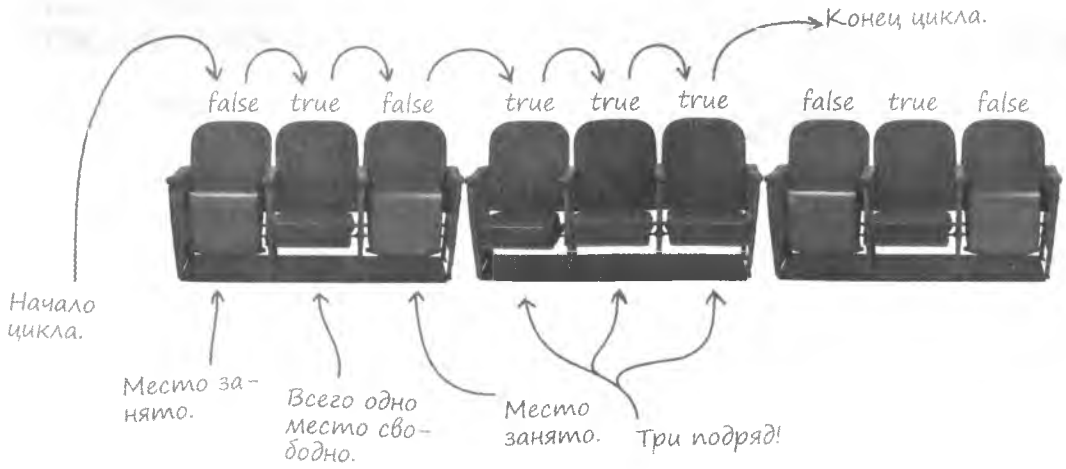
Взяв за основу рисунок снизу, покажите, каким образом при помощи цикла `for` вы будете искать места для наших мачо.

Возьми в руку карандаш

Решение

Итак, вот каким образом при помощи цикла `for` можно найти подходящие места для мачо.

Представив доступность места в виде логической переменной, нужно найти при помощи цикла три подряд значения true.



Циклы, HTML и свободные кресла

На странице Mandango все кресла показаны в виде рисунков.

Основная идея программы Mandango понемногу становится понятной, но каким же образом записать ее в виде кода HTML?

```
<img id="seat1" src="seat_unavail.png" alt="Unavailable" />
<img id="seat2" src="seat_avail.png" alt="Available" />
<img id="seat3" src="seat_unavail.png" alt="Unavailable" />
<img id="seat4" src="seat_avail.png" alt="Available" />
<img id="seat5" src="seat_avail.png" alt="Available" />
<img id="seat6" src="seat_avail.png" alt="Available" />
<img id="seat7" src="seat_unavail.png" alt="Unavailable" />
<img id="seat8" src="seat_avail.png" alt="Available" />
<img id="seat9" src="seat_unavail.png" alt="Unavailable" />
```

HTML и графические фрагменты для этого примера можно скачать по адресу http://www.headfirstlabs.com/books/hfjs/.

Вам нужно не только просмотреть при помощи цикла графические фрагменты с изображением кресел, но и сохранить информацию о доступности мест в коде JavaScript.

Места, как переменные

Перед тем как приступить к выполнению цикла, нужно представить информацию о доступности каждого кресла в виде кода JavaScript. Для ряда из девяти кресел вам потребуется девять логических переменных.

```
var seat1 = false;
var seat2 = true;
var seat3 = false;
var seat4 = true;
var seat5 = true;
var seat6 = true;
var seat7 = false;
var seat8 = false;
var seat9 = false;
```

Информация о доступности каждого места хранится в виде переменной типа boolean.

True означает, что место свободно.

False означает, что место занято.

Получается, я должен просматривать различные фрагменты данных с помощью всего одной переменной?

Теперь все готово для создания цикла `for`, который будет искать три свободных места подряд.

```
for (var i = 0; i < 10; i++) {

  if (seat1)
    ...

}
```

Вы не можете менять имя переменной при каждом прогоне цикла!

Кажется, у нас проблема. Цикл `for` должен проверять значения различных переменных `seat` на каждом шаге. Но так как все переменные имеют разные имена, это попросту невозможно.

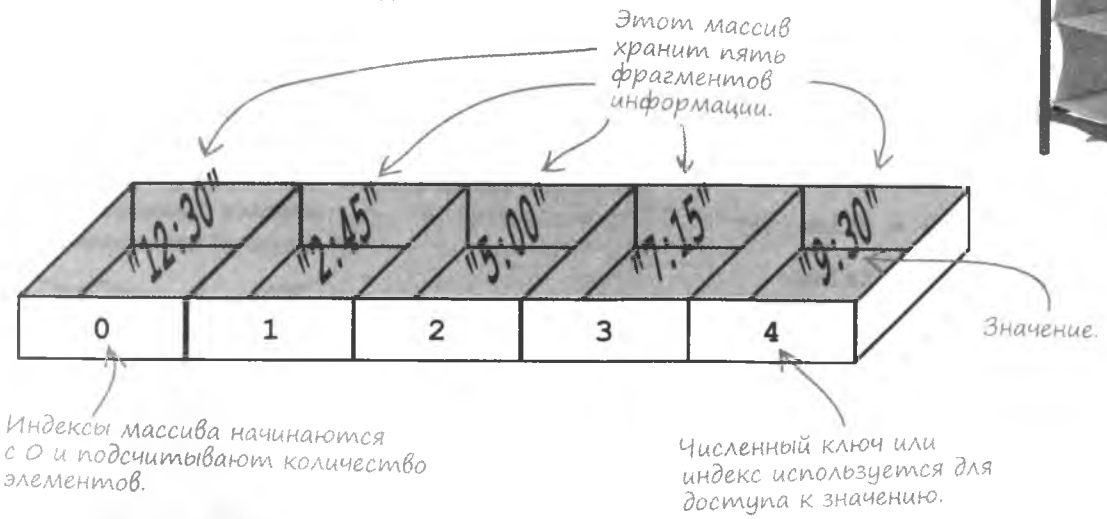
МОЗГОВОЙ ШТУРМ

Каким же образом сохранить информацию в том случае, когда отдельные переменные не работают?

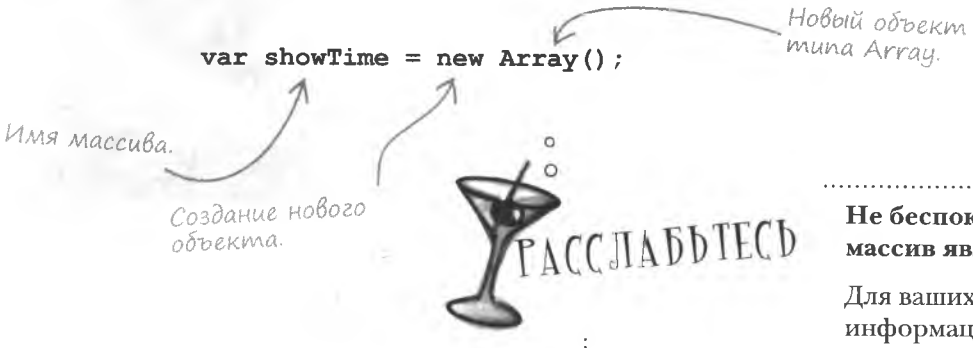
Массивы

JavaScript позволяет сохранять наборы данных в одной переменной особого типа, называемой массивом. Массив, как и обычная переменная, имеет всего одно имя, зато у него целый набор мест для хранения. Представьте себе шкаф с полками – это один предмет мебели, но с массой мест для хранения.

Каждый элемент массива состоит из двух частей: значение и уникальный ключ, по которому осуществляется доступ к этому значению. В роли ключей обычно выступают цифры, начиная с нуля. Такие ключи обычно называются индексами:



Для создания массива достаточно дать знать об этом JavaScript. По большому счету вы создаете объект.

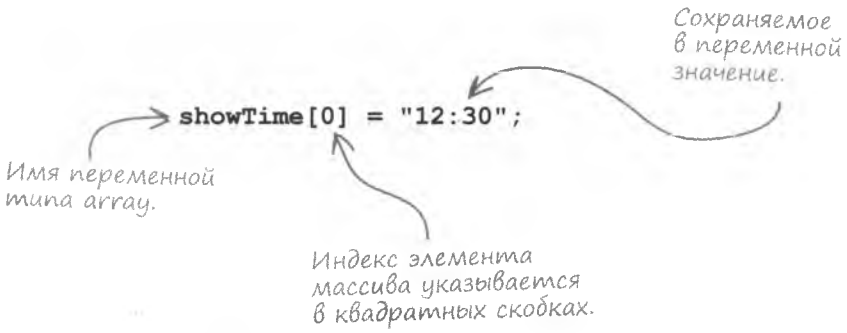


Не беспокойтесь о том, что массив является объектом.

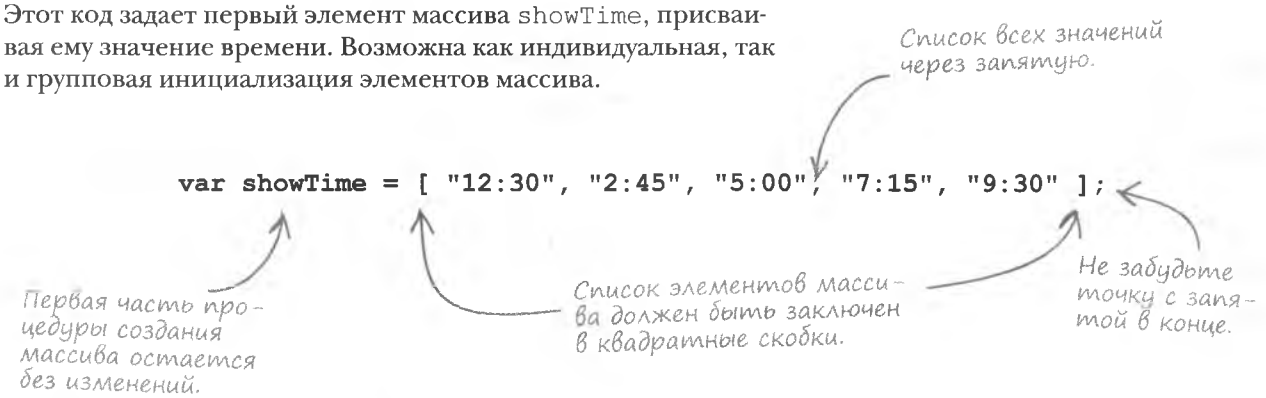
Для ваших текущих целей эта информация не имеет особого значения. Подробно мы поговорим об объектах в главах 9 и 10.

Значения сохраняются с ключами

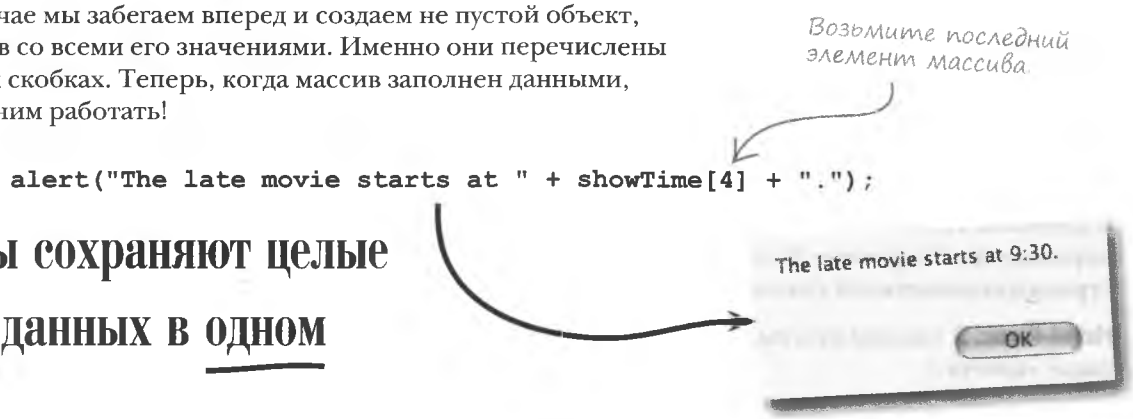
Массив готов, теперь в него можно добавлять данные. Для доступа к ним будут использоваться **ключи**. Каждый ключ уникален и связан со своим фрагментом данных.



Этот код задает первый элемент массива showTime, присваивая ему значение времени. Возможна как индивидуальная, так и групповая инициализация элементов массива.



Подождите, это не похоже на создание объекта. Что происходит? В данном случае мы забегаем вперед и создаем не пустой объект, а сразу массив со всеми его значениями. Именно они перечислены в квадратных скобках. Теперь, когда массив заполнен данными, мы готовы с ним работать!



Массивы сохраняют целые наборы данных в одном месте.

МАССИВ О СЕБЕ

Интервью недели: Секреты хранителя данных

Head First: Рады видеть вас, Массив. Слышал, что вы умеете сохранять наборы данных.

Массив: Да. Если вам нужно сохранить 50 строк текста или 300 чисел, обращайтесь.

Head First: Звучит заманчиво. Но ведь данные можно сохранить и в обычной переменной.

Массив: Разумеется. А на работу можно ходить босиком. Видишь ли, некоторые вещи можно делать разными способами. В данном случае более удобный способ хранения наборов данных представляю именно я.

Head First: Да, я предпочитаю ходить на работу в ботинках. Но чем именно вы лучше других?

Массив: Представь, что ты ведешь дневник. Каждый день. Что со всеми этими страницами станет через несколько лет?

Head First: А что с ними может случиться?

Массив: Ты сейчас представляешь страницы, каким-то способом соединенные друг с другом. А если представить обычные листы бумаги, складываемые в коробку? Рано или поздно, тебе станет сложно их систематизировать.

Head First: Ты намекаешь, что сохранение данных в массиве — это то же самое, что и соединение отдельных листов в книгу?

Массив: Именно. Ведь я систематизирую данные, обеспечивая к ним легкий доступ. В дневнике ты можешь найти нужную запись на определенной странице. В случае массива вместо страниц используются ключи.

Head First: Я слышал про индексы массива. А что такое «ключи»?

Массив: Ключом называется фрагмент информации, при помощи которого легко найти нужные данные. Индекс — это всего лишь частный случай ключа. Таким образом, номера страниц в дневнике являются не только ключами, но еще и индексами.

Head First: Я понял. А каким образом все это связано с циклами?

Массив: В принципе я могу сохранять данные и без участия циклов. Но их помощь упрощает доступ к моим элементам.

Head First: Каким образом?

Массив: Помните, что циклы используют численный счетчик? Он может служить и в качестве индекса, в итоге облегчается циклический просмотр хранимых во мне данных.

Head First: То есть можно использовать счетчик цикла в качестве индекса массива?

Массив: Именно так.

Head First: Это потрясающе!

Массив: Я знаю. Именно поэтому меня так любят сценарии, в которых необходим циклический просмотр данных. Несколько строчек кода, и вы легко просматриваете целый массив.

Head First: Могу вообразить. Спасибо, что рассказали о себе и о вашей совместной работе с циклами.

Массив: Не за что. Всегда рад помочь!

Возьми в руку карандаш

Напишите код создания массива `seats` для программы Mandango, а затем просмотрите в цикле его элементы, выводя для пользователя информацию о доступности каждого из кресел.

Часто Задаваемые Вопросы

В: Возможен ли бесконечный цикл `for`?

О: Страшный бесконечный цикл? Конечно, вы можете запрограммировать цикл, который не закончится, пока вы не перезагрузите страницу. Но такие циклы считаются вредной вещью, так как не дают сценарию перейти к следующей задаче, — их можно считать эквивалентом заблокированного приложения.

Бесконечный цикл возникает или при некорректном обновлении счетчика шагов, или в случае, когда он не меняется и результатом проверки условия всегда является значение `false`. Поэтому всегда следует тщательно проверять условие выхода из цикла `for`. Понять, что у вас бесконечный цикл, можно по тем признакам, что сценарий долгое время не выполняет никаких действий.

В: Можно ли использовать в цикле `for` составные операторы?

О: Конечно! Более того, во всех случаях, кроме самых простых сценариев, вам придется их использовать.

В: Когда результатом проверки условия становится значение `false`, выполняется ли цикл в последний раз?

О: Нет. Операторы в цикле `for` выполняются, только когда проверка условия дает результат `true`. В противном случае происходит немедленный выход из цикла и никакой код не запускается.

В: Первым индексом массива всегда является 0?

О: И да, и нет. По умолчанию нумерация всех числовых массивов начинается с 0. Но это можно переопределить, начав нумерацию с любого другого числа, хотя это и не совсем удобно. Поэтому если у вас нет на то веской причины, не начинайте нумерацию элементов массива с чисел, отличных от нуля.

В: Сохраняемые в массивах данные должны принадлежать к одному типу?

О: Вовсе нет. Все зависит от предназначения массива. Например, если вы хотите циклически просмотреть массив оценок и вычислить среднюю, вы не сможете этого сделать, если часть значений будет принадлежать, например, к логическому типу. Поэтому, несмотря на наличие принципиальной возможности сохранять в массив данные различных типов, лучше этого не делать.

Возьми в руку карандаш

Решение

Итак, вот код массива `seats` для программы Mandango, который при помощи цикла проверяет места в кинотеатре и сообщает об их доступности пользователю.

Так как индексы массива начинаются с 0, начнем с 0 и счетчик цикла.

Создаем массив seats и присваиваем его элементам логические значения.

Значения массива указываются через запятую.

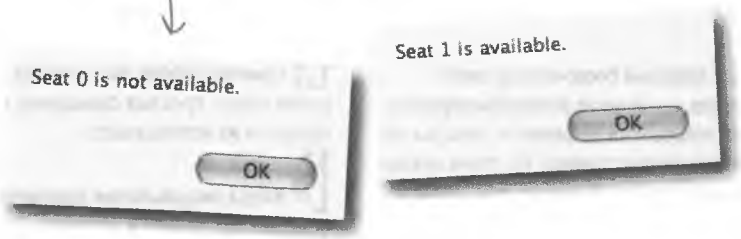
Увеличиваем значение счетчика на единицу.

Хотя в данном случае можно было обойтись числом 9, лучше использовать свойство length на случай, если длина массива потом изменится.

```
var seats = [ false, true, false, true, true, true, false, true, false ];
for (var i = 0; i < seats.length; i++) {
  if (seats[i])
    alert("Seat " + i + " is available.");
  else
    alert("Seat " + i + " is not available.");
}
```

Счетчик цикла используется в качестве индекса массива и по очереди дает доступ ко всем его элементам.

В зависимости от того, доступно место (true) или недоступно (false), будут появляться разные окна диалога.



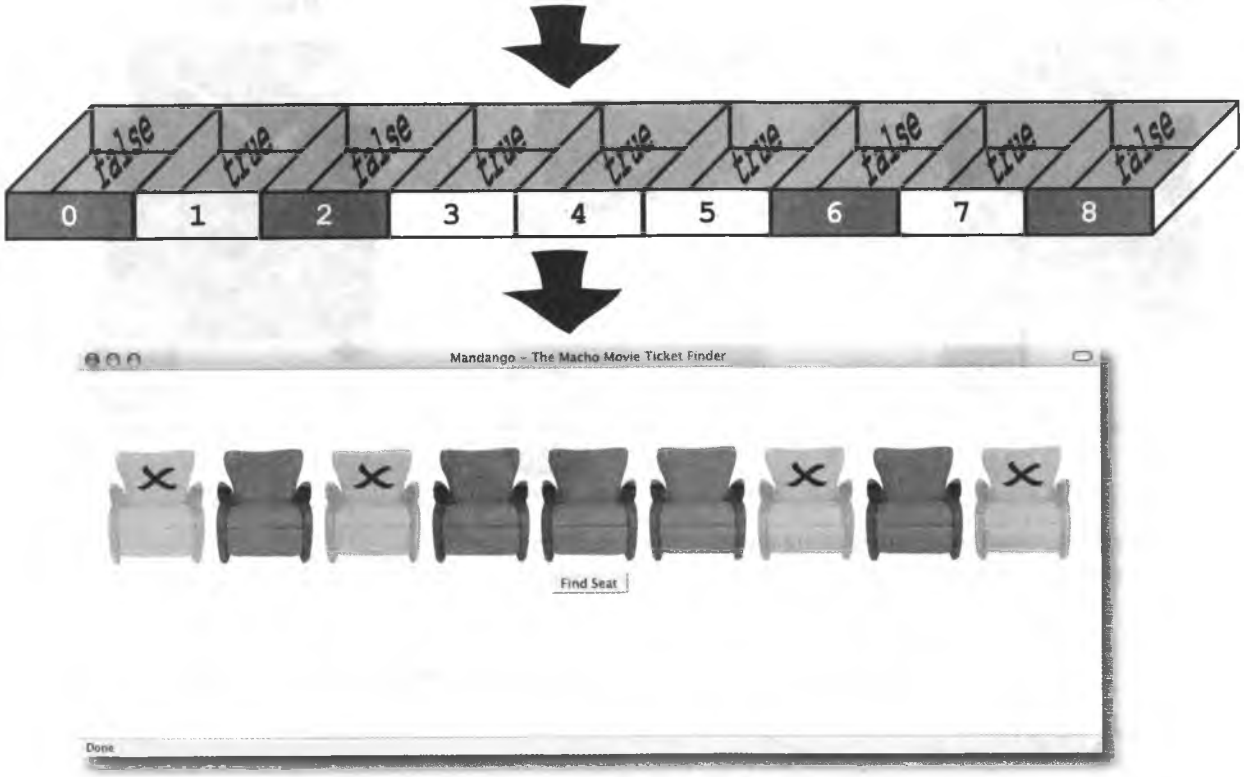
КЛЮЧЕВЫЕ МОМЕНТЫ

- Циклы `for` повторяют фрагменты кода JavaScript **заданное число раз**.
- Операторы инкремента (`++`) и декремента (`--`) позволяют легко менять показания счетчика цикла.
- Массив — это способ хранения наборов данных в **одном месте**.
- Массив хранит набор фрагментов данных, но при этом имеет всего одно имя.
- Доступ к элементам числового массива осуществляется с помощью **индексов**.
- Счетчик цикла удобно использовать для доступа к данным числового массива.

От JavaScript к HTML

На данный момент доступность кресел представлена в виде массива логических значений. Теперь нам нужно преобразовать этот массив в набор изображений HTML (их можно скачать по адресу *http://www.headfirstlabs.com/books/hfjs/*), которые будут показывать свободные кресла на веб-странице Mandango.

```
var seats = [ false, true, false, true, true, true, false, true, false ];
```

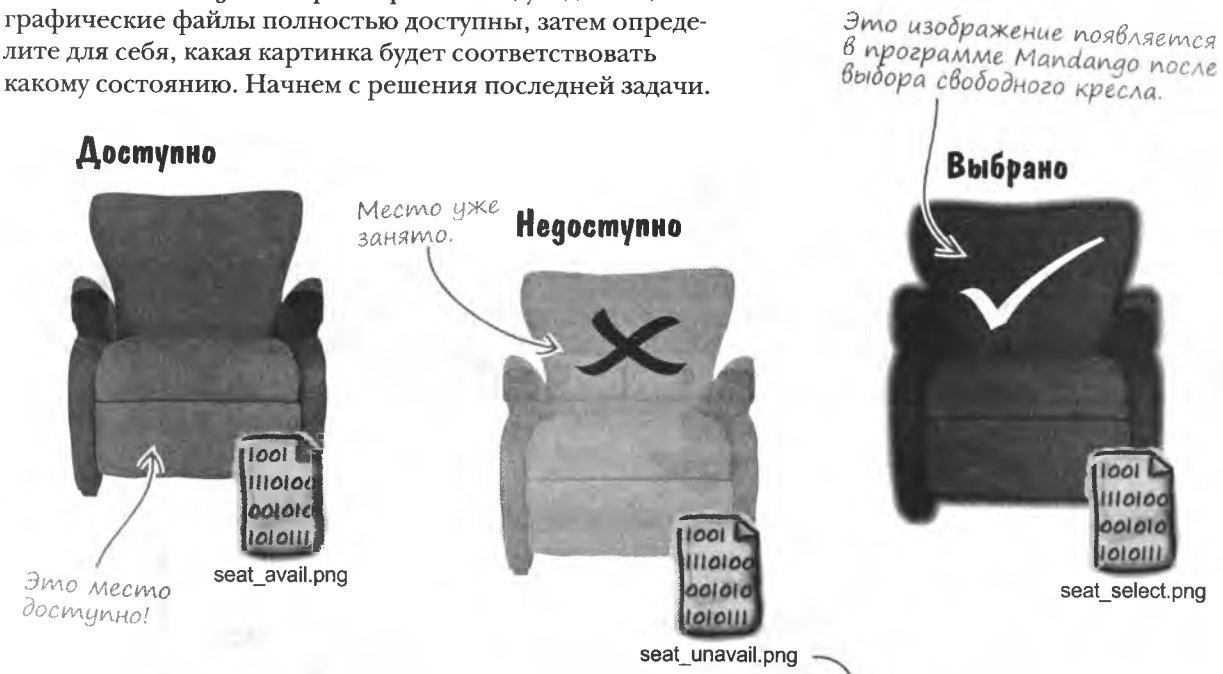


Выглядит все прекрасно, но проблема в том, что у нас нет кода, переводящего массив логических элементов в визуальные элементы на веб-странице.

Каким образом вы реализовали бы связь между массивом логических элементов JavaScript и картинками на веб-странице?

Визуализация кресел

Связывая массив JavaScript с картинками, убедитесь, что графические файлы полностью доступны, затем определите для себя, какая картинка будет соответствовать какому состоянию. Начнем с решения последней задачи.



Соответствующие графические файлы назначены атрибуту `src` в HTML-коде изображений:

```
<img id="seat8" src="seat_unavail.png" alt="Unavailable" />
```

Параметр ID определяет соответствие между элементами массива и изображениями — он должен начинаться с 0 и заканчиваться 8, как и индексы массива.

Теперь вам нужно в цикле просмотреть массив логических элементов, сопоставив изображение с каждым HTML-тегом `<img>` на странице. Фактически вы будете использовать уже знакомый вам цикл просто с другим набором действий:

1. Присвойте счетчику цикла `i` значение `0`.
2. Удостоверьтесь, что `i` не превышает 9. Если это так, перейдите к шагу 3, в противном случае выйдите из цикла.
3. Запустите код, **задающий изображение кресла**.
4. Увеличьте значение `i` на 1 и вернитесь к шагу 2.

Подробно про функцию initSeats()

Инициализация мест в программе Mandango осуществляется при помощи функции `initSeats()`, циклически сопоставляющей массив JavaScript с изображениями кресел.

Инициализация, выполняемая функцией initSeats(), присваивает изображения различным креслам, что отличается от инициализации в цикле for.

Счетчик цикла начинается с 0, поскольку именно таким является первый индекс массива.

Проверяет, все ли места были просмотрены.

Увеличение счетчика цикла на 1.

```
function initSeats() {
  // Задаем вид кресел
  for (var i = 0; i < seats.length; i++) {
    if (seats[i]) {
      // Вид доступных кресел
      document.getElementById("seat" + i).src = "seat_avail.png";
      document.getElementById("seat" + i).alt = "Available seat";
    }
    else {
      // Вид занятых кресел
      document.getElementById("seat" + i).src = "seat_unavail.png";
      document.getElementById("seat" + i).alt = "Unavailable seat";
    }
  }
}
```

Если переменная seat имеет значение true, помечаем кресло как доступное.

В качестве параметра ID кресла выступает счетчик цикла.

Если переменная seat имеет значение false, помечаем кресло как занятое.

```
<body onload="initSeats();">
  <div style="margin-top:75px; text-align:center">
    <img id="seat0" src="" alt="" />
    <img id="seat1" src="" alt="" />
    <img id="seat2" src="" alt="" />
    <img id="seat3" src="" alt="" />
    <img id="seat4" src="" alt="" />
    <img id="seat5" src="" alt="" />
    <img id="seat6" src="" alt="" />
    <img id="seat7" src="" alt="" />
    <img id="seat8" src="" alt="" /><br />
  </div>
</body>
</html>
```

Атрибуты src и alt динамически редактируются для каждого кресла.

id этого изображения "seat6".

Поиск любых свободных кресел

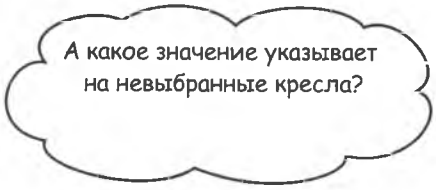
Теперь, когда инициализация завершена, можно перейти к поиску, ради которого, собственно, и была задумана программа Mandango. Сет и Джейсон решили, что сначала имеет смысл написать алгоритм поиска одного свободного места. В итоге мы подойдем к решению сложной задачи постепенно, решая более простые варианты.

Для поиска свободного места нам потребуется переменная, отслеживающая результат выбора.

Если это глобальная переменная, то она будет доступна для всего сценария.

selSeat

Переменная, хранящая информацию о выбранном кресле, будет нужна нам на протяжении всей жизни сценария, поэтому она должна быть глобальной. Присвоим ей имя `selSeat`. Именно в эту переменную функция `indSeat()` будет сохранять индекс выбранного кресла.



Сет задал хороший вопрос. Переменная `selSeat` хранит информацию о выбранном кресле, то есть число от 0 до 8. А как быть в тех случаях, когда пользователь пока **не** выбрал ни одного варианта. Для этого нам потребуется специальное значение. Пусть это будет 1. Именно такое начальное значение должна иметь переменная `selSeat`.

Переменной selSeat присвоено начальное значение –1. Ведь сценарий начинает работу, когда ни одного места еще не выбрано.

```
var selSeat = -1;
```

Итак, работа над переменной выбора места окончена, и все готово для написания функции `findSeat()`. Эта функция будет просматривать все места, находить незанятые и спрашивать пользователя, не хочет ли он забронировать именно это место. Разумеется, для целей настоящих мачо эта программа пока не подходит, но мы движемся в нужном направлении!

Магниты JavaScript

Функция `findSeat()` должна искать незанятые кресла и предлагать их пользователю, который может как подтвердить бронь, так и вернуться к дальнейшему поиску. Воспользуйтесь магнитами, чтобы завершить код.

```
function findSeat() {
  // Если место уже выбрано, произведите повторную инициализацию

  if (.....>= 0) {

    ..... = -1;

    ..... ();
  }

  // Поиск свободного места среди всех возможных
  for (var i = 0; i < seats.length; i++) {
    // Проверка доступности рассматриваемого в данный момент кресла

    if (.....) {
      // Выделяет кресло и обновляет его вид

      ..... = i;

      document.getElementById("seat" + i)...... = "seat_select.png";

      document.getElementById("seat" + i).....= "Ваше место";

      // Пользователю предлагается принять предложенный вариант

      var ..... = confirm("Seat " + (i + 1) + " is available. Accept?");

      if (.....) {
        // Пользователь отказался, продолжаем поиск

        ..... = -1;

        document.getElementById("seat" + i).....= "seat_avail.png";

        document.getElementById("seat" + i).....= "Свободное место";
      }
    }
  }
}
```

Решение задачи с магнитами

Итак, вот как должна выглядеть функция `findSeat()`, ищущая свободное место и предлагающая им воспользоваться.

```
function findSeat() {
  // Если место уже выбрано, произведите повторную инициализацию
  if ( selSeat >= 0) {
    selSeat = -1;
    initSeats();
  }

  // Поиск свободного места среди всех возможных
  for (var i = 0; i < seats.length; i++) {
    // Проверка доступности рассматриваемого в данный момент кресла
    if ( seats[i] ) {
      // Выделяет кресло и обновляет его вид
      selSeat = i;
      document.getElementById("seat" + i).src = "seat_select.png";
      document.getElementById("seat" + i).alt = "Your seat";
      // Пользователю предлагается принять предложенный вариант
      var accept = confirm("Seat " + (i + 1) + " is available. Accept?");
      if ( !accept ) {
        // Пользователь отказался, продолжаем поиск
        selSeat = -1;
        document.getElementById("seat" + i).src = "seat_avail.png";
        document.getElementById("seat" + i).alt = "Available seat";
      }
    }
  }
}
```

Если значение переменной selSeat отлично от –1, сбросьте информацию обо всех местах и начните поиск заново.

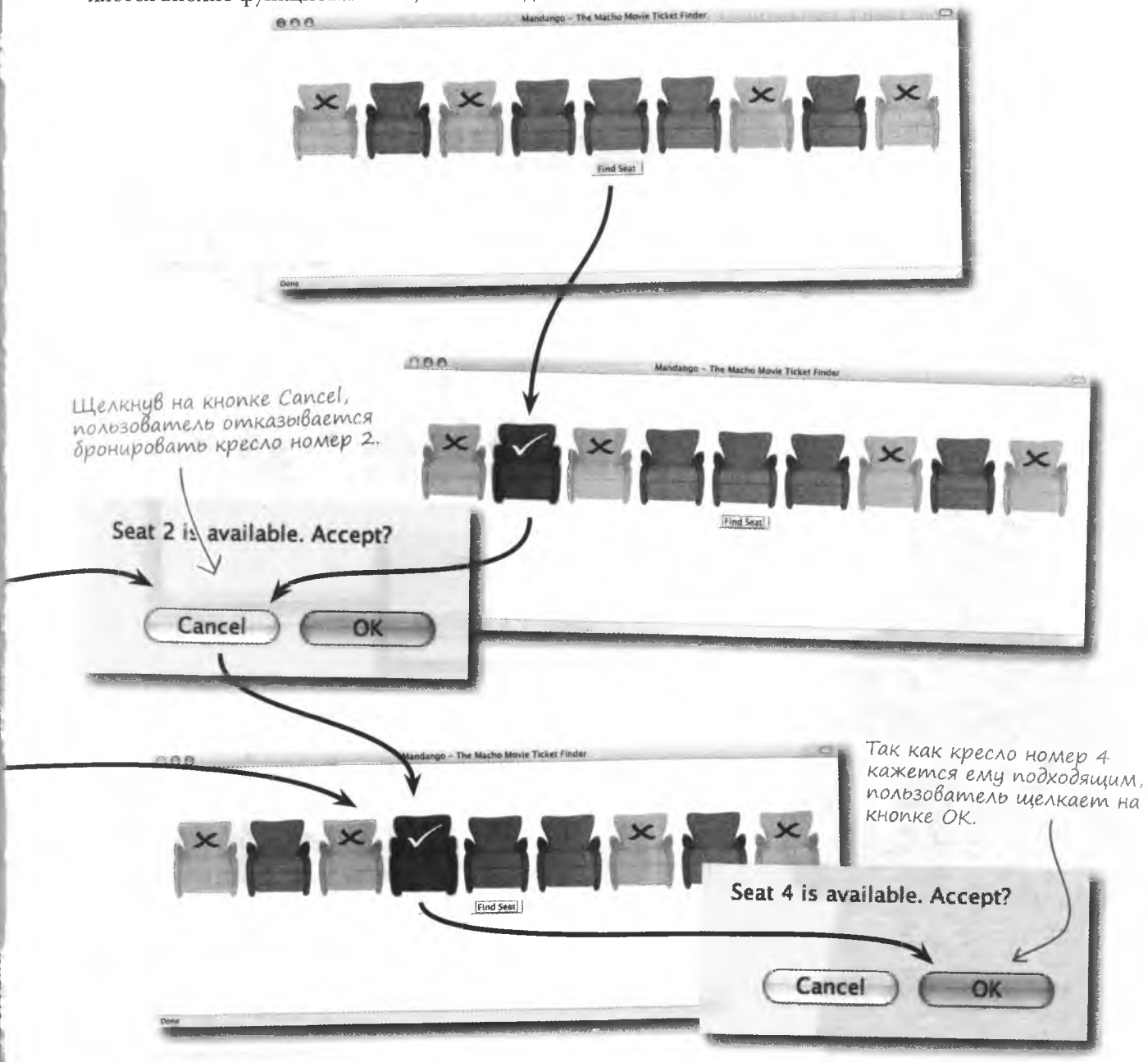
Если место свободно, seats[i] имеет значение true.

Показываемые пользователю номера мест на единицу больше реальных, так как в кинотеатрах нумерация начинается с 1, а не с 0.

Проверяем, согласился ли пользователь занять место.

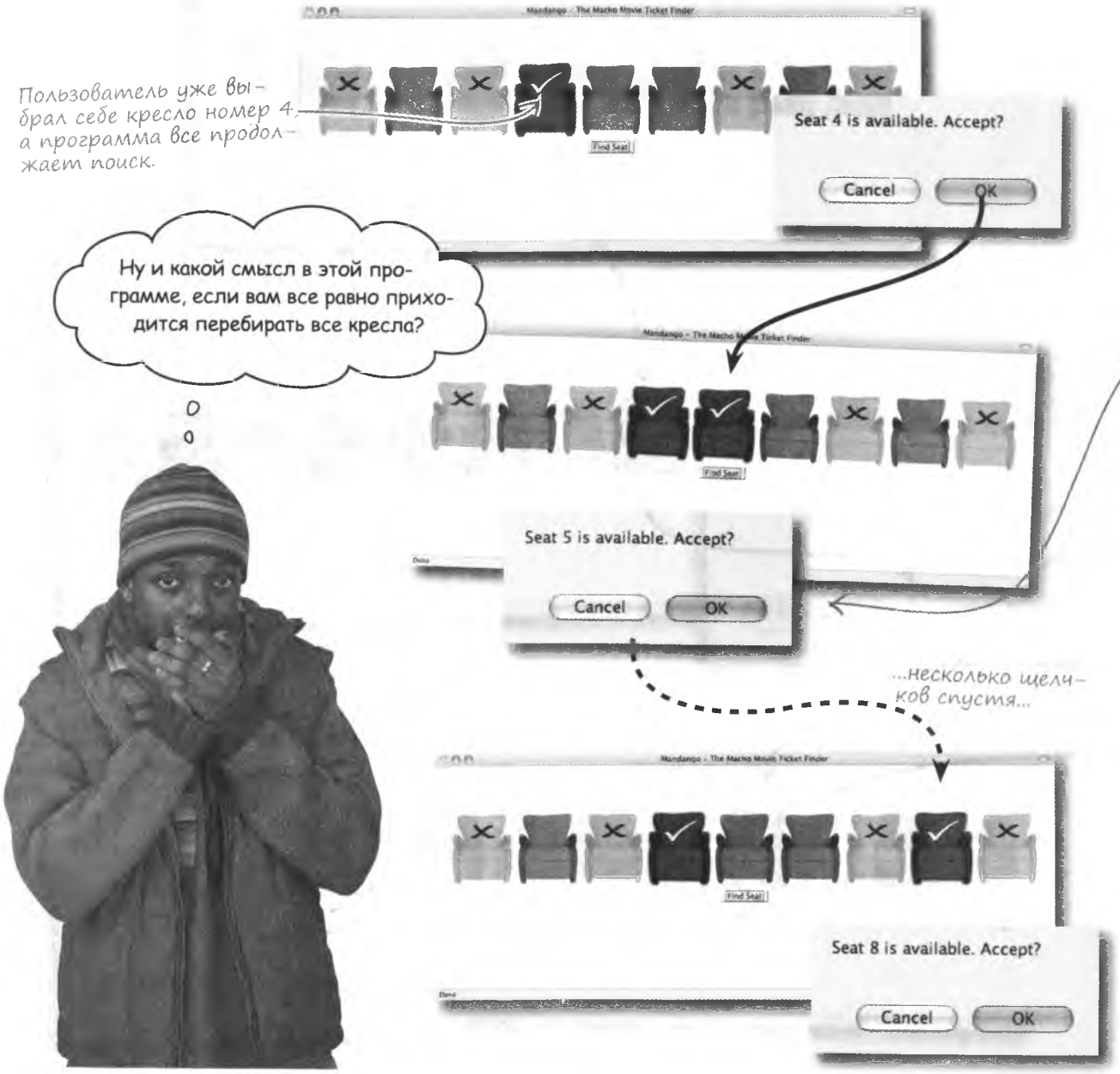
Проверка

Первая версия приложения Mandango использует цикл `for` и массив для поиска одиночных свободных мест. И является вполне функциональной, хотя и не для мачо.



Бесконечные циклы

Поиск незанятых одиночных мест с технической точки зрения работает хорошо, вот только цикл не знает, когда ему следует прекратить свою работу. Даже после того, как пользователь забронировал себе место, щелкнув на кнопке OK, перебор и поиск дальнейших свободных мест продолжается.



Условие выхода из цикла

Так как причиной слишком большого усердия поисковой программы, судя по всему, является бесконечный цикл, Джейсон решил внимательней присмотреться к циклу `for` в функции `findSeat()`.

Метод confirm() предлагает пользователю ответить на вопрос да/нет, возвращая значение true (да) или false (нет).

```
for (var i = 0; i < seats.length; i++) {
  // Проверка доступности рассматриваемого в данный момент кресла
  if (seats[i]) {
    // Выделяем кресло и обновляем его вид
    selSeat = i;
    document.getElementById("seat" + i).src = "seat_select.png";
    document.getElementById("seat" + i).alt = "Your seat";

    // Пользователю предлагается принять предложенный вариант
    var accept = confirm("Seat " + (i + 1) + " is available. Accept?");
    if (!accept) {
      // Пользователь отказался, продолжаем поиск
      selSeat = -1;
      document.getElementById("seat" + i).src = "seat_avail.png";
      document.getElementById("seat" + i).alt = "Available seat";
    }
  }
}
```

?

Когда пользователь бронирует свободное место, ничего не происходит, и цикл продолжает работу.

Этот код запускается, если пользователь отказывается бронировать свободное место.

При щелчке на кнопке Cancel переменной `selSeat` присваивается значение –1 (ничего не выбрано), и цикл продолжает работу. Однако нет кода, который выполнялся бы после бронирования свободного места. С одной стороны, это хорошо, так как позволяет переменной `selSeat` запомнить выбранное значение. А с другой — ничто не мешает циклу продолжить поиск свободных мест.

МОЗГОВОЙ ШТУРМ

Что должно происходить при щелчке на кнопке OK и бронировании места?

Прерывание действия

Проблема с кодом приложения Mandango в том, что вам нужно было выходить из цикла сразу же после бронирования кресла. Этого можно добиться, к примеру, путем присвоения счетчику цикла `for` значения, превышающего длину массива.

```
i = seats.length + 1;
```

В этом случае проверка условия приведет к прерыванию цикла... но есть и более изящные варианты решения проблемы.

Данный код является замечательным обходным маневром, но есть и более простой путь выхода из цикла. Это оператор `break`, созданный специально для подобных случаев.

```
break;
```

Заставляет немедленно выйти из цикла.

Обнаружив оператор `break`, цикл немедленно завершает работу, игнорируя процедуру проверки условия. Таким образом, данный оператор дает вам возможность выйти из цикла в любой момент.

С ним близко связан оператор `continue`, который прекращает выполнение текущих операторов без выхода из цикла. Другими словами, оператор `continue` принудительно переводит цикл на следующий шаг.

Завершает все операции на текущем шаге цикла, принудительно переводя на следующий.

```
continue;
```

Операторы `break` и `continue` позволяют настраивать работу циклов нужным вам образом. И именно оператор `break` решает проблему с бесконечным циклом в программе Mandango.

Часто Задаваемые Вопросы

В: Выполняется ли код, стоящий в цикле `for` после оператора `break`?

О: Нет. Оператор `break` приводит к немедленному прекращению работы цикла, и все стоящие после него операторы игнорируются.

В: Чем плох вариант с увеличением показаний счетчика для выхода из цикла?

О: Вы заставляете счетчик работать непредусмотренным образом, и это может привести к проблемам. Он предназначен для подсчета элементов массива, а вы присваиваете ему значение, выходящее за границы диапазона, провоцируя выход из цикла. В общем случае показания счетчика должны обновляться только в одном месте. Разумеется, бывают случаи, когда допустимы различные обходные маневры, но это — не один из них. В конце концов, к вашим услугам всегда оператор `break`, мгновенно прерывающий цикл без принудительного изменения значения счетчика.

Возьми в руку карандаш

Цикл `for` в функции `findSeat()` должен прерываться после того, как пользователь забронировал кресло. Впишите недостающие строчки кода и комментарии к ним.

```
// Поиск свободного места среди всех возможных
for (var i = 0; i < seats.length; i++) {
  // Проверка доступности рассматриваемого в данный момент кресла
  if (seats[i]) {
    // Выделяем кресло и обновляем его вид
    selSeat = i;
    document.getElementById("seat" + i).src = "seat_select.png";
    document.getElementById("seat" + i).alt = "Your seat";

    // Пользователю предлагается принять предложенный вариант
    var accept = confirm("Seat " + (i + 1) + " is available. Accept?");
    .....
    .....
    .....
    .....
    .....
    else {
      // Пользователь отказался, продолжаем поиск
      selSeat = -1;
      document.getElementById("seat" + i).src = "seat_avail.png";
      document.getElementById("seat" + i).alt = "Available seat";
    }
  }
}
```

Возьми в руку карандаш

Решение

Цикл `for` в функции `findSeat()` должен прерываться после того, как пользователь забронировал кресло. Вот как выглядят недостающие строчки кода и комментарии к ним.

```
// Поиск свободного места среди возможных
for (var i = 0; i < seats.length; i++) {
  // Проверка доступности рассматриваемого в данный момент кресла
  if (seats[i]) {
    // Выделяем кресло и обновляем его вид
    selSeat = i;
    document.getElementById("seat" + i).src = "seat_select.png";
    document.getElementById("seat" + i).alt = "Your seat";

    // Пользователю предлагается принять предложенный вариант
    var accept = confirm("Seat " + (i + 1) + " is available. Accept?");
    if (accept) {
      // Кресло забронировано, поэтому работа цикла завершается
      break;
    }
    else {
      // Пользователь отказался, продолжаем поиск
      selSeat = -1;
      document.getElementById("seat" + i).src = "seat_avail.png";
      document.getElementById("seat" + i).alt = "Available seat";
    }
  }
}
```

Места для мачо

Как вы помните, изначально планировалось создать программу, которая будет искать группы из трех подряд свободных мест. Вариант для поиска одного свободного места уже готов, и наши друзья Сет и Джейсон готовы начать работу над новой версией своей программы.

Три свободных кресла подряд... сколько места!

Думаю, что несколько вложенных операторов if без проблем помогут найти три свободных места подряд. Да, именно так я и сделаю!

Код в деталях

Последовательность из трех кресел проверяется при помощи вложенных операторов if.

При обнаружении трех мест подряд выделяем первое из них.

```
for (var i = 0; i < seats.length; i++) {
  // Проверяем, свободно ли текущее место и два места за ним
  if (seats[i]) {
    if (seats[i + 1]) {
      if (seats[i + 2]) {
        // Выделяем кресла и обновляем их вид
        selSeat = i;
        document.getElementById("seat" + i).src = "seat_select.png";
        document.getElementById("seat" + i).alt = "Your seat";
        document.getElementById("seat" + (i + 1)).src = "seat_select.png";
        document.getElementById("seat" + (i + 1)).alt = "Your seat";
        document.getElementById("seat" + (i + 2)).src = "seat_select.png";
        document.getElementById("seat" + (i + 2)).alt = "Your seat";

        // Пользователю предлагается принять предложенный вариант
        var accept = confirm("Seats " + (i + 1) + " through " + (i + 3) + " are available. Accept?");
        if (accept) {
          // Кресла забронированы, поэтому работа цикла закончена
          break;
        }
        else {
          // Пользователь отказался, продолжаем поиск
          selSeat = -1;
          document.getElementById("seat" + i).src = "seat_avail.png";
          document.getElementById("seat" + i).alt = "Available seat";
          document.getElementById("seat" + (i + 1)).src = "seat_avail.png";
          document.getElementById("seat" + (i + 1)).alt = "Available seat";
          document.getElementById("seat" + (i + 2)).src = "seat_avail.png";
          document.getElementById("seat" + (i + 2)).alt = "Available seat";
        }
      }
    }
  }
}
```

Меняем изображение всех трех кресел на «занятые», чтобы пользователь видел, какие места остались свободными.

Если пользователь не бронирует места, возвращаем им изображение «незанятые».

* Напоминаем: код программы Mandango и все сопутствующие изображения можно скачать по адресу http://www.headfirstlabs.com/books/hfjs/.


А здорово было бы, если бы все эти вложенные операторы if заменил какой-нибудь более изящный вариант?

Логичное и элегантное решение

Проверку доступности трех мест подряд в программе Mandango можно выполнить и более простыми средствами. Вложенные операторы `if` вполне справляются со своей задачей, но код можно улучшить, сделав его более изящным.

Несмотря на возражения Сета, стоит внести в код изменения, сделав его более «изящным», то есть более эффективным, легким для понимания и редактирования. Разве не здорово было бы превратить вложенные операторы `if` в один оператор?

Логический оператор И (&&) проверяет, все ли параметры имеют значение true.

```
if (seats[i] && seats[i + 1] && seats[i + 2]) {
  ...
}
```

true *true* *true*

Логический оператор И (`&&`) проверяет, все ли параметры имеют одно и то же значение. В данном случае два оператора И используются для проверки доступности трех мест подряд. Если все три раза мы наблюдаем значение `true`, значит, мы нашли то, что нужно. Проблема решена... и как изящно!

Логические операторы

Вы уже знакомы с такими операторами сравнения, как == и <. Они сопоставляют два значения и получают результат в форме `true/false`. Аналогичный результат дают логические операторы, которые мы рассмотрим ниже.

Этот оператор вы уже видели!

AND

```
a && b
```

Возвращает значение true, если a И b имеют значение true.

OR

```
a || b
```

Возвращает значение true, если a ИЛИ b имеет значение true.

NOT

```
!a
```

Возвращает значение false, если a имеет значение true, и значение true в противном случае.

Логические операторы позволяют соединять друг с другом несколько условий, давая возможности принимать решения с соблюдением достаточно сложных условий.

Логические выражения группируются при помощи скобок.

```
if ((largeDrink && largePopcorn) || coupon)
  freeCandy();
```

Получить бесплатную конфету (free candy) можно путем комбинированного заказа ИЛИ при наличии купона.

В данном примере оператор И проверяет, сделал ли клиент комбинированный заказ, купив большой стакан напитка и большое ведро попкорна! В этом случае он получает в награду конфету. Существует и другой способ получения бесплатной конфеты, как показывает оператор ИЛИ, – наличие купона. То есть вам дают конфету, если вы заказываете большой напиток И большой попкорн ИЛИ предъявляете купон. Без логических операторов запрограммировать такое условие было бы крайне сложно.

Комбинируя логические операторы, мы создаем сложные условия.

Часто Задаваемые Вопросы

В: В чем разница между операторами сравнения и логическими операторами.

О: Начнем с того, что все они являются логическими операторами в том смысле, что результатом их работы является выражение `true` или `false`. Различаются они типами обрабатываемых данных. Операторы сравнения могут работать с любыми данными, для которых имеют смысл выражения «равно», «не равно», «больше чем» и т. п. Логические операторы работают только с логическими данными, объединяя их друг с другом.

В: То есть оператор НЕ является логическим?

О: Именно так. Ведь он работает с логическими данными. Кроме того, это *унарный* оператор, так как для него требуется всего один операнд.

В: Какую функцию в логических операторах выполняют скобки?

О: Скобки позволяют менять заданный по умолчанию порядок выполнения операторов. Причем это утверждение верно вообще для всех операторов, а не только для логических. Первым делом определяется значение выражения в скобках. Поэтому в показанном на предыдущей странице примере сначала выполняется часть `largeDrink && largePopcorn`, а только потом `||`.

Возьми в руку карандаш

Это шестой прогон цикла `for` в программе Mandango (`i = 5`). Определите доступность рассматриваемых мест и укажите, подходят ли они для настоящих мачо.

```
for (var i = 0; i < seats.length; i++) {
  // Проверяем, свободно ли текущее место и два места за ним
  if (seats[i] && seats[i + 1] && seats[i + 2]) {
    ...
  }
  ...
}
```

..... && ..... && ..... = ...

Возьми в руку карандаш

Решение

Вот каким образом распределились доступные места на шестом прогоне цикла `for` в программе Mandango (`i = 5`). К сожалению, рассматриваемые на этом шаге места не подойдут нашим мачо.


```
for (var i = 0; i < seats.length; i++) {
  // Проверяем, свободно ли текущее и следующие два места
  if (seats[i] && seats[i + 1] && seats[i + 2]) {
    ...
  }
  ...
}
```

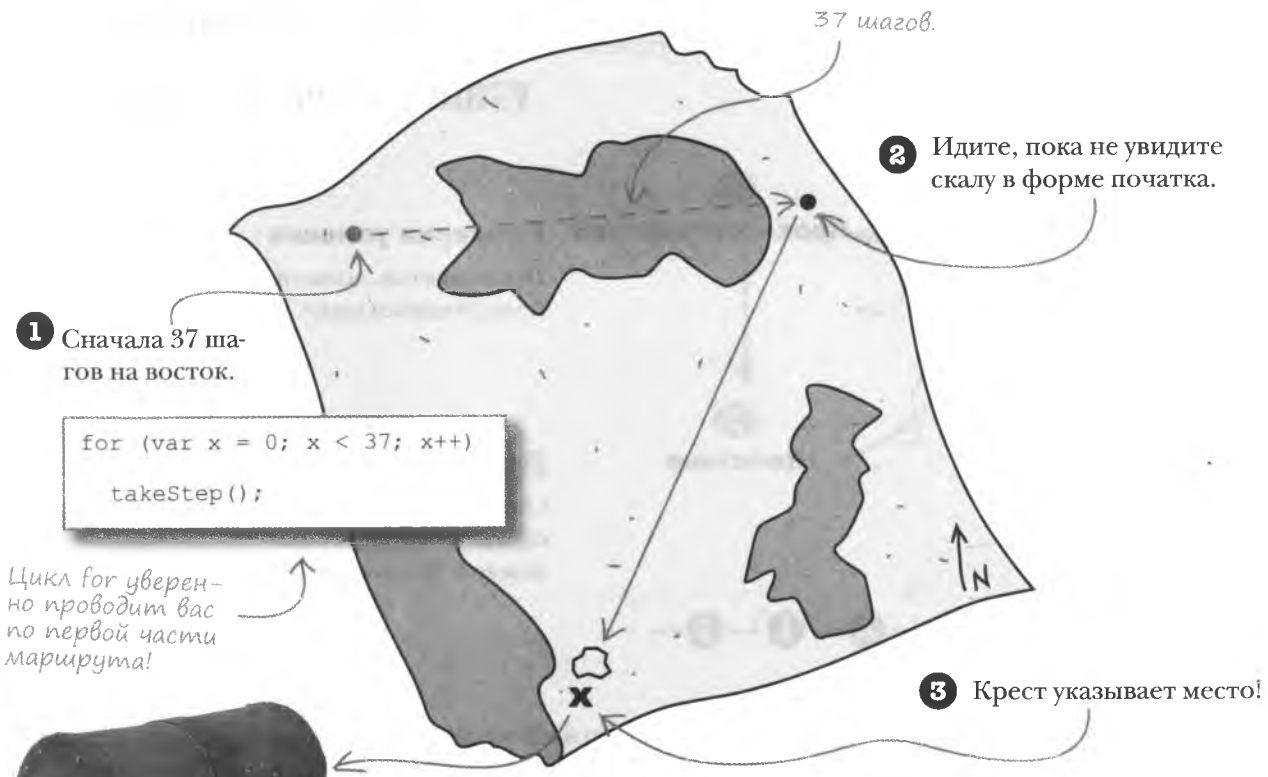
true && false && true = false

Наконец nouck заработал

Теперь программа Mandango корректно ищет последовательности из трех подряд незанятых мест, подняв сервис резервирования билетов на такую высоту, что оценить его могут даже наши мачо.

И снова карта сокровищ

Раз программа Mandango прекрасно работает, можно вернуться к поиску сокровищ. Помните эту карту?



Сокровище ждет...

Для прохождения первой части маршрута вам достаточно цикла `for`. Но вот преодолеть следующий участок он вам не поможет. Ведь невозможно создать цикл `for`, не зная точного количества его шагов.

МОЗГОВОЙ ШТУРМ

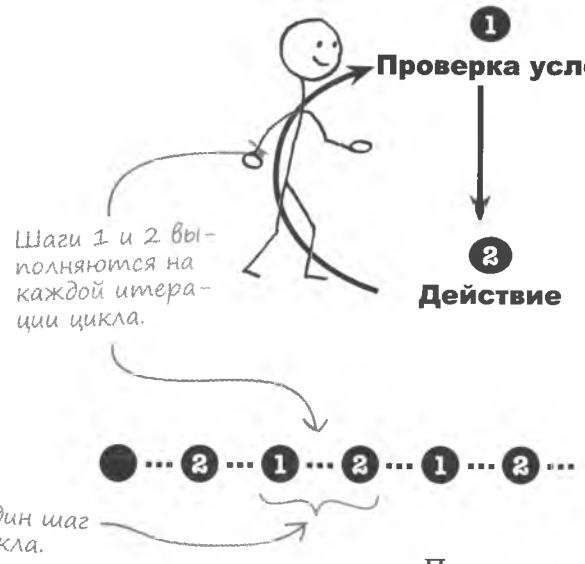
В чем разница между двумя частями карты? Каким образом вы бы создали цикл для прохождения второй части?

Цикл while

Хотя теоретически цикл `for` можно создать и для второй части карты, существует лучший вариант. Основой цикла `for` является счетчик, а вот цикл `while` работает, **пока** не будет выполнено заданное условие. И это условие может быть никак не связано с числом итераций (повторений).

Цикл `while` состоит из двух частей:

Цикл while повторяет код, нока результатом проверки условия не станет значение true.



Проверка условия

Проверяется, должен ли продолжаться цикл.

Действие

Операторы, которые следует выполнять на каждом прогоне цикла.

Примененный ко второй части карты цикл `while` дает такой же неожиданно простой код, как и цикл `for` в первой части:

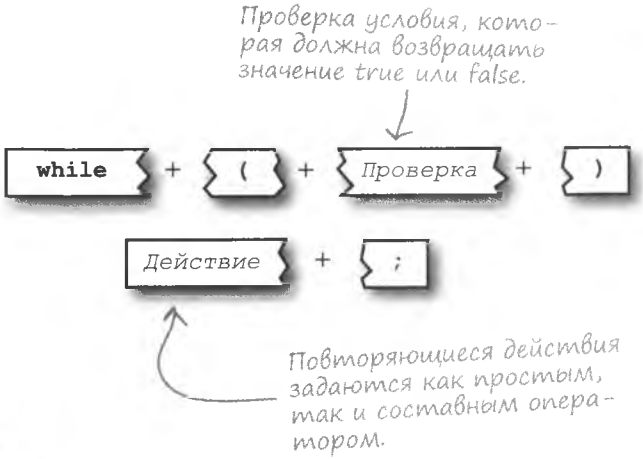
```
1
while (!rockVisible)
  takeStep();
2
```

Вот как функционируют различные части цикла `while`:

1. Проверяем, видна ли скала. Если да, переходим к шагу 2, если нет, выходим из цикла.
2. Запускаем код цикла. В данном случае вызываем функцию `takeStep()`.

Анализ цикла while

Более простой по структуре, чем цикл `for`, цикл `while` точно так же описывается определенной формулой:



Будьте аккуратны с условием выхода из цикла.

Цикл while не имеет встроенного кода обновления, поэтому его внутренний код должен быть написан таким образом, чтобы влиять на условие выхода. Иначе цикл может оказаться бесконечным.

Перепишите код уже знакомого вам упражнения, в котором пользователю предлагалось ввести число больше 0 и который использовался для обратного отсчета перед началом фильма (4, 3, 2, 1, Roll film!). Замените цикл `for` циклом `while`.

```
var count = prompt("Enter a number greater than 0:", "10");

if (count > 0) {

.....

.....

.....

.....

.....

.....

}

else

  alert("The number wasn't greater than 0. No movie for you!");
```

Упражнение Решение

Вот как после замены цикла `for` циклом `while` выглядит код уже знакомого вам упражнения, в котором пользователю предлагалось ввести число больше 0 и которое использовалось для обратного отсчета перед началом фильма (4, 3, 2, 1, Roll film!).

Сохраним число в переменной count.

Пользователю предлагается ввести число.

```
var count = prompt("Enter a number greater than 0:", "10");

if (count > 0) {
  var x = count;
  while (x > 0) {
    alert("Starting in..." + x);
    x--;
  }
  alert("Roll film!");
}
else
  alert("The number wasn't greater than 0. No movie for you!");
```

Убедимся, что переменная count больше 0.

Обратный отсчет до 0.

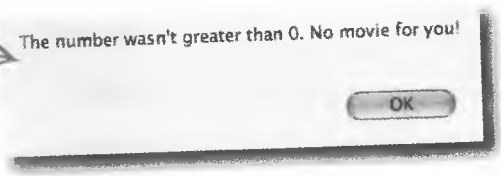
В данном случае счетчик создается вне цикла while.

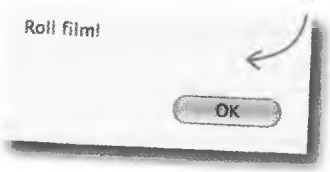
Значение переменной x уменьшается на 1 внутри цикла.

Внутри цикла составной оператор.

Некорректные данные.

Обратный отсчет закончен!





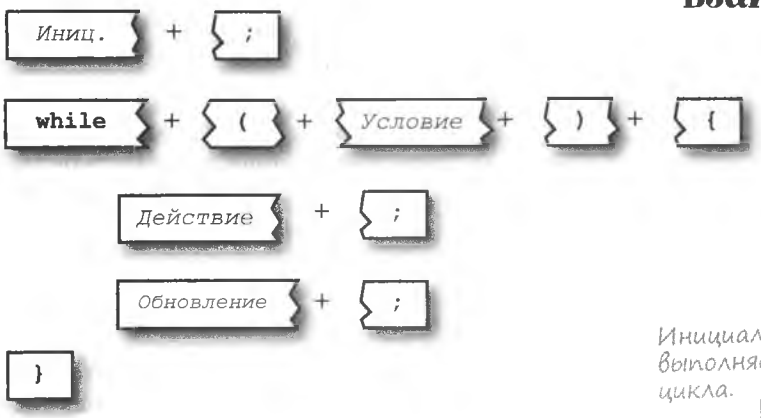
КЛЮЧЕВЫЕ МОМЕНТЫ

- Оператор `break` **немедленно** завершает цикл, пропуская оставшийся код.
- Логические операторы позволяют создавать сложные условия.
- Цикл `while` выполняет указанный код, **пока** выполняется заданное условие.
- Код в теле цикла `while` должен влиять на условие выхода.

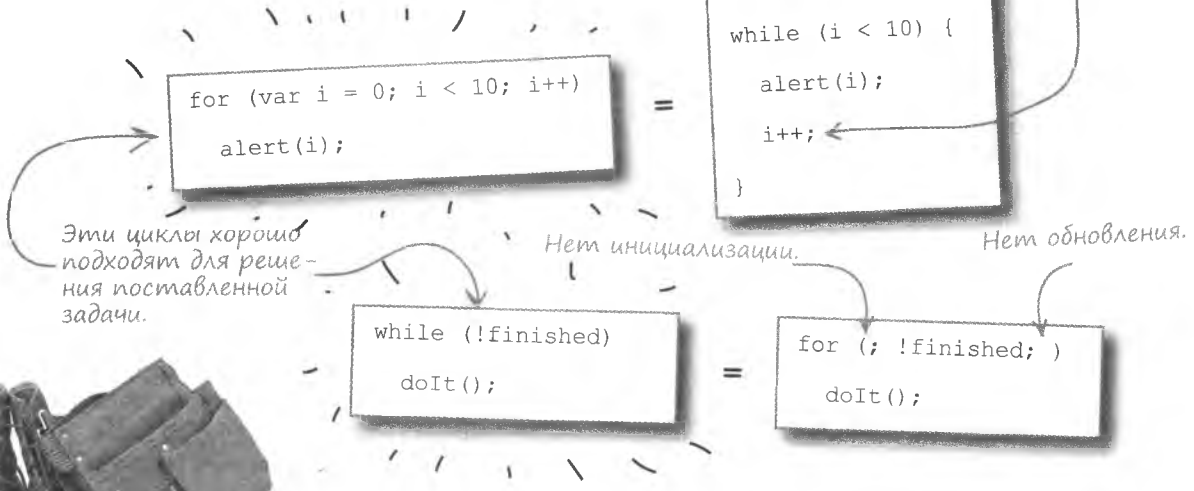
Выбор подходящего цикла

Циклы while и цикл for являются взаимозаменяемыми.

Предыдущее упражнение показало, что иногда одну и ту же проблему можно решить как при помощи цикла `for`, так и при помощи цикла `while`. Более того, превращение цикла `for` в цикл `while` осуществляется по следующей формуле:



Итак, вы убедились, что технически одну и ту же задачу можно решить с помощью любого из циклов. Но в большинстве случаев следует использовать строго определенный цикл. Может быть, все дело в изяществе?



Выбор между циклами `for` и `while` подобен выбору наиболее подходящего для работы инструмента. Другими словами, «механика» цикла должна соответствовать поставленной задаче.

Беседа у камина

Циклам For и While сложно не повториться.

Цикл For:

Вот и мы, пара повторяющихся парней, которые держатся вместе.

Я вовсе не сложный, просто ко мне добавлена определенная структура. Если нужен цикл с численным счетчиком, я легко позволяю инициализировать и редактировать данный параметр.

Звучит слишком неопределенно, хотя я думаю, что это должно работать. Но мне по душе бóльшая точность. Именно поэтому я предпочитаю инициализироваться до начала работы и обновляться в процессе итераций, просто чтобы быть уверенным, что все работает так, как нужно. Я маниакально слежу за тем, чтобы работать как часы.

Я знаю, что возможны циклы разной структуры. Но лично я предпочитаю надежность.

Цикл While:

Да. Хотя должен сказать, что количество шагов, при помощи которых тебя заставляют работать, представляется мне избыточным.

Это так, но, как ты помнишь, далеко не все циклические повторения связаны со счетом. Есть потрясающие циклы, вообще никак не связанные с цифрами. Когда требуется только сказать «Эй, делай вот это некоторое время!», я незаменим.

Восхищен твоим подходом к работе. Но понимаешь ли ты, что циклическое повторение действий вполне осуществимо и без всех этих формальностей вроде предварительной инициализации и обновления? Кроме того, я часто повторяю код, для которого не требуется инициализации. А обновление происходит прямо внутри меня. В итоге я могу сфокусироваться на непосредственной работе, то есть на итерациях.

Цикл For:

Да, это так. Главное, что мы оба делаем свою работу. И я даже иногда вижу, где мой подход является излишним.

Я готов слушать это снова и снова!

Ничего страшного. Спасибо за беседу!

Цикл While:

Наверное, все дело в том, что каждый цикл имеет свой стиль работы. Тебе нравится иметь под рукой все элементы управления, в то время как я сосредоточен на процессе.

Полностью с тобой согласен! Главное, что работы хватит нам обоим.

Главное, что работы... ой, кажется, мне пора прерывать работу. Прошу прощения.

Часто Задаваемые Вопросы

В: Цикл `while` кажется очень простым. Я чего-то не понимаю?

О: Вовсе нет. Простота не является синонимом слабости или ограниченности. Более того, вы удивитесь, обнаружив, насколько мощным инструментом является цикл `while`. Разумеется, он состоит только из условия и выполняемого кода, но в некоторых случаях большего и не требуется. Особенно в тех случаях, когда условие выхода из цикла является достаточно сложным. Кроме того, в тело цикла можно поместить любое количество кода, ведь вы можете воспользоваться составным оператором.

В: Будет ли работать цикл вида `while (true)`...?

О: Да... и даже слишком усердно. Ведь вы создали бесконечный цикл, потому что результат проверки условия всегда будет иметь значение `true`. А цикл `while` продолжает работу, пока условие не перестанет выполняться, то есть результатом его проверки не станет значение `false`. Страшно подумать, сколько бесконечных циклов вынуждены работать, работать и работать... эй, прекратите немедленно!

В: Может ли случиться так, что тело цикла никогда не будет выполнено?

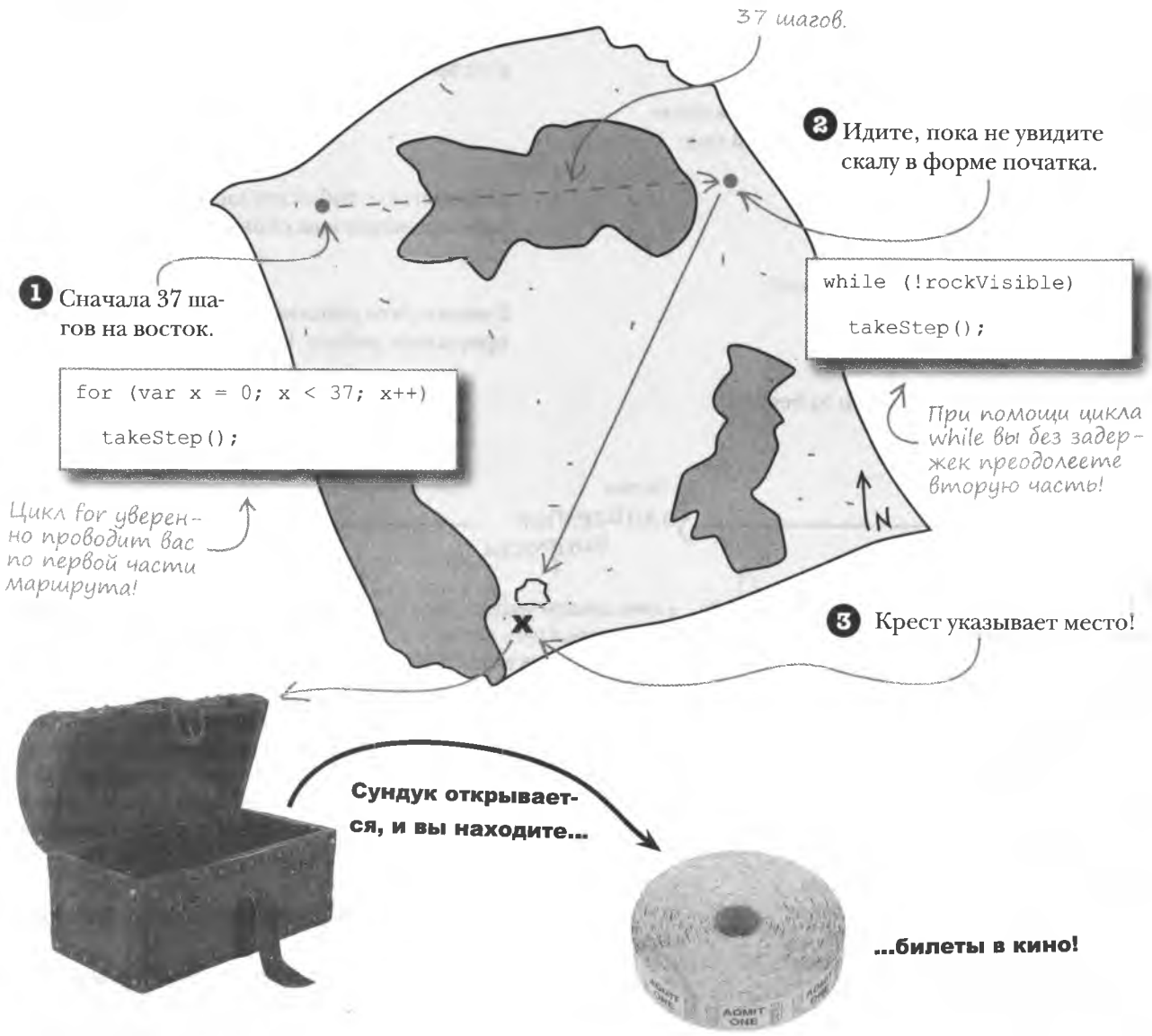
О: Да. Для перехода к операторам в теле цикла нужно, чтобы результатом проверки условия было значение `true`. Поэтому если условие не проходит проверку, перехода к телу цикла не происходит. Вы выходите из него еще до начала его работы.

В: Допустимы ли вложенные циклы?

О: Конечно! Вложенные циклы позволяют создавать различные уровни повторений. На данном этапе это может звучать для вас странно, но это — правда. Вы познакомитесь с вложенными циклами позже, когда программа Mandango от ряда перейдет к целому кинотеатру!

Сокровище в награду

Воспользовавшись циклом `for`, за которым следует цикл `while`, вы сможете пройти указанный на карте маршрут и прибыть в точку, помеченную крестом.



Разве это не знак судьбы? Вы познакомились с циклом `while` и обнаружили билеты в кино, значит, нам пора вернуться... к работе над приложением Mandango!

Возьми в руку карандаш

Перепишите функцию `findSeats()`, заменив цикл `for` циклом `while`. Добавьте переменную `finished`, которая вместо оператора `break` будет отвечать за выход из цикла.

```
.....
.....
  // Проверяем текущее место и два места за ним
  if (seats[i] && seats[i + 1] && seats[i + 2]) {
    // Выделяем кресла и обновляем их вид
    ...

    // Пользователю предлагается принять предложенный вариант
    var accept = confirm("Seats " + (i + 1) + " through " + (i + 3) +
      " are available. Accept?");

    .....
    .....
    .....
    .....
    else {
      // Пользователь отказался, продолжаем поиск
      ...
    }
  }

  // Увеличиваем показания счетчика на единицу

  .....
}
```

Возьми в руку карандаш

Решение

Вот как выглядит функция `findSeats()` после замены цикла `for` циклом `while` и ввода новой переменной `finished`, которая вместо оператора `break` будет отвечать за выход из цикла.

Инициализация счетчика цикла и переменной "finished".

Цикл работает, пока значение счетчика меньше числа сидений И переменная "finished" не принимает значение true.

Этот цикл в каком-то смысле является гибридом того, что мы видели раньше. Ведь выход из цикла зависит как от показаний счетчика, так и от логического выражения. В таких случаях лучше использовать цикл while.

```
var i = 0, finished = false;
while ((i < seats.length) && !finished) {
  // Проверяем текущее место и два места за ним
  if (seats[i] && seats[i + 1] && seats[i + 2]) {
    // Выделяем кресла и обновляем их вид
    ...

    // Пользователю предлагается принять предложенный вариант
    var accept = confirm("Seats " + (i + 1) + " through " + (i + 3) +
      " are available. Accept?");

    if (accept) {
      // Кресла забронированы, поэтому работа цикла закончена
      finished = true;
    }
    else {
      // Пользователь отказался, продолжаем поиск
      ...
    }
  }

  // Увеличиваем показания счетчика на единицу
  i++;
}
```

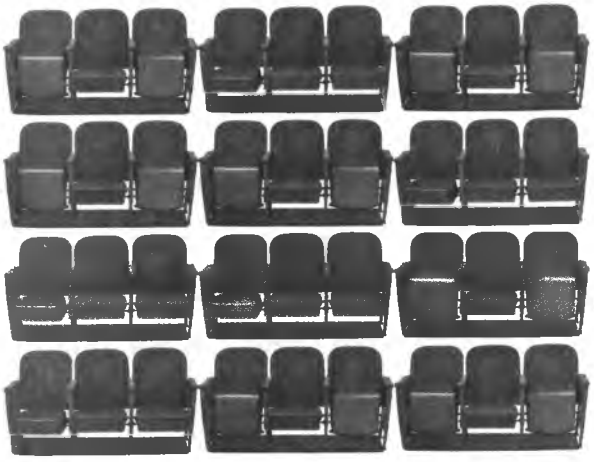
Инкремент счетчика цикла.

Присвоение переменной "finished" значения true приводит к выходу из цикла. Именно поэтому в данном месте вам больше не требуется оператор break.

Приложение Mandango выглядит здорово, вот только я не знаю ни одного кинотеатра со всего одним рядом кресел. Надо бы его отредактировать...

Кинотеатр — место моделирования данных

Джейсон прав. Чтобы программа Mandango стала функциональной, ее нужно переписать с учетом реальных условий. Пример с одним рядом кресел очень помог, так как позволил прямо связать массив логических данных с изображениями. Теперь нам нужно добавить к массиву второе измерение. Да, да, мы говорим именно о двумерном массиве!



В этом кинотеатре четыре ряда по девять кресел в каждом. Очень уютно!

Нам потребуется массив размером 9 × 4, который будет соответствовать четырем рядам из девяти кресел.

Каждый элемент двумерного массива относится к логическому типу.

Это еще одно измерение индексов массива.

	false	true	false	true	true	true	false	true	false
0	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
2	0	1	2	3	4	5	6	7	8
3	0	1	2	3	4	5	6	7	8

Двумерные массивы

Для создания двумерного массива вам не потребуется никаких новых сведений. Вы всего лишь сформируете массив, элементами которого будет другой массив. Именно это добавит второе измерение. В результате вы получаете **таблицу** данных, распределенных по строкам и столбцам.

Создадим дополнительные массивы, которые станут элементами первого массива. Это второе измерение!

Начнем с массива, который станет основой для дополнительных массивов. Это первое измерение!

```
var seats = new Array(new Array(9), new Array(9), new Array(9), new Array(9));
```

Четыре вложенных массива дадут нам четыре ряда.

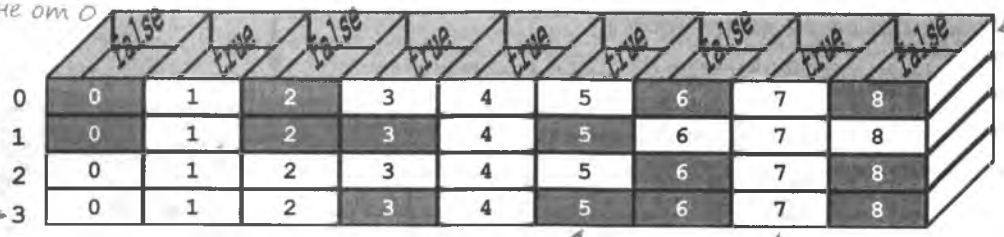
Работая с программой Mandango, мы знаем начальные значения элементов, поэтому для создания двумерного массива воспользуемся массивом констант. Другими словами, мы одновременно создадим и инициализируем массив!

Двойные скобки указывают на двумерный массив.

```
var seats = [[ false, true, false, true, true, true, false, true, false ],
  [ false, true, false, false, true, false, true, true, true ],
  [ true, true, true, true, true, true, false, true, false ],
  [ true, true, true, false, true, false, false, true, false ]];
```

Каждый вложенный массив имеет свои собственные индексы, в данном случае в диапазоне от 0 до 3.

Первый перечень логических значений соответствует первому ряду двумерного массива.



False — место уже занято.

True — место свободно.

Двумерные массивы позволяют сохранять строки и столбцы таблиц.

Два ключа доступа

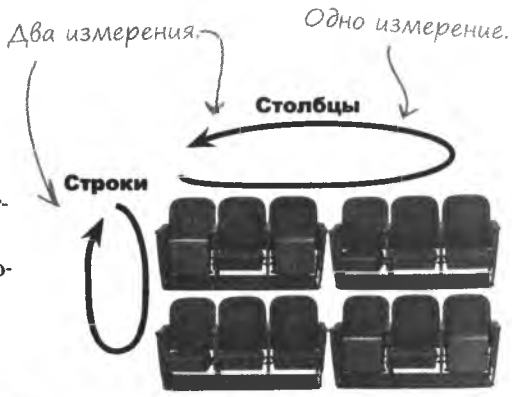
Доступ к данным двумерного массива отличается от уже привычной вам процедуры указанием дополнительного индекса. Другими словами, вам приходится как бы указывать номер строки и столбца, определяя местоположение массива. Вот как будет выглядеть код доступа к элементу, четвертому во втором ряду:

Второму ряду соответствует индекс 1 (нумерация начинается с 0).

```
alert(seats[1][3]);
```

Четвертому элементу в ряду соответствует индекс 3.

Для просмотра элементов двумерных массивов нам потребуются вложенные циклы. Внешний цикл будет управлять переходом от одного ряда к другому, в то время как внутренний — просматривать места.



Вложенные циклы дают доступ к данным двумерных массивов.

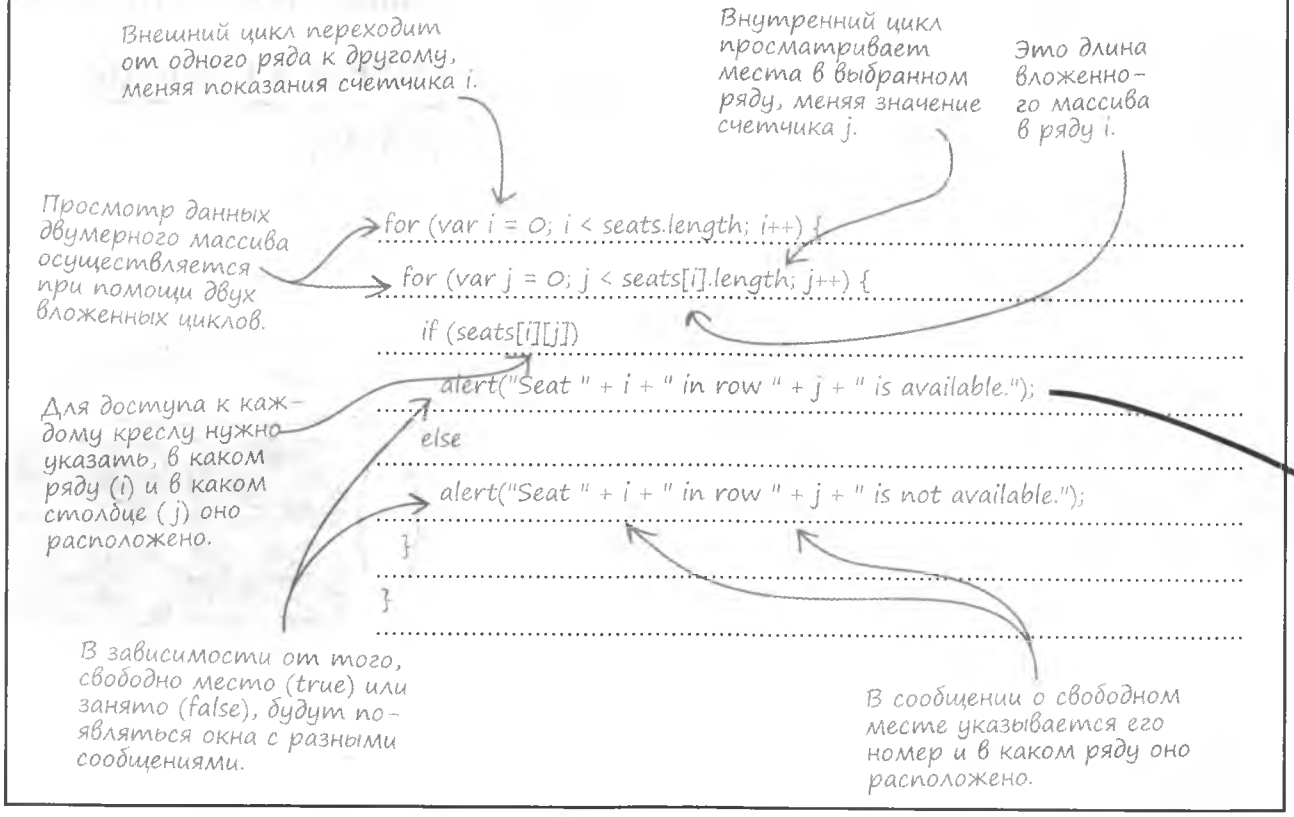
Возьми в руку карандаш

Напишите код просмотра элементов двумерного массива `seats`, извещающий пользователя о состоянии каждого места.

Возьми в руку карандаш

Решение

Вот код просмотра элементов двумерного массива `seats`, извещающий пользователя о состоянии каждого места.



Часто Задаваемые Вопросы

В: Существуют ли массивы большей размерности?

О: Да, хотя визуализировать их сложнее. Трехмерные массивы обычно соответствуют x-y-z–координатам точки в пространстве. Массивы еще большей размерности применяются в редких случаях. Чтобы добавить очередное измерение, достаточно сделать элементы внешнего массива в свою очередь массивами.

В: Если инициализация элементов массива происходит в момент его создания, можно ли добавлять в него элементы постфактум?

О: Вы в любой момент можете назначить данные неиспользуемому элементу массива. Например, в нашем примере можно создать еще один ряд (с индексом 4). Достаточно назначить массив элементу `seats[4]`. Можно также воспользоваться функцией `push()` и добавить новый элемент в конец массива.

В: Двумерные массивы должны содержать одно и то же количество рядов?

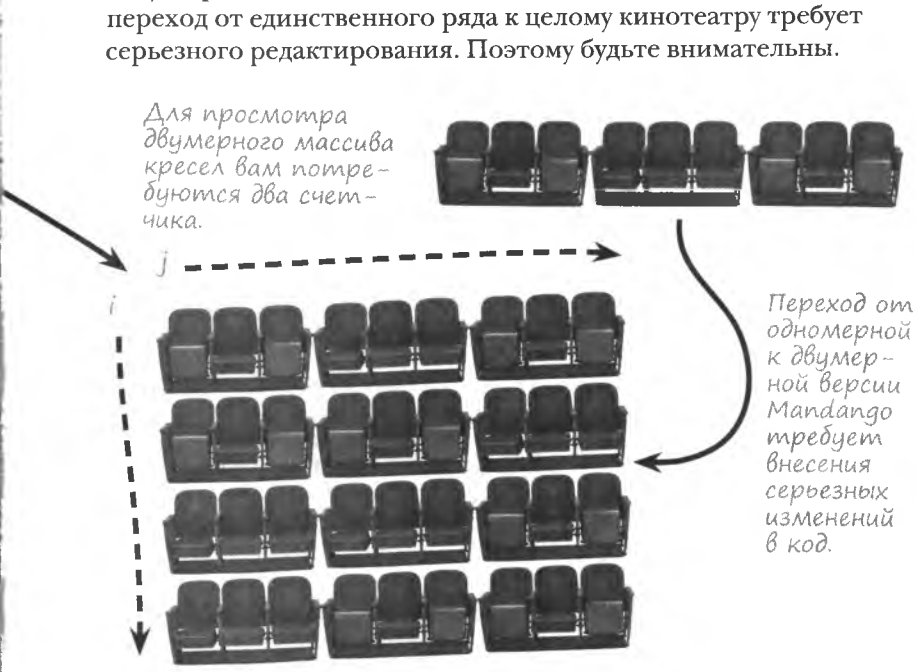
О: Не обязательно. Но помните, что в этом случае могут возникнуть проблемы с циклами, так как вложенные циклы обычно работают с массивами одинаковой длины. Так что, несмотря на принципиальную возможность варьировать длину рядов двумерного массива, лучше этого избегать.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Двумерные массивы позволяют сохранять строки и столбцы **таблиц**.
- Для доступа к элементам двумерного массива указывайте индексы **строки и столбца**.
- Для просмотра элементов двумерных массивов используются **вложенные циклы**.
- Как и в обычном случае, двумерные массивы могут быть инициализированы набором констант в момент их создания.

Двумерная версия Mandango

Вы уже работали с отдельными частями кода Mandango, но переход от единственного ряда к целому кинотеатру требует серьезного редактирования. Поэтому будьте внимательны.

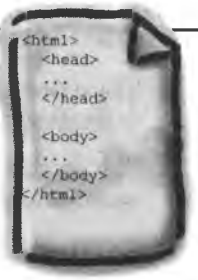


МОЗГОВОЙ ШТУРМ

Каким образом нужно поменять код программы Mandango, чтобы она начала работать с целым кинотеатром? Как бы вы это визуализировали?

Двумерное Mandango

Вот полный код для двумерной версии Mandango!



mandango.html

```
<html>
<head>
  <title>Mandango - поиск билетов для мачо</title>

  <script type="text/javascript">
    var seats = [[ false, true, false, true, true, true, false, true, false ],
                 [ false, true, false, false, true, false, true, true, true ],
                 [ true, true, true, true, true, true, false, true, false ],
                 [ true, true, true, false, true, false, false, true, false ]];
    var selSeat = -1;

    function initSeats() {
      // Задаем вид всех кресел
      for (var i = 0; i < seats.length; i++) {
        for (var j = 0; j < seats[i].length; j++) {
          if (seats[i][j]) {
            // Задаем свободные места
            document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
            document.getElementById("seat" + (i * seats[i].length + j)).alt = "Available seat";
          }
          else {
            // Задаем занятые места
            document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_unavail.png";
            document.getElementById("seat" + (i * seats[i].length + j)).alt = "Unavailable seat";
          }
        }
      }
    }

    function findSeats() {
      // Если места уже выбраны, произведите повторную инициализацию
      if (selSeat >= 0) {
        selSeat = -1;
        initSeats();
      }

      // Поиск свободных мест среди всех возможных
      var i = 0, finished = false;
      while (i < seats.length && !finished) {
        for (var j = 0; j < seats[i].length; j++) {
          // Проверяем, свободно ли текущее место и два места за ним
          if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
            // Выделяем кресла и обновляем их вид
            selSeat = i * seats[i].length + j;
            document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_select.png";
            document.getElementById("seat" + (i * seats[i].length + j)).alt = "Your seat";
            document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_select.png";
            document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Your seat";
            document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_select.png";
            document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Your seat";

            // Пользователю предлагается принять предложенный вариант
            var accept = confirm("Seats " + (j + 1) + " through " + (j + 3) +
              " in Row " + (i + 1) + " are available. Accept?");
            if (accept) {
              // Кресла забронированы, поэтому работа внешнего цикла закончена
              finished = true;
              break;
            }
            else {
```

Создан двумерный массив логических констант, содержащий информацию о доступности кресел.

Если пользователь начинает новый поиск щелчком на кнопке Find Seats, повторно инициализируем элементы массива.

Для просмотра рядов используется цикл while, в то время как цикл for осуществляет просмотр отдельных кресел.

```
            // Пользователь отказался, продолжаем поиск
            selSeat = -1;
            document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
            document.getElementById("seat" + (i * seats[i].length + j)).alt = "Available seat";
            document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_avail.png";
            document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Available seat";
            document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_avail.png";
            document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Available seat";
          }
        }
      }

      // Увеличиваем счетчик внешнего цикла на единицу
      i++;
    }
  }
  </script>
</head>

<body onload="initSeats();">
  <<div style="margin-top:25px; text-align:center">
    <img id="seat0" src="" alt="" />
    <img id="seat1" src="" alt="" />
    <img id="seat2" src="" alt="" />
    <img id="seat3" src="" alt="" />
    <img id="seat4" src="" alt="" />
    <img id="seat5" src="" alt="" />
    <img id="seat6" src="" alt="" />
    <img id="seat7" src="" alt="" />
    <img id="seat8" src="" alt="" /><br />
    <img id="seat9" src="" alt="" />
    <img id="seat10" src="" alt="" />
    <img id="seat11" src="" alt="" />
    <img id="seat12" src="" alt="" />
    <img id="seat13" src="" alt="" />
    <img id="seat14" src="" alt="" />
    <img id="seat15" src="" alt="" />
    <img id="seat16" src="" alt="" />
    <img id="seat17" src="" alt="" /><br />
    <img id="seat18" src="" alt="" />
    <img id="seat19" src="" alt="" />
    <img id="seat20" src="" alt="" />
    <img id="seat21" src="" alt="" />
    <img id="seat22" src="" alt="" />
    <img id="seat23" src="" alt="" />
    <img id="seat24" src="" alt="" />
    <img id="seat25" src="" alt="" />
    <img id="seat26" src="" alt="" /><br />
    <img id="seat27" src="" alt="" />
    <img id="seat28" src="" alt="" />
    <img id="seat29" src="" alt="" />
    <img id="seat30" src="" alt="" />
    <img id="seat31" src="" alt="" />
    <img id="seat32" src="" alt="" />
    <img id="seat33" src="" alt="" />
    <img id="seat34" src="" alt="" />
    <img id="seat35" src="" alt="" /><br />
    <input type="button" id="findseats" value="Find Seats" onclick="findSeats();" />
  </div>
 </body>
</html>
```

Именно благодаря счетчикам меняются изображения кресел и всплывающий текст.

Функция initSeats() вызывается при первой загрузке страницы.

Для четырех рядов по девять сидений потребуется 36 изображений... кошмар!

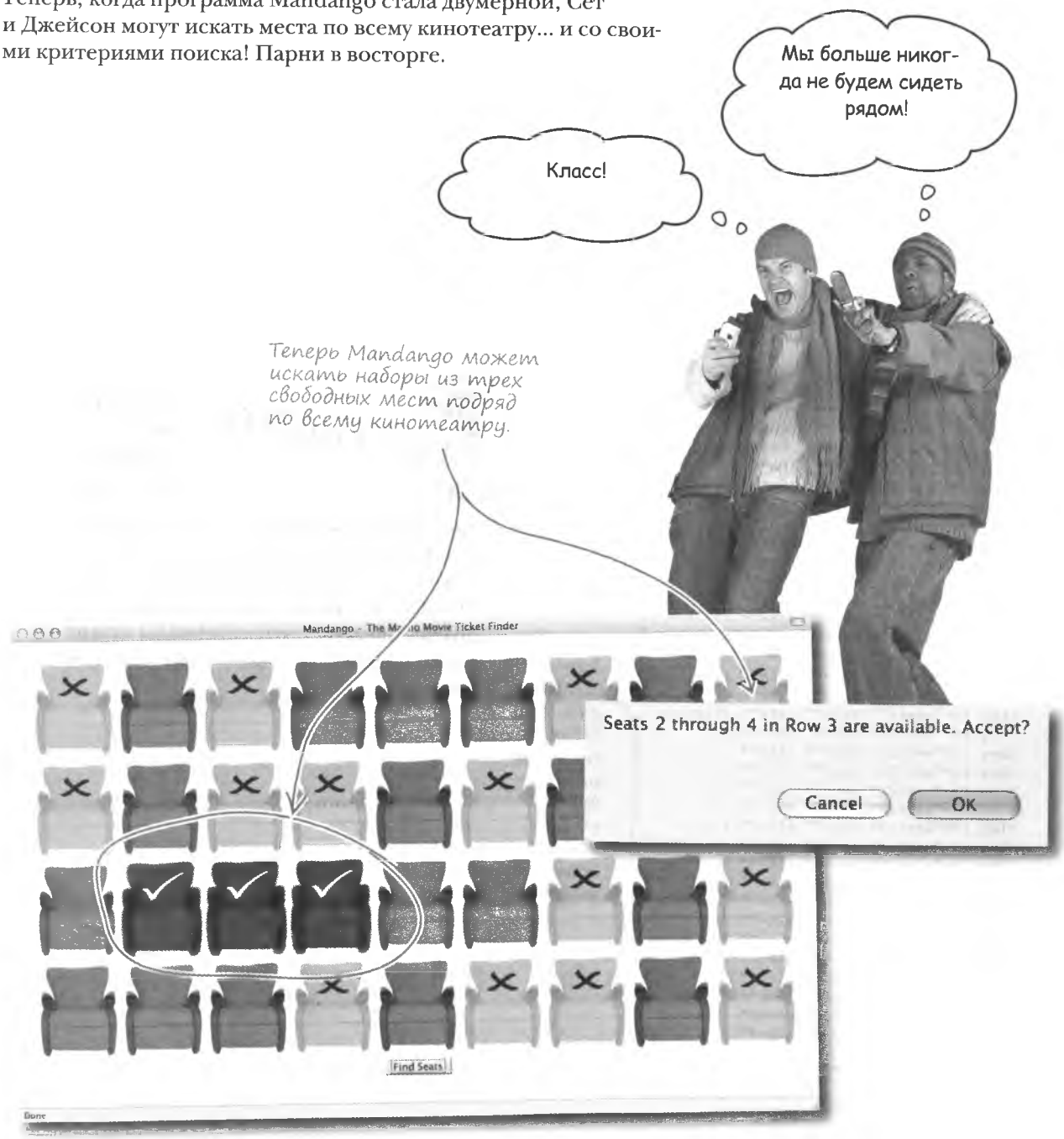
Функция findSeats() вызывается щелчком на кнопке Find Seats.

Не пугайтесь размеров этого кода.

Здесь используется уже знакомая вам техника работы с двумерными массивами, адаптированная для приложения Mandango, с его HTML-кодом и изображениями (все эти материалы можно скачать здесь: *http://www.headfirstlabs.com/books/hfjs/*).

Целый кинотеатр мест для мачо

Теперь, когда программа Mandango стала двумерной, Сет и Джейсон могут искать места по всему кинотеатру... и со своими критериями поиска! Парни в восторге.

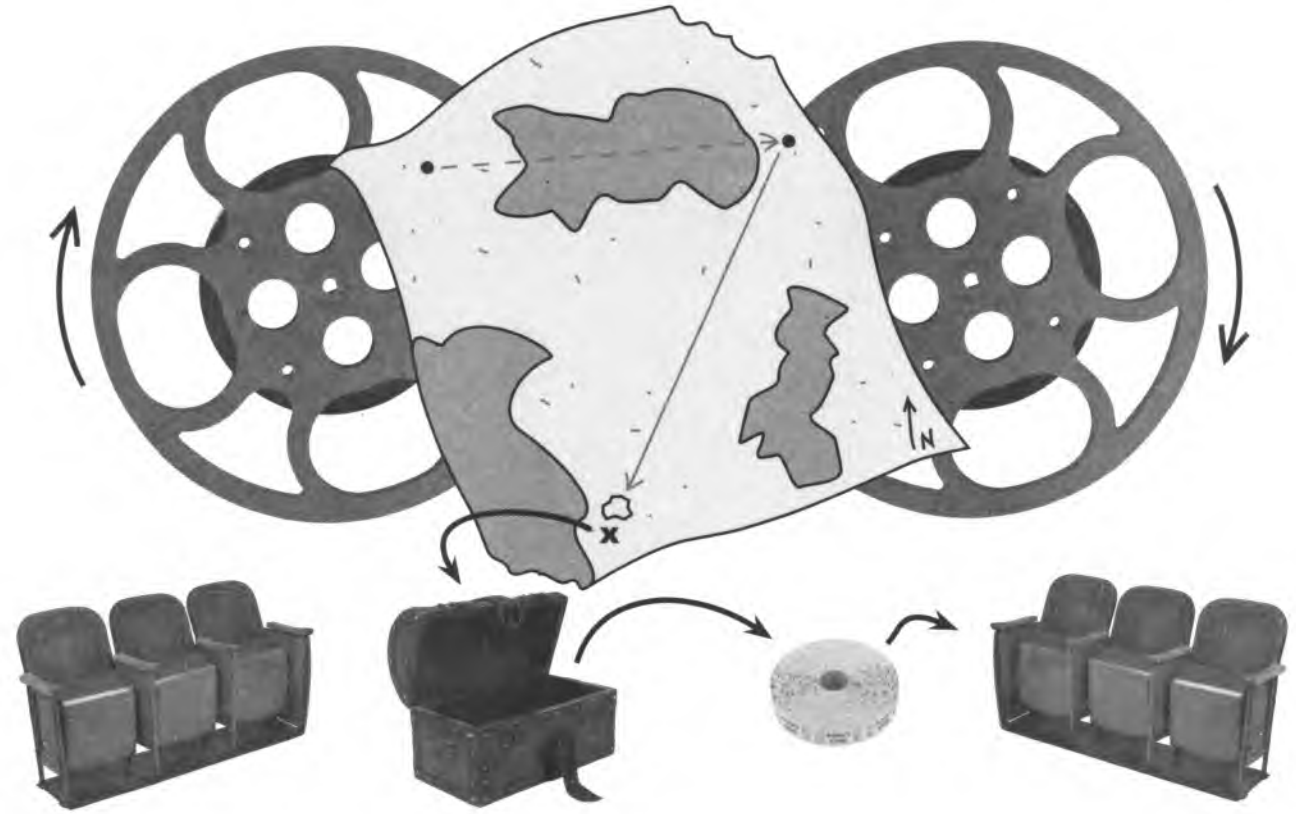


Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Что общего у циклов с фильмами?

Ум хорошо, а два лучше!



Некоторые фильмы имеют сюжет, за которым сложно уследить.

Другие фильмы привлекают зрителя постоянным движением.

Но любой фильм — это только фильм.

6 Функции

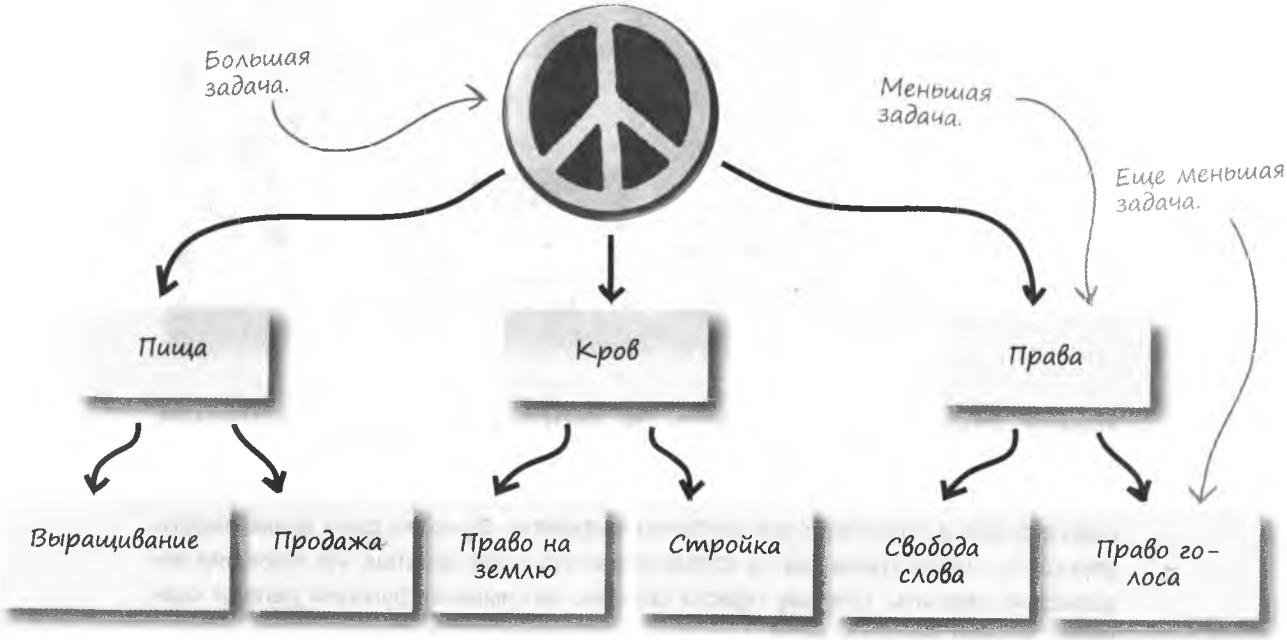
Многократное использование

Начни JavaScript выступать за экологию, это выступление возглавили бы функции. Ведь именно они увеличивают эффективность кода и позволяют использовать его многократно. Они ориентированы на решение задач и позволяют все систематизировать. Функции дают возможность упростить любой сценарий, ну кроме разве что и так простых. Их значение невозможно оценить, поэтому просто скажем, что именно функции делают сценарии такими экологичными.

Источник всех проблем

По большому счету написание сценариев для веб-страниц является решением задач. Вдумчивый подход и планирование всегда позволяют найти верное решение. Но как быть с **очень большими задачами**?

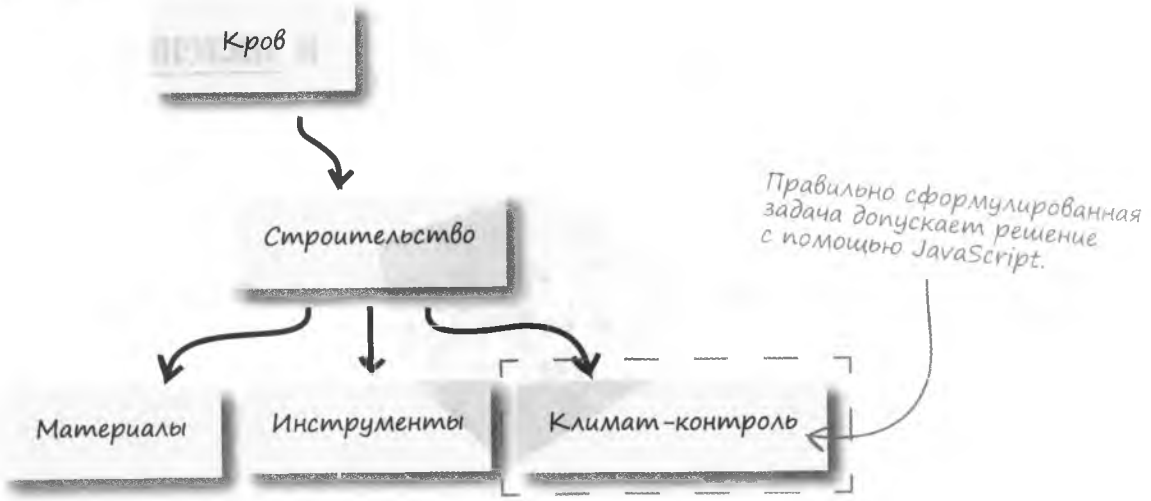
Большие задачи имеет смысл решать путем разделения их на более мелкие. Если даже после этого задачи остаются слишком большими, поделите их еще раз.



Эту процедуру можно продолжить снова, и снова, и снова...

Решение задач снизу

Разделяя вопрос о мире во всем мире на более мелкие, мы рано или поздно придем к задачам, решить которые можно при помощи JavaScrip.



Представим проблему управления климатом в терминах JavaScript. Вам потребуется эквивалент термостата, который обычно используется для контроля температуры окружающей среды. Самый простой термостат состоит из единственной кнопки «Heat».

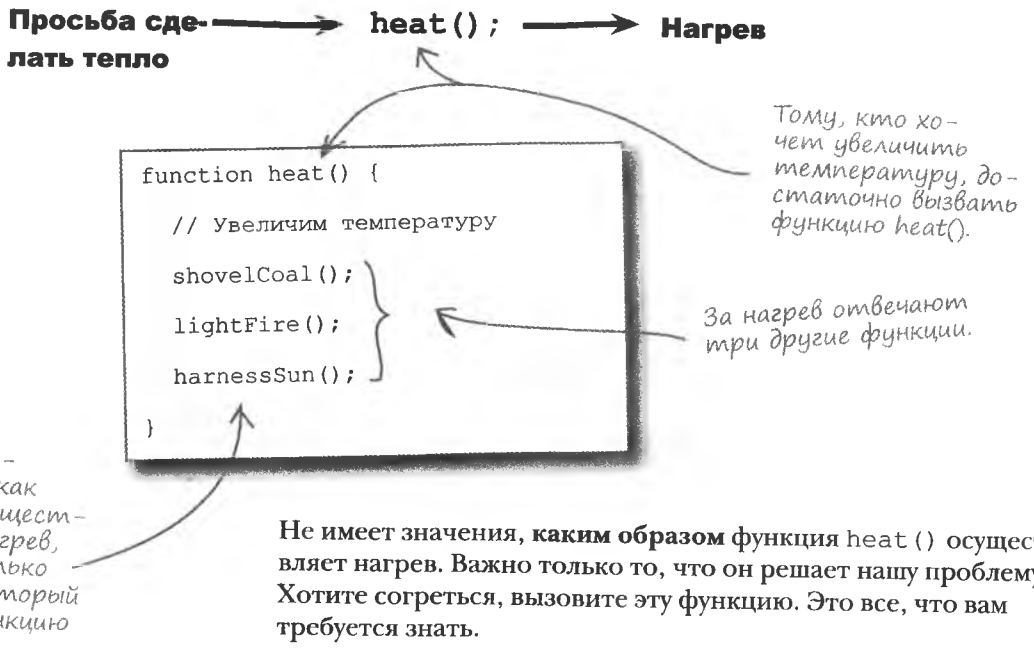
Подобная модель термостата не даст вам представления о том, как осуществляется нагрев. Вы нажимаете кнопку Heat, и становится тепло. Проблема управления климатом решена!

Функции как способ решения

Кнопка Heat на термостате эквивалентна **функции** в JavaScript. Идея простая — человек просит сделать потеплее, и функция осуществляет нагрев. Детали реализации скрыты внутри и для кода, который его вызывает, не важны. Функцию можно представить в виде «черного ящика» — информация на входе и информация на выходе, а за то, что происходит внутри, отвечает сам ящик.

Функции превращают большие задачи в маленькие.

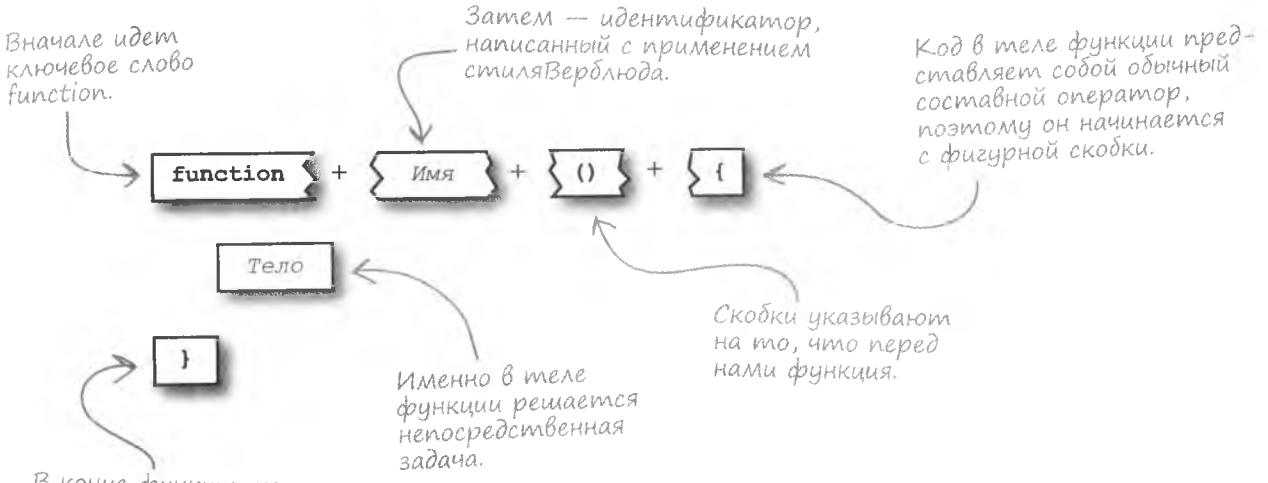
На языке JavaScript нажатие кнопки Heat эквивалентно вызову функции `heat()`...



Не имеет значения, **каким образом** функция `heat()` осуществляет нагрев. Важно только то, что он решает нашу проблему. Хотите согреться, вызовите эту функцию. Это все, что вам требуется знать.

Из чего состоит функция

Решив создать функцию, вы берете на себя ответственность за решение определенной задачи. Вы должны использовать определенный синтаксис, связывающий имя функции с запускаемым ею кодом. Вот как он выглядит в общем виде:



Еще раз внимательно посмотрим на код функции `heat()`, чтобы закрепить полученные знания о синтаксисе:

```
function heat() {
  // Увеличим температуру
  shovelCoal();
  lightFire();
  harnessSun();
}
```

Нагревание осуществляется в теле функции.

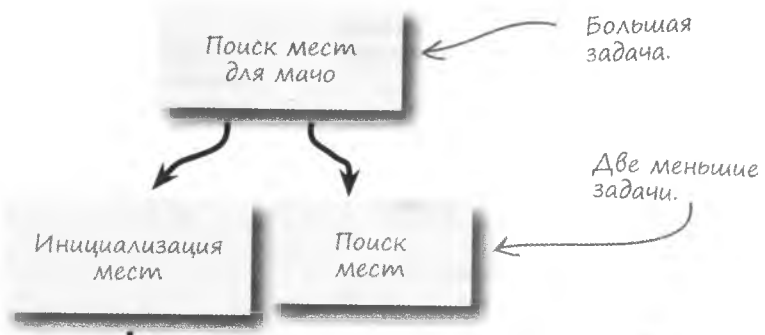
Тело функции заключено в фигурные скобки и является обычным составным оператором.

МОЗГОВОЙ ШТУРМ

Какие еще функции вы можете вспомнить?

Уже знакомые вам функции

Не так давно мы прекрасно справились с проблемой. Помните программу Mandango для поиска мест в кинотеатре? Вам требовалось инициализировать данные о местах. Вот каким образом мы поделили глобальную задачу на более мелкие:



Инициализация кресел — это маленькая задача, которая может быть решена при помощи функции `initSeats()`:

```
function initSeats() {
  // Задаем вид всех кресел
  for (var i = 0; i < seats.length; i++) {
    for (var j = 0; j < seats[i].length; j++) {
      if (seats[i][j]) {
        // Задаем свободные места
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Available seat";
      }
      else {
        // задаем занятые места
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_unavail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Unavailable seat";
      }
    }
  }
}
```

Функция `initSeats()` является частью веб-страницы Mandango. Для ее вызова нам потребуется связать ее с обработчиком события `onload`. В результате функция будет запускаться после загрузки страницы.

initSeats() не единственная функция в Mandango.

```
<body
onload="initSeats();">
    <div style="height:25px"></div>
    <div style="text-align:center">
      <img id="seat0" src="" alt="" />
      ...
      <img id="seat35" src="" alt="" /><br />
      <input type="button" id="findseats" value="Find Seats" onclick="findSeats();" />
    </div>
  </body>
</html>
```

Часто Задаваемые Вопросы

В: Каким образом составляются имена функций?

О: Первое слово пишется прописными буквами, в то время как все остальные начинаются со строчных. Поэтому функцию оценки фильмов мы назовем `rateMovie()`, а функцию вывода из зала мужчины, который хочет во время сеанса говорить по телефону — `removeInappropriateGuy()`.

В: Всегда ли функции превращают большие задачи в набор более мелких?

О: Не всегда. В некоторых ситуациях функции используются сами по себе, как набор кода. То есть одна задача может решаться целым набором функций. В этом случае каждая из них имеет свое собственное назначение. Точно так же людей назначают на различные должности, чтобы каждый мог сфокусироваться на своих обязанностях. В таких ситуациях функции улучшают структуру сценария, делая его более читаемым.

В: Как определить, какой фрагмент кода может быть помещен внутрь функции?

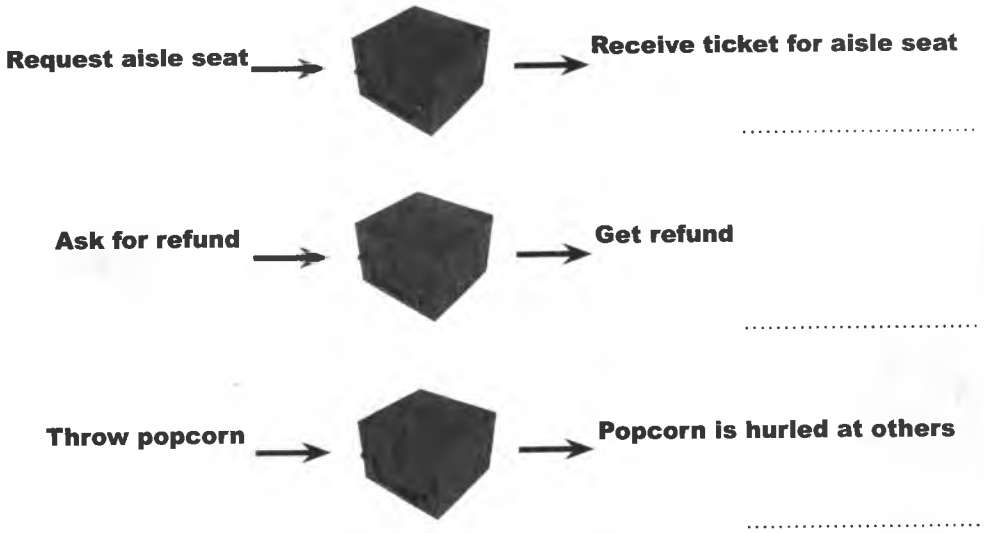
О: К сожалению, алгоритма, позволяющего однозначно определить, что вот этот фрагмент кода будет уместен в виде функции, не существует. Но существует ряд признаков, на которые имеет смысл обратить внимание. Одним из таких признаков является дублирование кода. В этом случае вы попадаете в ситуацию, когда, например, вносить одинаковые исправления приходится в несколько мест сценария одновременно. Поэтому дублирующиеся фрагменты желательно соединить в функцию. Вторым признаком является слишком большой фрагмент кода, который можно поделить на несколько логических частей.

В: В книге упоминалось, что функциям можно передавать аргументы и получать от них данные. Это действительно так?

О: Вы правы. Функции действительно берут и возвращают данные. И скоро вы увидите эту процедуру на примере функции `heat()`.

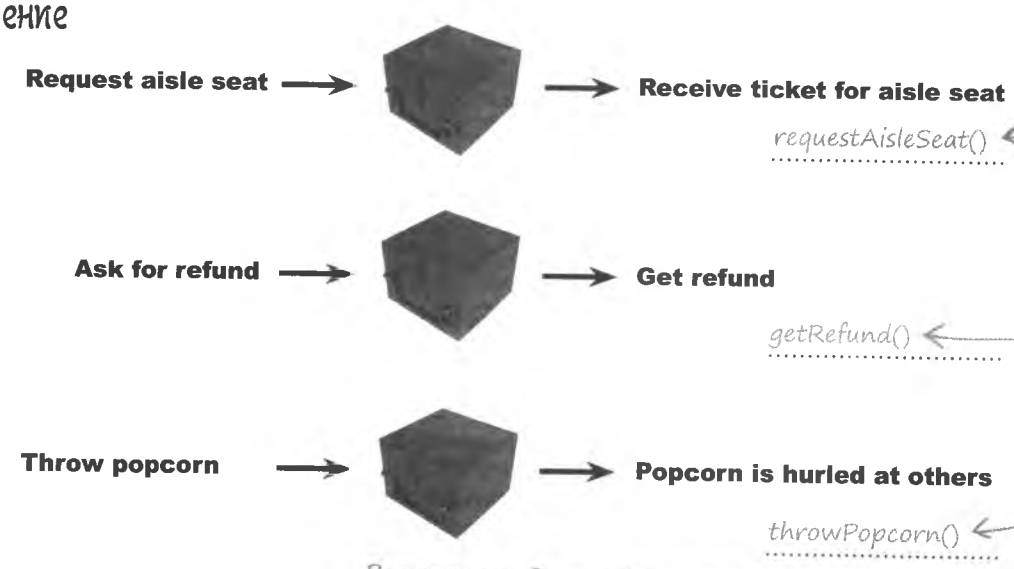
Упражнение

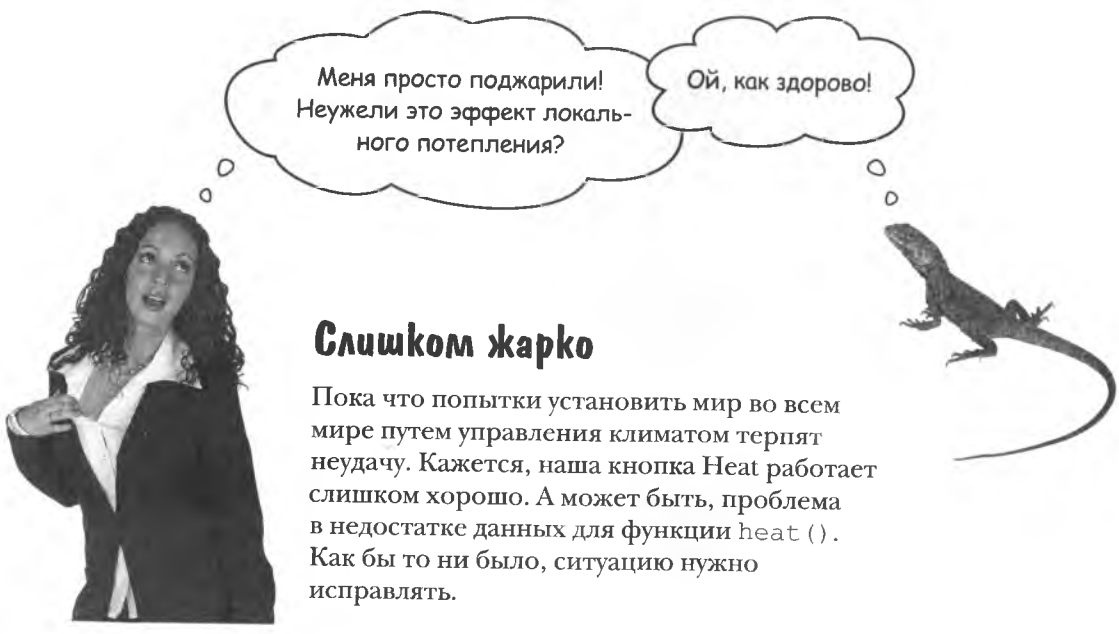
Значимое имя позволяет сразу понять назначение функции. Присвойте этим функциям названия, не забывая про стильВерблюда.



Упражнение Решение

Вот какие имена можно присвоить описанным ниже функциям.



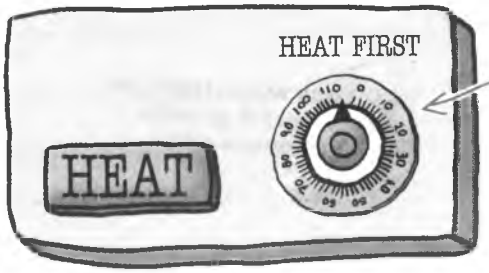


Слишком жарко

Пока что попытки установить мир во всем мире путем управления климатом терпят неудачу. Кажется, наша кнопка Heat работает слишком хорошо. А может быть, проблема в недостатке данных для функции `heat()`. Как бы то ни было, ситуацию нужно исправлять.

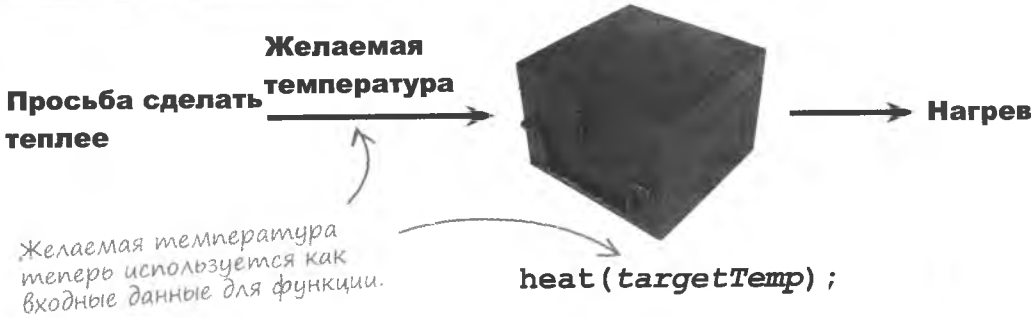
Улучшаем наш термостат

Термостат не знает, когда следует остановить нагрев, так как нужную нам температуру мы не указали. Получается, что для эффективного решения проблемы недостаточно данных. И после нажатия кнопки Heat температура будет расти бесконечно!



Этот регулятор поможет задать желаемую температуру.

Новая версия термостата позволяет указывать желаемую температуру. И благодаря этому лучше справляется с задачей «нагрева».



Упражнение

Напишите код функции `heat()`. Функция должна использовать в качестве параметра указанную пользователем температуру и осуществлять нагрев только до ее достижения. Подсказка: температуру в данный момент определяет функция `getTemp()`.

Первая строчка уже написана.

```
function heat(targetTemp){
.....
.....
.....
.....
.....
.....
.....
.....
```

Упражнение Решение

Вот как выглядит новый код функции `heat()`, написанный с учетом перечисленных на предыдущей странице пожеланий:

```
function heat(targetTemp){
  while (getTemp() < targetTemp) {
    // Нагрев
    shovelCoal();
    lightFire();
    harnessSun();
  }
}
```

Желаемая температура передается функции в качестве аргумента.

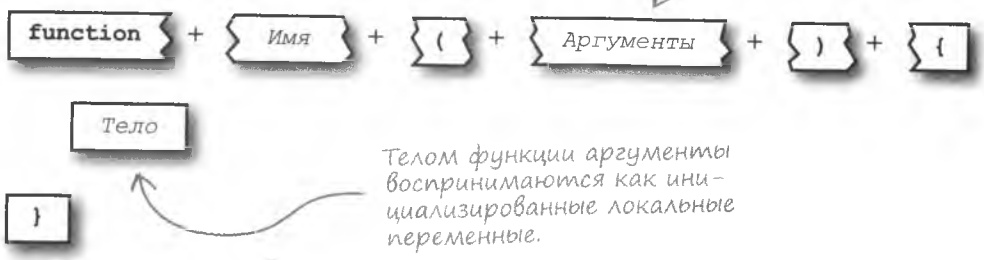
Желаемая температура указывается в условии выхода из цикла while.

Нагрев начинается только в случае, когда окружающая температура меньше желаемой.

Передача информации функциям

Данные передаются функциям JavaScript при помощи аргументов. Снова внимательно рассмотрим синтаксис; обратите внимание на то, что аргументы функции заключены в скобки.

В скобках может быть указано произвольное число аргументов.



Телом функции аргументы воспринимаются как инициализированные локальные переменные.

Теоретически вы можете передать функции произвольное количество аргументов, но с практической точки зрения желательно ограничиться двумя-тремя. В качестве аргумента может выступать любой фрагмент информации: константа (`Math.PI`), переменная (`temp`) или постоянное значение (`72`).

Аргументы как данные

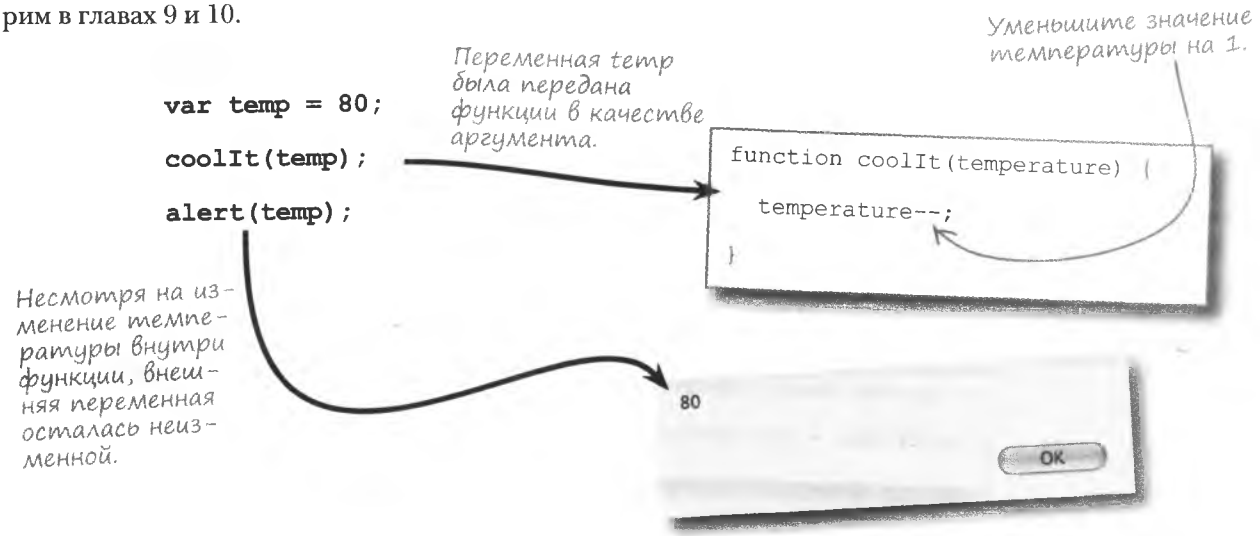
Данные, передаваемые функции в качестве аргумента, подобны инициализированным локальным переменным. В качестве примера рассмотрим функцию `heat()`, которой передается желаемая температура:

```
heat(72);
```

Желаемая температура передается функции в виде числовой константы.

Функция `heat()` воспринимает аргумент `targetTemp` как локальную переменную, которой было присвоено начальное значение 72. Поставим в тело этой функции код, вызывающий окно диалога со значением аргумента.

Хотя аргумент и напоминает локальную переменную, редактирование его **внутри** функции ни на что **не влияет**. Это правило не относится к передаваемым в качестве аргументов объектам, но о них мы поговорим в главах 9 и 10.



Избавляемся от дублирующегося кода

Функции не только позволяют разделить большую задачу на несколько более мелких, но и дают возможность избавиться от дублирующегося кода. Именно так называются одни и те же фрагменты, появляющиеся в разных местах сценария. Код может не быть полностью идентичным, но во многих случаях все равно имеет смысл превратить его в **функцию**.

В результате, если вам необходимо отредактировать какой-то фрагмент, уже не потребуется искать все его вхождения в сценарий, достаточно будет внести изменения в тело функции:

Вычисление скидки представляет собой пример ненужного дублирования кода.

```
// Билет на дневной сеанс, меньше на 10%
matineeTicket = adultTicket * (1 - 0.10);
```

```
// Билет для пенсионеров, меньше на 15%
seniorTicket = adultTicket * (1 - 0.15);
```

```
// Детский билет, меньше на 20%
childTicket = adultTicket * (1 - 0.20);
```

Показанные выше задачи сводятся к вычислению цены билетов с учетом различных скидок. Но на самом деле можно ограничиться задачей по вычислению цены на основе заданного процента скидки:

```
function discountPrice(price, percentage) {
  return (price * (1 - (percentage / 100)));
}
```

Функции возвращают данные.

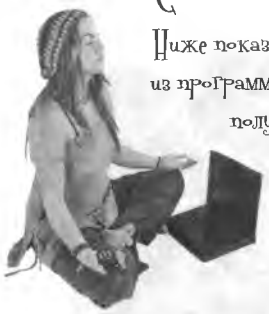
Вот как будут выглядеть указанные выше фрагменты кода после создания функции `discountPrice`:

```
// Билет на дневной сеанс, меньше на 10%
matineeTicket = discountPrice(adultTicket, 10);
```

```
// Билет для пенсионеров, меньше на 15%
seniorTicket = discountPrice(adultTicket, 15);
```

```
// Детский билет, меньше на 20%
childTicket = discountPrice(adultTicket, 20);
```

Стань экспертом по эффективности



Ниже показан код функции findSeats() из программы Mandango. Используя полученные знания, обведите фрагменты кода, которые могут послужить основой для новых функций.

```
function findSeats() {
  // Если места уже выбраны, произведите повторную инициализацию
  if (selSeat >= 0) {
    selSeat = -1;
    initSeats();
  }

  // Поиск свободных мест среди всех возможных
  var i = 0, finished = false;
  while (i < seats.length && !finished) {
    for (var j = 0; j < seats[i].length; j++) {
      // Проверяем, свободно ли выделенное место и два места за ним
      if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
        // Выделяем кресла и обновляем их вид
        selSeat = i * seats[i].length + j;
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Your seat";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Your seat";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Your seat";

        // Пользователю предлагается принять предложенный вариант
        var accept = confirm("Seats " + (j + 1) + " through " + (j + 3) +
          " in Row " + (i + 1) + " are available. Accept?");
        if (accept) {
          // Кресла забронированы, поэтому работа внешнего цикла закончена
          finished = true;
          break;
        }
        else {
          // Пользователь отказался, продолжаем поиск
          selSeat = -1;
          document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j)).alt = "Available seat";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Available seat";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Available seat";
        }
      }
    }

    // Увеличиваем счетчик внешнего цикла на единицу
    i++;
  }
}
```

Решение задачи

Вот какие фрагменты функции findSeats() из программы Mandango могут, в свою очередь, служить основой для других функций.

```
function findSeats() {
  // Если места уже выбраны, произведите повторную инициализацию
  if (selSeat >= 0) {
    selSeat = -1;
    initSeats();
  }

  // Поиск свободных мест среди всех возможных
  var i = 0, finished = false;
  while (i < seats.length && !finished) {
    for (var j = 0; j < seats[i].length; j++) {
      // Проверяем, свободно ли выделенное место и два места за ним
      if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
        // Выделяем кресла и обновляем их вид
        selSeat = i * seats[i].length + j;
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Your seat";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Your seat";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Your seat";

        // Пользователю предлагается принять предложенный вариант
        var accept = confirm("Seats " + (j + 1) + " through " + (j + 3) +
          " in Row " + (i + 1) + " are available. Accept?");
        if (accept) {
          // Кресла забронированы, поэтому работа внешнего цикла закончена
          finished = true;
          break;
        }
        else {
          // Пользователь отказался, продолжаем поиск
          selSeat = -1;
          document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j)).alt = "Available seat";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Available seat";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Available seat";
        }
      }
    }

    // Увеличиваем счетчик внешнего цикла на единицу
    i++;
  }
}
```

Так как эти шесть фрагментов кода выполняют одну и ту же задачу, их можно превратить в функцию.

Свойство length определяет число элементов вложенного массива.

Из дублирующегося кода можно извлечь некоторые атрибуты.

Функция, задающая места

Теперь, когда наши приятели-мачо осознали всю пользу от увеличения эффективности, они хотят сократить код программы Mandango, добавив в нее функции. Но прежде, чем мы приступим к написанию функции `setSeat()`, требуется понять, какие аргументы ей нужны. Для этого поищем в дублирующемся коде те фрагменты информации, которые отличаются друг от друга. Вот какие аргументы будущей функции `findSeats()` мы обнаружим:

Номер места

В данном случае это не индекс массива, а номер, который был бы присвоен месту при счете слева направо и сверху вниз, начиная с 0.

Статус

Место может быть свободным, занятым или выбранным. От этого зависит, каким изображением оно будет представлено.

Описание

Статус мест может иметь одно из трех описаний «Available seat», «Unavailable seat» и «Your seat». Он помещается в атрибут `alt` изображений.

Атрибуты функции findSeats() получены исследованием дублирующегося кода.

Возьми в руку карандаш

Напишите функцию `setSeat()` для программы Mandango.

.....

.....

.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит функция `setSeat()` в программе Mandango.

Индивидуальные данные из исходного кода заменены обобщенными аргументами.

Три аргумента перечислены через запятую.

```
function setSeat(seatNum, status, description) {
  document.getElementById("seat" + seatNum).src = "seat_" + status + ".png";
  document.getElementById("seat" + seatNum).alt = description;
}
```

Редактируем код Mandango

Превращение дублирующегося кода в функцию `setSeat()` значительно упростило код функции `findSeats()`.

Номер кресла, его статус и его описания передаются в качестве аргументов при каждом вызове функции setSeat().

Новая функция setSeat() вызывается шесть раз.

```
function findSeats() {
  ...
  // Поиск свободных мест среди всех возможных
  var i = 0, finished = false;
  while (i < seats.length && !finished) {
    for (var j = 0; j < seats[i].length; j++) {
      // Проверяем, свободно ли выделенное место и два места за ним
      if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
        // Выделяем кресла и обновляем их вид
        selSeat = i * seats[i].length + j;
        setSeat(i * seats[i].length + j, "select", "Your seat");
        setSeat(i * seats[i].length + j + 1, "select", "Your seat");
        setSeat(i * seats[i].length + j + 2, "select", "Your seat");

        // Пользователю предлагается принять предложенный вариант
        var accept = confirm("Seats " + (j + 1) + " through " + (j + 3) +
          " in Row " + (i + 1) + " are available. Accept?");
        if (accept) {
          // Кресла забронированы, поэтому работа внешнего цикла закончена
          finished = true;
          break;
        }
        else {
          // Пользователь отказался, продолжаем поиск
          selSeat = -1;
          setSeat(i * seats[i].length + j, "avail", "Available seat");
          setSeat(i * seats[i].length + j + 1, "avail", "Available seat");
          setSeat(i * seats[i].length + j + 2, "avail", "Available seat");
        }
      }
    }

    // Увеличиваем счетчик внешнего цикла на единицу
    i++;
  }
}
```

Усовершенствование Mandango

Функция `setSeat()` выгодна не только с точки зрения сокращения кода функции `findSeats()`. Увеличивается также **эффективность** функции `initSeats()`, в которой присутствует похожий код.

```
function initSeats() {
  // Задаем вид всех кресел
  for (var i = 0; i < seats.length; i++) {
    for (var j = 0; j < seats[i].length; j++) {
      if (seats[i][j]) {
        // Задаем свободные места
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Available seat";
      }
      else {
        // Задаем занятые места
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_unavail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Unavailable seat";
      }
    }
  }
}
```

Две строчки кода превратились в два вызова функции.

```
function initSeats() {
  // Задаем вид всех кресел
  for (var i = 0; i < seats.length; i++) {
    for (var j = 0; j < seats[i].length; j++) {
      if (seats[i][j]) {
        // Задаем свободные места
        setSeat(i * seats[i].length + j, "avail", "Available seat");
      }
      else {
        // Задаем занятые места
        setSeat(i * seats[i].length + j, "unavail", "Unavailable
seat");
      }
    }
  }
}
```

Если обобщить процедуру задания вида кресла, функция setSeat() хорошо работает и в этом случае.

Итак, в сценарии программы Mandango теперь восемь раз вызывается несложная функция, состоящая из пары строк кода. Это не только **упрощает** код сценария, но и облегчает его редактирование. Ведь для изменения способа задания мест достаточно внести исправления в функцию `setSeat()`. Это намного удобней редактирования восьми разбросанных по всему сценарию фрагментов кода.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Функции позволяют разбить **большую задачу на ряд более мелких** и более простых для решения.
- Функции предоставляют механизм разделения задач и превращения их в код **многократного использования**.
- Функции позволяют избавиться от **дублирующегося кода**, ведь их можно вызывать произвольное количество раз.
- Передача данных функциям осуществляется при помощи **аргументов**.

Часто Задаваемые Вопросы

В: Ограничено ли количество передаваемых функции аргументов?

О: И да, и нет. Если не брать в расчет ограниченность памяти компьютера, то формальных ограничений на количество передаваемых аргументов нет. Впрочем, если вы передали функции столько аргументов, что возникли проблемы с оперативной памятью, вам явно нужно сделать паузу и подумать над своими действиями. С практической точки зрения количество аргументов желательно выбирать таким образом, чтобы вызов функции не превратился в несуразно сложную задачу.

В: Мы узнали, что функции позволяют делить крупные задачи на более мелкие, разделяют процесс написания сценария и устраняют дублирующийся код. Они действительно настолько универсальны?

О: Да, действительно. Часто функция помогает решить несколько проблем одновременно. Впрочем, если попытаться указать главное достоинство функций, наверное, это будет разделение процесса написания сценария на фрагменты.

В: Напомните, где именно должны появляться функции. В заголовке или в теле веб-страницы?

О: Функции могут располагаться, как внутри тега `<script>` в заголовке страницы, так и во внешнем файле, импортированном в заголовок.

В: Как сделать, чтобы функция меняла значения аргументов?

О: Напрямую изменить аргументы функции нельзя, точнее говоря, эти изменения никак не отразятся на сценарии за пределами функции. Так что для изменения фрагмента данных, который передается в качестве аргумента, нужно получить от функции измененное значение. О том, как это сделать, вы узнаете чуть позже!

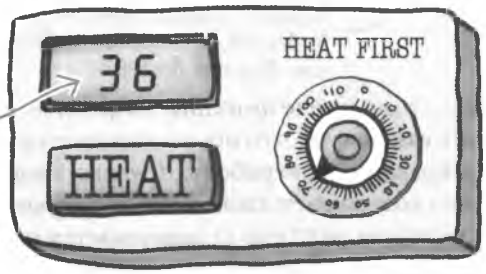
Зима в июле

С помощью функций мы улучшили программу Mandango, а вот в области климат-контроля такими результатами похвастаться пока не получается. Кажется, термостат все равно работает некорректно. Только посмотрите на дрожащего пользователя, скучающего по непрерывному нагреву после нажатия кнопки Heat.

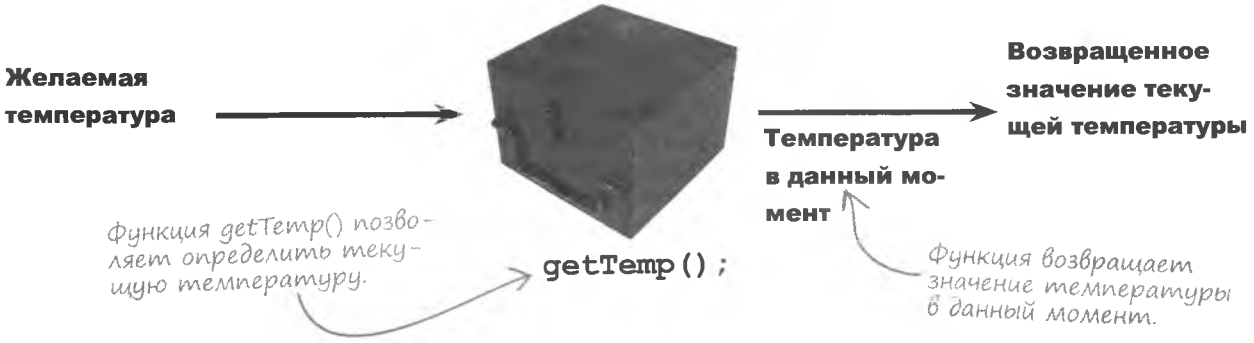
Обратная связь

Благодаря функции наш термостат позволяет задать нужную температуру, но он не сообщает о том, насколько тепло в помещении в данный момент. А это очень важный параметр, так как именно он является основой для достижения желаемой температуры. Кроме того, в одних и тех же условиях разные термостаты могут иметь разные показания. Соответственно, для корректной работы системы нам нужна обратная связь.

Окно «температура в данный момент» дает информацию о том, насколько тепло сейчас в помещении, и позволяет предположить, до какой именно температуры следует осуществлять нагрев.



В качестве обратной связи термостат теперь периодически показывает температуру в помещении в данный момент.



Итак, нужно, чтобы функция возвращала информацию вызвавшему ее коду.

МОЗГОВОЙ ШТУРМ

Каким образом можно получить данные от функции?

Возврат данных

Чтобы заставить функцию вернуть данные, используется ключевое слово `return`. После него указывается, какие именно данные вы хотите получить.

Оператор return позволяет получить данные от функции.

Ключевое слово return указывает на то, что функция возвращает значение.

Какое именно значение будет возвращено, зависит от вашего выбора.

Положение ключевого слова `return` в теле функции теоретически может быть любым; но следует учитывать, что после выполнения этого оператора функция прекращает свою работу. Другими словами, данный оператор не только возвращает данные, но и прерывает работу функции. Например, функция `getTemp()` завершается после возвращения прочитанной с дисплея текущей температуры.

```
function getTemp() {

  // Считывание и преобразование текущей температуры

  var rawTemp = readSensor();

  var actualTemp = convertTemp(rawTemp);

  return actualTemp;

}
```

На дисплее данные отображаются в странном формате, и их требуется преобразовать в градусы.

Благодаря оператору return функция возвращает значение текущей температуры.

Если помните, функция `getTemp()` уже использовалась в сценарии работы нашего термостата:

```
function heat(targetTemp) {

  while (getTemp() < targetTemp) {

    // Начинаем нагрев

    ...

  }

}
```

Функция getTemp() предоставляет значение, используемое при проверке условия работы цикла while в функции heat().

Возвращаемое значение появляется на месте вызова функции.

Значение, возвращаемое функцией `getTemp()`, появляется **вместо** вызова этой функции и становится частью проверки условия в цикле `while`.

Возвращаемые значения

Так как оператор `return` приводит к завершению функции, его можно использовать для управления процессом работы. В общем случае результатом применения функции считается именно возвращенное ей значение. Рассмотрим функцию `heat()`:

```
function heat(targetTemp) {
  if (getTemp() >= targetTemp)
    return false;
  while (getTemp() < targetTemp) {
    // Осуществляется нагрев
    ...
  }
  return true;
}
```

Помните переменную actualTemp? Именно благодаря ей функция getTemp() возвращает значение.

Если нагрев не требуется, возвращаем значение false и завершаем работу функции.

Этот код выполняет нагрев, влияя, тем самым, на температуру и на значение, возвращаемое функцией getTemp().

Нагрев завершен, поэтому возвращается значение true.

То есть при помощи возвращаемого логического значения можно управлять работой функции и указывать как на успешное завершение неких операций, так и на неудачу. Для выхода из функции обычно используется оператор `return`, при этом никаких значений не возвращается. Посмотрите на еще одну версию функции `heat()`, работа которой прерывается оператором `return`, если нагрев не требуется.

```
function heat(targetTemp) {
  if (getTemp() >= targetTemp)
    return;
  while (getTemp() < targetTemp) {
    // Осуществляется нагрев
    ...
  }
}
```

Оператор return завершает работу функции на этом этапе, так как нагрев в этом случае не требуется.

Функция завершает работу без помощи оператора return.

Оператор return может использоваться для завершения работы функции.

Оператор return о себе

Интервью недели: Секреты мастера прерывания функций

Head First: Я слышал, что вы способны найти выход из любой ситуации.

Return: Именно так. Поместите меня в тело любой функции, и я немедленно оттуда выйду. И заберу с собой данные.

Head First: И куда вы после этого направитесь?

Return: Не забывайте, что функция вызывается каким-то кодом. Так что выход из функции означает просто возвращение в этот код. Там же оказываются и возвращенные функцией данные.

Head First: И как же это все работает?

Return: Представьте, что вызов функции — это выражение, имеющее некий результат. Если функция не возвращает данных, результат оказывается нулевым.

Head First: Если функцию можно представить в виде выражения, значит, можно назначить возвращенное ей значение какой-нибудь переменной. Я прав?

Return: И нет, и да. Выражением является не сама по себе функция, а ее вызов. И именно вызов функции можно поместить в код таким образом, что возвращенное значение будет назначено переменной.

Head First: А что происходит с выражением, если функция не возвращает значения?

Return: Если для выхода из функции используюсь я, она не возвращает данных, и выражение не имеет никакого значения.

Head First: А это не приводит к проблемам?

Return: Нет. Заботиться о том, что делать с возвращаемым значением, следует только при наличии этого значения. Если функция ничего не возвращает, то и заботиться не о чем.

Head First: Понятно. Тогда давайте поговорим о ваших способностях. Зачем вообще нужно прерывать функции? Почему не дать им завершаться естественным путем?

Return: Тот факт, что в теле функции имеется набор строк кода, не означает, что все они должны быть выполнены. О функциях в принципе не имеет смысла думать как о чем-то, имеющем начало и конец. «Естественный» конец может располагаться в середине помещенного в ее тело кода. Вот тут-то на помощь прихожу я.

Head First: То есть вы утверждаете, что нормально иметь в теле функции код, который, может быть, никогда не будет запущен?

Return: Я бы сказал, что существует более одного способа создать тело функции, и именно это я помогаю сделать. Если происходит нечто, указывающее на невозможность продолжения работы функции, я немедленно ее завершаю. В то же самое время достаточно случаев, когда тело функции выполняется от первой до последней строчки, и я там появляюсь в лучшем случае, чтобы вернуть данные.

Head First: Другими словами, вы позволяете как возвращать данные, так и управлять выполнением функций?

Return: Да, именно так!

Head First: Потрясающе. Спасибо, что согласились с нами побеседовать.

Return: Всегда пожалуйста. А теперь мне нужно отсюда выйти!

Упражнение

Кажется, JavaScript попал в эпицентр скандала вокруг климатических изменений. Люди, выступающие за прохладную атмосферу, создали сценарий, распространяющий воззвание против потепления. Но их оппоненты, уставшие от холодов, внесли в код свою лепту, и сообщение перестало появляться. Вам нужно вернуть сценарию первоначальный вид и узнать, что же хотели сказать противники нагрева.

```
function showClimateMsg() {
  return;
  alert(constructMessage());
}

function constructClimateMsg() {
  var msg = "";

  msg += "Глобальное "; // "Локальное ";

  if (getTemp() > 80)
    msg += "потепление ";
  else
    msg += "похолодание ";

  if (true)
    msg += "не ";
  else
    msg += "это ";

  if (getTemp() <= 70)
    return msg + "обман!";
  else
    return msg + "правда!";

  return "Я в это не верю.";
}

function getTemp() {
  // Определяет текущую температуру
  var actualTemp = readSensor();
  return 64;
}
```

Упражнение Решение

Вот какие исправления внесли в код сторонники локального нагрева, чтобы не дать появиться сообщению от группы, выступающей против потепления.

Оператор return не дает появиться всплывающему окну.

С этим кодом все в порядке, так как появляющееся сообщение будет зависеть от текущей температуры.

Из-за оператора if-else эта строчка никогда не будет выполняться, поэтому писать ее не имеет смысла.

```
function showClimateMsg() {
  return;
  alert(constructMessage());
}

function constructClimateMsg() {
  var msg = "";

  msg += "Глобальное "; // "Локальное ";

  if (getTemp() > 80)
    msg += "потепление ";
  else
    msg += "похолодание ";

  if (true)
    msg += "не ";
  else
    msg += "это ";

  if (getTemp() <= 70)
    return msg + "обман!";
  else
    return msg + "правда!";

  return "Я в это не верю.";
}

function getTemp() {
  // Определяет текущую температуру
  var actualTemp = readSensor();
  return 64; actualTemp;
}
```

Исходный текст был превращен в комментарий, а вместо него вписан другой вариант.

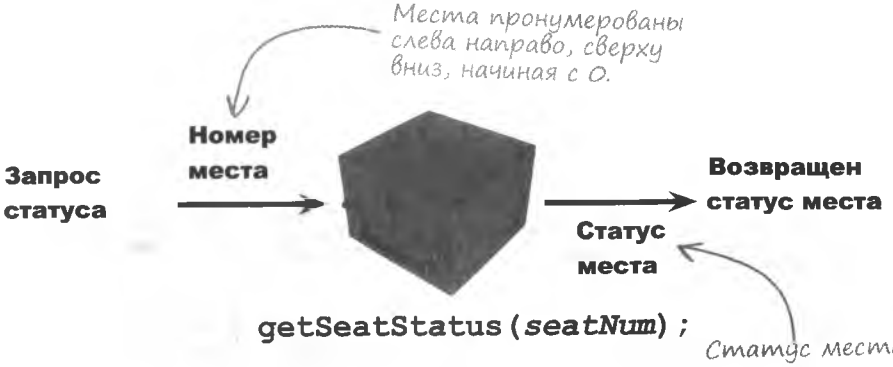
Подобное условие не имеет смысла.

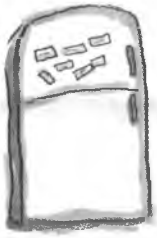
Возвращаем температуру, считанную с экрана.

Локальное потепление — это правда!

Информация о статусе места

Сета и Джейсона уже тошнит от разговоров об окружающей температуре, поэтому вернемся к программе Mandango. Часть пользователей жалуется, что по цвету картинки сложно определить состояние места. Поэтому им хотелось бы иметь возможность получать данную информацию, например, по щелчку. Словом, программе наших мачо требуется еще одна функция.





Функция getSeatStatus( )

Функции `getSeatStatus()` не хватает кода, позволяющего определить статус выделенного места. Функция сначала проверяет, относится ли указанное место к набору из трех, выбранных пользователем. В случае отрицательного результата проверяется, свободно это место или занято. Расставьте магниты, чтобы получить недостающие фрагменты.

```
function getSeatStatus(seatNum) {
  if (.....!= -1 &&
    (.....==.....||.....==(.....+ 1) ||.....==(.....+ 2)))
    return "yours";
  else if (.....[Math.floor(...../.....[0].length)][.....%.....[0].length])
    return "available";
  else
    return "unavailable";
}
```

seats

selSeat

seatNum

Решение задачи с магнитами

Вот как стал выглядеть код функции `getSeatStatus()` после подстановки магнитов.

Если место уже выбрано, глобальная переменная selSeat имеет значение –1.

Так как места выбираются по три, мы проверяем не только выбранное место, но и два следующих за ним.

```
function getSeatStatus(seatNum) {
  if ( selSeat != -1 &&
    ( seatNum == selSeat || seatNum == ( selSeat + 1) || seatNum == ( selSeat + 2)))
    return "yours";
  else if ( seats [Math.floor( seatNum / seats [0].length)][ seatNum % seats [0].length])
    return "available";
  else
    return "unavailable";
}
```

Чтобы определить номер ряда, разделим номер места на число мест в ряду и округлим результат до целого.

Чтобы узнать второй индекс массива, поделите номер кресла на количество мест в ряду.

Здесь можно было поставить цифру 9, но если в будущем вы решите поменять число мест в ряду, программа перестанет корректно работать.

Отображение статуса

Нужно, чтобы по щелчку пользователя на изображении кресла появлялась информация о статусе этого кресла. Для этого нам потребуется функция `showSeatStatus()`. Впрочем, вся основная работа будет выполнена только что написанной функцией `getSeatStatus()`.

Чтобы узнать статус места, передайте его номер функции getSeatStatus().

```
function showSeatStatus(seatNum) {
  alert("This seat is " + getSeatStatus(seatNum) + ".");
}
```

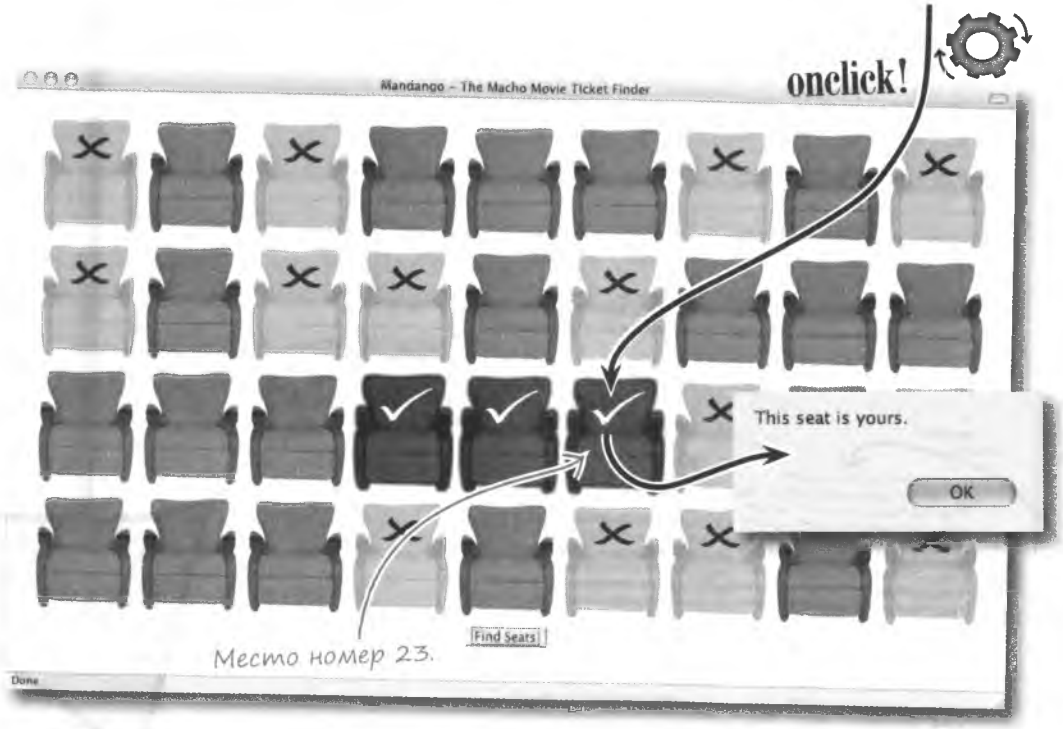
Сообщение о состоянии места формируется путем соединения строк.

Связь функции с изображением

Связав новую функцию с изображениями мест на веб-странице Mandango, вы дадите пользователям возможность получать информацию о состоянии мест, щелкнув на изображении выбранного кресла. Событие `onclick` каждого из изображений должно вызывать функцию `showSeatStatus()`:

В данном случае функция showSeatStatus() вызвана щелчком на изображении "seat23".

```
<img id="seat23" src="" alt="" onclick="showSeatStatus(23);" />
```



Таким образом, после щелчка на изображении кресла появляется окно диалога с информацией о статусе этого кресла. Данный вариант подходит для пользователей, которые не хотят ориентироваться по цветам изображений, предпочитая прочитать, занято место или свободно.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Оператор `return` заставляет функцию вернуть данные в вызвавший ее код.
- Возвращенные данные подставляются вместо кода, вызвавшего функцию.
- Функция может вернуть только один фрагмент данных.
- Оператор `return` используется также для прерывания работы функций.

Дублирующийся код

Сценарий Mandango работает вполне корректно, но парни стали задумываться о долгосрочных перспективах. В частности, Джейсон начал исследование и обнаружил, что в современных веб-приложениях выгодно разделять код HTML, JavaScript и CSS.

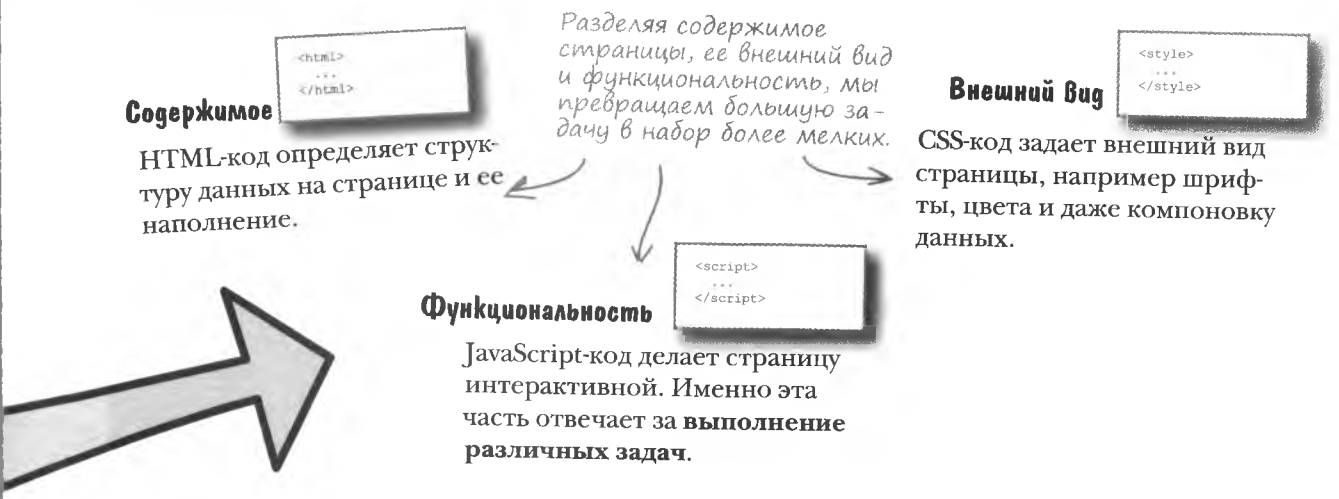

```
<html>
  <head>
    <title>Mandango - The Macho Movie Ticket Finder</title>

    <script type="text/javascript">
      ...

      function initSeats() {
        ...       }

      function getSeatStatus(seatNum) {
        ...
      }

      function showSeatStatus(seatNum) {
        alert("This seat is " + getSeatStatus(seatNum) + ".");
      }

      function setSeat(seatNum, status, description) {
        document.getElementById("seat" + seatNum).src = "seat_" + status + ".png";
        document.getElementById("seat" + seatNum).alt = description;
      }

      function findSeats() {
        ...
      }
    </script>
  </head>

  <body onload="initSeats();">
    <div style="margin-top:25px; text-align:center">
      <img id="seat0" src="" alt="" onclick="showSeatStatus(0);" />
      <img id="seat1" src="" alt="" onclick="showSeatStatus(1);" />
      <img id="seat2" src="" alt="" onclick="showSeatStatus(2);" />
      <img id="seat3" src="" alt="" onclick="showSeatStatus(3);" />
      <img id="seat4" src="" alt="" onclick="showSeatStatus(4);" />
      <img id="seat5" src="" alt="" onclick="showSeatStatus(5);" />
      <img id="seat6" src="" alt="" onclick="showSeatStatus(6);" />
      <img id="seat7" src="" alt="" onclick="showSeatStatus(7);" />
      <img id="seat8" src="" alt="" onclick="showSeatStatus(8);" /><br />
      <img id="seat9" src="" alt="" onclick="showSeatStatus(9);" />
      <img id="seat10" src="" alt="" onclick="showSeatStatus(10);" />
      <img id="seat11" src="" alt="" onclick="showSeatStatus(11);" />
      <img id="seat12" src="" alt="" onclick="showSeatStatus(12);" />
      <img id="seat13" src="" alt="" onclick="showSeatStatus(13);" />
      <img id="seat14" src="" alt="" onclick="showSeatStatus(14);" />
      <img id="seat15" src="" alt="" onclick="showSeatStatus(15);" />
      <img id="seat16" src="" alt="" onclick="showSeatStatus(16);" />
      <img id="seat17" src="" alt="" onclick="showSeatStatus(17);" /><br />
      <img id="seat18" src="" alt="" onclick="showSeatStatus(18);" />
      <img id="seat19" src="" alt="" onclick="showSeatStatus(19);" />
      <img id="seat20" src="" alt="" onclick="showSeatStatus(20);" />
      <img id="seat21" src="" alt="" onclick="showSeatStatus(21);" />
      <img id="seat22" src="" alt="" onclick="showSeatStatus(22);" />
      <img id="seat23" src="" alt="" onclick="showSeatStatus(23);" />
      <img id="seat24" src="" alt="" onclick="showSeatStatus(24);" />
      <img id="seat25" src="" alt="" onclick="showSeatStatus(25);" />
      <img id="seat26" src="" alt="" onclick="showSeatStatus(26);" /><br />
      <img id="seat27" src="" alt="" onclick="showSeatStatus(27);" />
      <img id="seat28" src="" alt="" onclick="showSeatStatus(28);" />
      <img id="seat29" src="" alt="" onclick="showSeatStatus(29);" />
      <img id="seat30" src="" alt="" onclick="showSeatStatus(30);" />
      <img id="seat31" src="" alt="" onclick="showSeatStatus(31);" />
      <img id="seat32" src="" alt="" onclick="showSeatStatus(32);" />
      <img id="seat33" src="" alt="" onclick="showSeatStatus(33);" />
      <img id="seat34" src="" alt="" onclick="showSeatStatus(34);" />
      <img id="seat35" src="" alt="" onclick="showSeatStatus(35);" /><br />
      <input type="button" id="findseats" value="Find Seats" onclick="findSeats();" />
    </div>
  </body>
</html>
```

Смесь JavaScript и HTML может быть изолирована в обработчиках событий HTML-атрибутов.

На веб-странице Mandango перемешаны между собой JavaScript и HTML.

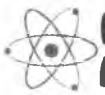
Отделите функциональность от содержимого

Чем же так плох смешанный код? В конце концов, он ведь работает. Проблемы начинают вырисовываться при попытке посмотреть на веб-страницы с JavaScript не как на страницы, а как на **приложения**. Для долговременной работы любого приложения важны аккуратное планирование и структура. Разделив внешний вид, содержимое и функциональность страницы, вы избежите множества ошибок и получите более удобный для редактирования код. Проверим справедливость этого утверждения на примере программы Mandango.



Подумайте, как реализовать разделение кода. Представим, что Сет и Джейсон нашли удачный алгоритм поиска мест, который они предпочли бы использовать вместо старой версии. Значит, требуется отредактировать приложение Mandango, но, так как код JavaScript переплетен с кодом HTML, в процессе исправлений можно повредить структуру страницы. Насколько удобнее было бы, если бы HTML-код был изолирован и его связь с JavaScript осуществлялась бы исключительно средствами последнего.

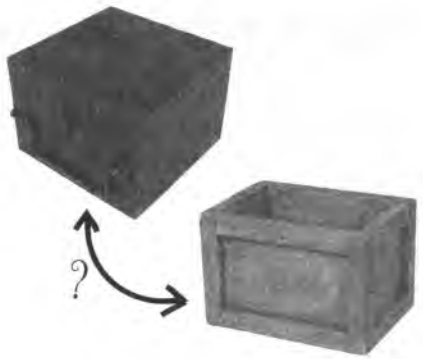
Отделение функциональности от содержимого облегчает последующее редактирование веб-приложений.



МОЗГОВОЙ ШТУРМ

Каким образом функции могут помочь нам отделить содержимое от функциональности в случае Mandango?

Функции — это тоже данные



Для эффективного разделения кода нужно понять, каким образом функции связаны с событиями; пока что мы осуществляли эту связь при помощи HTML-атрибутов. Но существует и другой способ, который более предпочтителен с точки зрения задачи разделения кодов JavaScript и HTML. Давайте посмотрим на функции с другой точки зрения.

Как ни удивительно, но функция — это не более чем переменная. Ее тело представляет собой значение, а имя — имя переменной. Вот как мы привыкли воспринимать функции:

Функция, созданная обычным способом.

```
function showSeatStatus(seatNum) {
  alert("This seat is " + getSeatStatus(seatNum) + ".");
}
```

А вот та же самая функция, созданная другим способом.

В качестве имени функции выступает имя переменной.

```
var showSeatStatus = function(seatNum) {
  alert("This seat is " + getSeatStatus(seatNum) + ".");
};
```

В роли значения переменной выступает тело функции. В подобных конструкциях оно называется литералом функции.

Этот код показывает процесс создания функции на примере синтаксиса, используемого для создания переменной. Процедура состоит из одних и тех же частей: уникальный идентификатор (имя функции) плюс его значение (тело функции). Тело функции, фигурирующее само по себе, без имени, называется **литералом функции**.

Другими словами, функциями можно управлять так же, как и переменными. Как вы думаете, что делает следующий код?

```
var myShowSeatStatus = showSeatStatus;
```

Функция showSeatStatus() назначена переменной myShowSeatStatus.

Вызов функции и ссылка на нее

Назначив имя функции другой переменной, вы даете этой переменной доступ к телу функции. То есть вы можете написать следующий код:

```
alert(myShowSeatStatus(23));
```

Вызов той же самой функции через переменную myShowSeatStatus.

Результат вызова переменной `myShowSeatStatus()` и функции `showSeatStatus()` идентичен, так как в обоих случаях происходит **ссылка** на один и тот же код. Соответственно, ссылкой на функцию является ее имя.

Функция — это такая переменная, значение которой является ссылкой на ее тело.

В чем же разница между ссылкой на функцию и ее вызовом? **Ссылка на функцию** фигурирует сама по себе, в то время как после **вызова функции** ставятся скобки, часто с набором аргументов.

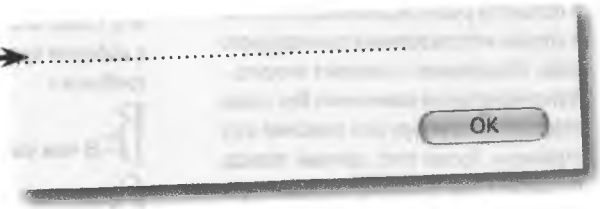
Вызовем функцию myShowSeatStatus(), что эквивалентно вызову showSeatStatus().

```
var myShowSeatStatus = showSeatStatus;
myShowSeatStatus(23);
```

Сделаем переменную myShowSeatStatus ссылкой на функцию.

Проанализируйте данный ниже код и определите, какое число появится в окне.

```
function doThis(num) {
  return num++;
}

function doThat(num) {
 return num--;
}

var x = doThis(11);
var y = doThat;
var z = doThat(x);
x = y(z);
y = x;
alert(doThat(z - y));
```

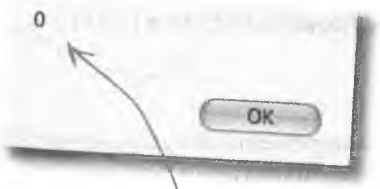


Упражнение Решение

Вот как работает данный код и к какому результату он приводит.

```
function doThis(num) {
  return num++;
}

function doThat(num) {
 return num--;
}

var x = doThis(11);
var y = doThat;
var z = doThat(x);
x = y(z);
y = x;
alert(doThat(z - y));
```

```
x = 12
y = doThat
z = doThat(12) = 11
x = doThat(11) = 10
y = 10
alert(doThat(11 - 10))
```



Часто Задаваемые Вопросы

В: Неужели разделение программы на части настолько важно?

О: И да, и нет. Для простых приложений это не обязательно. Почувствовать преимущества от разделения HTML, CSS и JavaScript-кода можно только в сложных, многострочных приложениях. Чем длиннее программа, тем сложнее ею управлять и тем проще допустить ошибку в процессе редактирования, особенно в случае использования разнородного кода. Разделение позволяет вносить функциональные изменения без риска повредить структуру или внешний вид страницы. Кроме того, данный подход позволяет веб-дизайнерам редактировать структуру и внешний вид страницы, не боясь случайно внести изменения в код JavaScript, которого они не понимают.

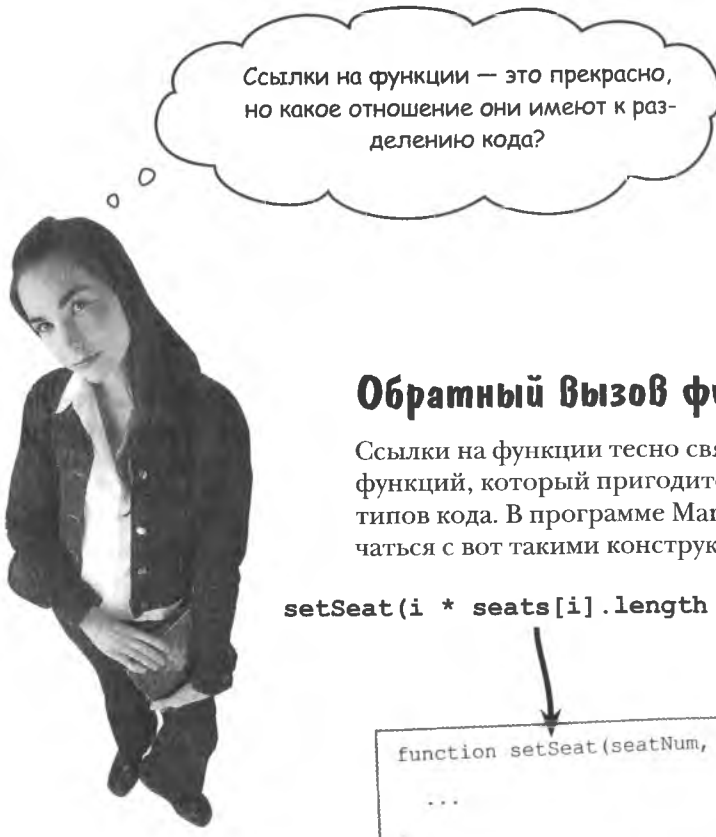
В: А чем функция отличается от обычной переменной?

О: Разница заключается в том, каким образом вы можете поступить с данными. Данные, связанные с функцией, можно запустить на выполнение. Вы указываете это, ставя за именем функции скобки с набором аргументов, если таковые требуются.

В: В чем смысл ссылки на функцию?

О: В отличие от обычной переменной, которая хранит данные в виде значения в памяти, функции сохраняют ссылку на свой код. То есть значение функции — это не сам по себе код, а ссылка на место в памяти, где этот код хранится. Точно так же, как почтовый адрес на конверте является только ссылкой на ваш дом, а не вашим домом.

Ссылки на функции используются в целях эффективности. Ведь проще хранить ссылку, чем набор копий одного и того же кода. Поэтому, назначая функцию обработчику событий (именно этим вы займетесь через пару секунд), вы на самом деле назначаете только ссылку на код функции.



Обратный вызов функций

Ссылки на функции тесно связаны с особым способом вызова функций, который пригодится нам при разделении различных типов кода. В программе Mandango вам уже приходилось встречаться с вот такими конструкциями.

```
setSeat(i * seats[i].length + j, "select", "Your seat");
```

```
function setSeat(seatNum, status, description) {
  ...
}
```

Но это не единственный способ вызова функций. Существует еще и так называемая процедура **обратного вызова**, не требующая вашего непосредственного участия.

МОЗГОВОЙ ШТУРМ

Каким образом вы можете использовать обратный вызов функции в Mandango?

Беседа у камина

Обычная функция против функции обратного вызова.

Обычная функция:

Я много слышала о том, что ты не приемлешь локальных вызовов. Интересно, почему?

Ты имеешь в виду браузеры? Тоже мне экзотика. Мне кажется, ты просто задираешь нос перед теми, кто общается с кодом сценария обычным способом.

Да что ты говоришь! Да кого волнует происходящее за пределами сценария?

Возможно, ты и права. Мне нравится узнавать о том, что страница уже загрузилась, что пользователь щелкнул кнопкой мыши или ввел текст. Так ты говоришь, что мы не знали бы об этом, если бы не ты?

Было приятно узнать, что мы не так уж различаемся, как это сначала казалось.

Не обращайся ко мне, я сама тебя вызову.

Функция обратного вызова:

Да потому, что у меня другая цель. Мне нравится, когда меня вызывают из экзотических мест.

Послушай, мы же не говорим о том, кто лучше, а кто хуже. Мы все являемся частью сценария, просто я предоставляю доступ к коду снаружи. Без меня невозможно узнать, что происходит за пределами сценария.

Всех и каждого. Не забывай, что сценарии создаются для удобства пользователей. Не имея возможности отследить происходящее снаружи, практически невозможно улучшить пользовательский интерфейс.

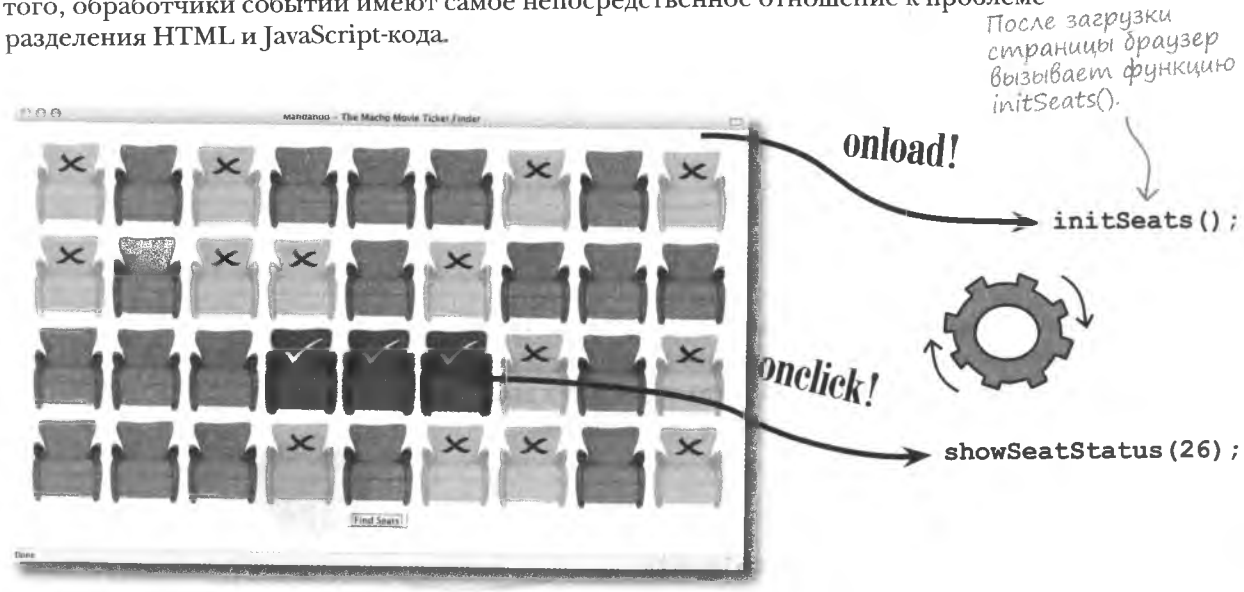
Именно так. Браузер вызывает меня, и во многих случаях уже я вызываю тебя. Ведь ответ на внешние события часто требует включения нескольких функций.

Да. Ты смотришь в корень.

Ну попробуй.

События, обратный вызов и атрибуты HTML

Вам уже приходилось встречаться с функциями, которые вызываются не вашим кодом, а браузером. Самым распространенным примером такой функции является обработчик событий. Именно с его помощью работает программа Mandango. Более того, обработчики событий имеют самое непосредственное отношение к проблеме разделения HTML и JavaScript-кода.



Данные функции обратного вызова связаны с событиями в HTML-коде страницы Mandango.

```
<body onload="initSeats();">

<img id="seat26" src="" alt="" onclick="showSeatStatus(26);" />
```

HTML-атрибут onload связывает функцию initSeats() с событием onload.

HTML-атрибут onclick связывает функцию showSeatStatus() с событием onclick.

К сожалению, прекрасно работающее связывание обработчиков событий с событиями при помощи HTML-атрибутов невозможно без взаимопроникновения JavaScript- и HTML-кода. Положение можно спасти при помощи ссылок на функции.

Ссылки на функции

Для связывания функции обратного вызова с событием можно не использовать HTML-атрибуты, а назначить ссылку на функцию непосредственно коду JavaScript. В результате вам вообще не придется иметь дела с HTML-кодом – просто воспользуйтесь ссылкой на функцию.

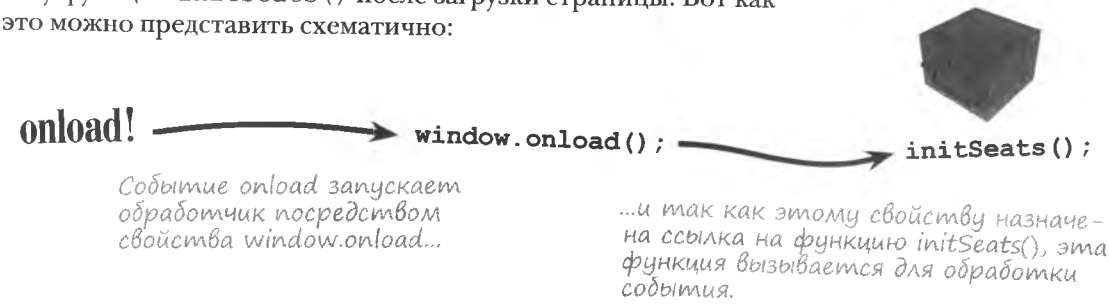
```
window.onload = initSeats;
```

После имени функции нет скобок, ведь мы не собираемся ее запускать, мы всего лишь ссылаемся на нее.

Событие onload является свойством объекта window.

Ссылка на функцию initSeats() назначена свойству события onload.

Чтобы связать обработчик событий исключительно с JavaScript-кодом, нужно назначить ссылку на функцию свойству события объекта. В случае с нашим событием `onload` это приводит к вызову функции `initSeats()` после загрузки страницы. Вот как это можно представить схематично:



Именно эта техника позволяет четко разделить JavaScript- и HTML-код – ведь вам уже не требуется назначать JavaScript-код HTML-атрибутам.

```
<body onload="initSeats();">
<body>
```

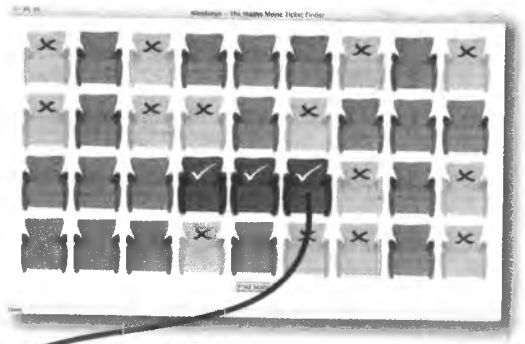
Теперь `<body>` не содержит ничего лишнего. Осталось только заставить код обработки события запускаться как можно раньше, именно поэтому он обычно помещается в заголовок страницы.

Но как же быть в ситуации, когда обработчику события нужно передать аргумент? Для события `onload` в Mandango такой проблемы не существует, а вот событию `onclick` нужно передать номер кресла, которое выбрал пользователь. Ссылки на функции не позволяют решить данную задачу...

Ссылки на функции позволяют легко связать обработчик события с самим событием.

Литерал функции

Событие `onlick`, связанное с изображениями кресел, должно вызывать функцию `showSeatStatus()` с номером кресла в качестве аргумента. Просто назначив ему ссылку на функцию, мы не сможем передать аргумент. Поэтому ссылаться в данном случае следует на литерал функции, из которого и будет осуществлен вызов.



Так мы получаем доступ к свойству onclick объекта с изображением сиденья.

```
document.getElementById("seat26").onclick = function(evt) {

  showSeatStatus(26);

};
```

Литерал функции служит как бы «контейнером» для вызова функции showSeatStatus(), позволяя передавать в нее аргумент.

Литерал функции назначен свойству события onclick как ссылка на функцию.

Объект event автоматически передается обработчику события как первый аргумент.

Литерал используется в качестве «оболочки» для вызова функции `showSeatStatus()`, и именно он позволяет передать номер кресла. Его можно представить в виде безымянной функции, обрабатывающей события. Именно по этой причине литералы иногда называют **анонимными функциями**.

Код показывает, каким образом в JavaScript представлен объект, передаваемый обработчику, в данном случае посредством аргумента `evt`. Этот объект содержит информацию о событии.

Литералы функции позволяют создавать анонимные обработчики событий.

Упражнение

Свяжите функцию `initSeats()` с событием `onload`, используя вместо ссылки литерал.

.....

.....

.....

Упражнение Решение

Вот каким образом функция `initSeats()` связывается с событием `onload` при помощи литерала.

```
window.onload = function(evt) {
  initSeats();
};
```

Функция initSeats() вызывается внутри обработчика события onload.

Аргумент evt игнорируется, так как обработчику события onload не требуется объект event.

А где же связывание?

По-прежнему остался нерешенным вопрос о связи событий с литералами функций. Мы увидели, что назначение обработчика события `onload` может осуществляться в заголовке страницы внутри тега `<script>`. В результате связанный с этим событием код не запускается до загрузки страницы. Точно такой же результат мы получали, назначая функцию `initSeats()` HTML-атрибуту `onload` в теге `<body>`. Но в каком месте связываются со своими обработчиками остальные события?

Это может происходить в функции обратного вызова обработчика события `onload`.

Обработчик события onload прекрасно подходит для инициализации всех остальных событий.

```
window.onload = function() {
  // Свяжем остальные события здесь
  ...
  // Задаем вид кресел
  initSeats();
};
```

Все остальные события страницы могут быть связаны внутри обработчика события onload.

Код, связанный с событием onload, до сих пор запускается внутри обработчика этого события.

Таким образом, именно обработчик события `onload` становится функцией, инициализирующей все прочие события. Это удобно, так как в результате запуск соответствующих фрагментов кода осуществляется после загрузки страницы.

Часто Задаваемые Вопросы

В: Почему функциям обратного вызова придается такое значение?

О: Функции обратного вызова позволяют реагировать на события, происходящие за пределами сценария. Для их запуска требуется, чтобы произошли определенные события. Узнать о том, что нужное событие произошло, функции обратного вызова позволяет браузер. Для этого нужно связать функцию с пусковым механизмом, роль которого обычно играют различные события.

В: Отличаются ли функции обратного вызова от обработчиков событий?

О: Да. В главе 12 вы познакомитесь с другим способом применения функций обратного вызова, в процессе которого они обрабатывают данные, посланные сервером в ответ на запрос Ajax.

В: Я так и не понял, почему важны литералы функции.

О: Литералом называется тело функции без имени. Литералы идеально подходят для тех ситуаций, когда функция обратного вызова вызывается только один раз. То есть когда функция вызывается один раз и не вашим кодом. Литералы назначаются непосредственно свойствам события, в то время как на именованные функции в таких ситуациях делается ссылка. Фактически это просто более эффективный способ работы в случаях, когда вам не требуются именованные функции. Кроме того, именно литералы приходят на помощь, если нужно не только сослаться на функцию, но и передать ей аргумент.

Возьми в руку карандаш

Впишите недостающий код в новый обработчик события `onload` программы Mandango.

```
window.onload = function() {
  // Связываем со щелчком на кнопке Find Seat
  document.getElementById("findseats")...... = .....;

  // Связываем со щелчками на изображениях кресел
  document.getElementById("seat0")...... = function(evt) { ..... };
  document.getElementById("seat1")...... = function(evt) { ..... };
  document.getElementById("seat2")...... = function(evt) { ..... };
  ...

  // Задаем вид кресел
  .....
};
```

Возьми в руку карандаш

Решение

Вот как выглядит код нового обработчика события `onload` в программе Mandango.

Обработчик события onload является анонимной функцией.

Функция findSeats() связана с событием onclick при помощи ссылки.

```
window.onload = function() {

  // Связываем со щелчком на кнопке Find Seat
  document.getElementById("findseats").onclick = findSeats;


  // Связываем со щелчками на изображениях кресел
  document.getElementById("seat0").onclick = function(evt) { showSeatStatus(0); };
  document.getElementById("seat1").onclick = function(evt) { showSeatStatus(1); };
  document.getElementById("seat2").onclick = function(evt) { showSeatStatus(2); };
  ...


  // Задаем вид кресел
  initSeats();
};
```

Функция initSeats() вызывается для завершения задач, связанных с загрузкой.

Свойство onlick для каждого из изображений задает код, запускаемый при щелчке.

Функция showSeatStatus() вызывается из литерала, поэтому ей можно передать аргумент.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Функции обратного вызова вызываются браузером в ответ на события **вне пределов сценария**.
- Ссылки позволяют **назначать функции**, как если бы они были **переменными**.
- Ссылки позволяют связать функции обработчиков событий с кодом JavaScript, **не затрагивая HTML**.
- Литералами называются **безымянные функции**.

Часто Задаваемые Вопросы

В: Почему обработчик события `onload` в Mandango создан как литерал функции?

О: Потому что у нас нет причин использовать в этом месте именованную функцию. Наша функция вызывается только один раз в ответ на событие `onload`. Можно было создать именованную функцию и назначить ссылку на нее `window.onload`, но связь между функцией и событием становится более прозрачной, если их связать друг с другом при помощи литерала.

В: Нужно ли связывать с обработчиком события `onload` все остальные функции обратного вызова?

О: Да. Вы можете решить, что лучше связать их непосредственно в теге `<script>`. Но вспомните о том, что на этом этапе содержимое страницы еще не загружено. Соответственно, функции `getElementById()` не будут вызываться. Полную загрузку страницы гарантирует только событие `onload`, с которым следует связать нужные вам обработчики.

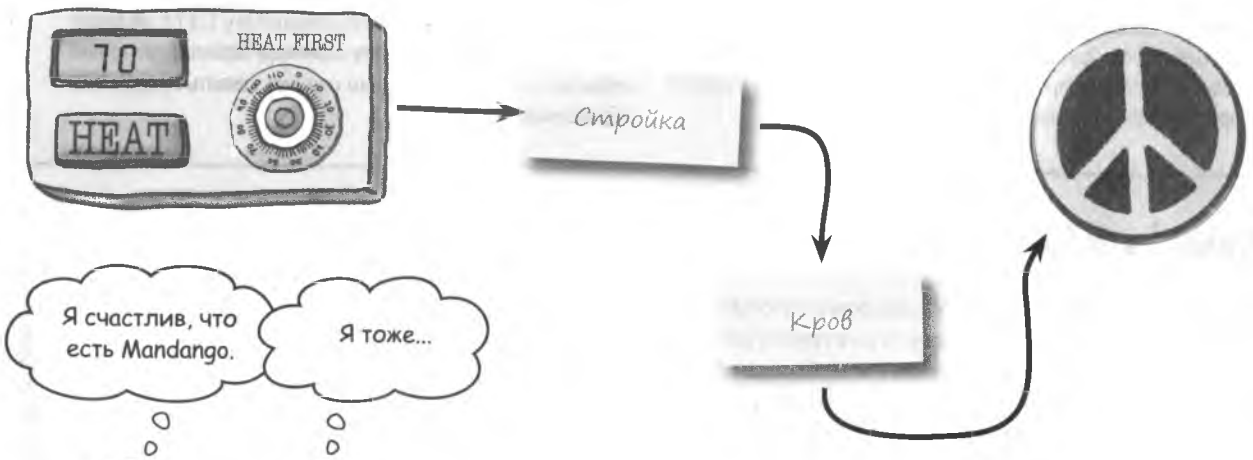
Оболочка HTML-страницы

Разделение JavaScript и HTML-кодов в программе Mandango приводит к тому, что структурная часть страницы становится совсем небольшой. В результате вы можете легко редактировать HTML, не боясь случайно внести изменения в код JavaScript.

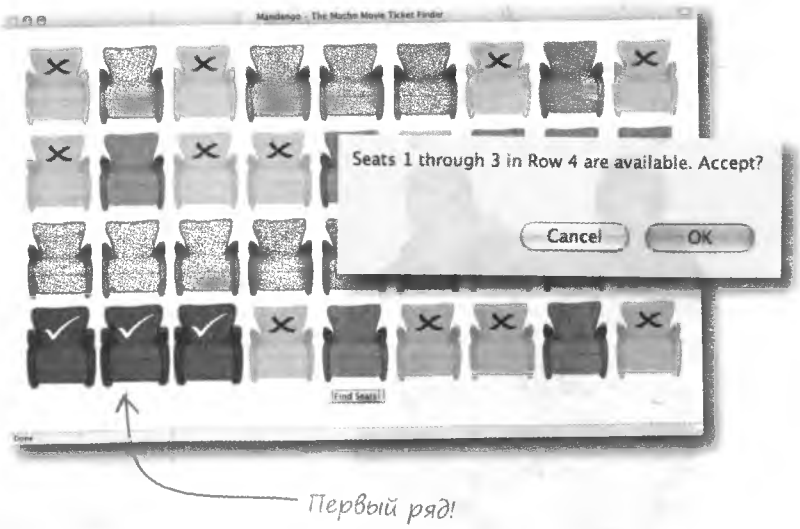

```
<body>
  <div style="margin-top:75px; text-align:center">
    <img id="seat0" src="" alt="" />
    <img id="seat1" src="" alt="" />
    <img id="seat2" src="" alt="" />
    <img id="seat3" src="" alt="" />
    <img id="seat4" src="" alt="" />
    <img id="seat5" src="" alt="" />
    <img id="seat6" src="" alt="" />
    <img id="seat7" src="" alt="" />
    <img id="seat8" src="" alt="" />
    <img id="seat9" src="" alt="" /><br />
    <img id="seat10" src="" alt="" />
    <img id="seat11" src="" alt="" />
    <img id="seat12" src="" alt="" />
    <img id="seat13" src="" alt="" />
    <img id="seat14" src="" alt="" />
    <img id="seat15" src="" alt="" />
    <img id="seat16" src="" alt="" />
    <img id="seat17" src="" alt="" />
    <img id="seat18" src="" alt="" />
    <img id="seat19" src="" alt="" /><br />
    <img id="seat20" src="" alt="" />
    <img id="seat21" src="" alt="" />
    <img id="seat22" src="" alt="" />
    <img id="seat23" src="" alt="" />
    <img id="seat24" src="" alt="" />
    <img id="seat25" src="" alt="" />
    <img id="seat26" src="" alt="" />
    <img id="seat27" src="" alt="" />
    <img id="seat28" src="" alt="" />
    <img id="seat29" src="" alt="" /><br />
    <img id="seat30" src="" alt="" />
    <img id="seat31" src="" alt="" />
    <img id="seat32" src="" alt="" />
    <img id="seat33" src="" alt="" />
    <img id="seat34" src="" alt="" />
    <img id="seat35" src="" alt="" />
    <img id="seat36" src="" alt="" />
    <img id="seat37" src="" alt="" />
    <img id="seat38" src="" alt="" />
    <img id="seat39" src="" alt="" /><br />
    <input type="button" id="findseats" value="Find Seats" />
  </div>
</body>
```

И еще немножко про JavaScript...

Добиться мира во всем мире мы не смогли, зато научились управлять температурой в комнате при помощи JavaScript. Мы узнали, как улучшить наши сценарии, деля крупные задачи на более мелкие, фокусируясь на единственной цели и создавая код, пригодный для многократного использования.



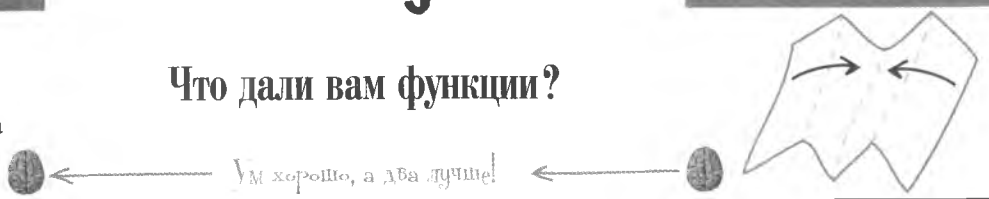
Сет и Джейсон также воспользовались данной технологией и создали более эффективную и простую в редактировании версию Mandango. По крайней мере, в души наших мачо пришло спокойствие...

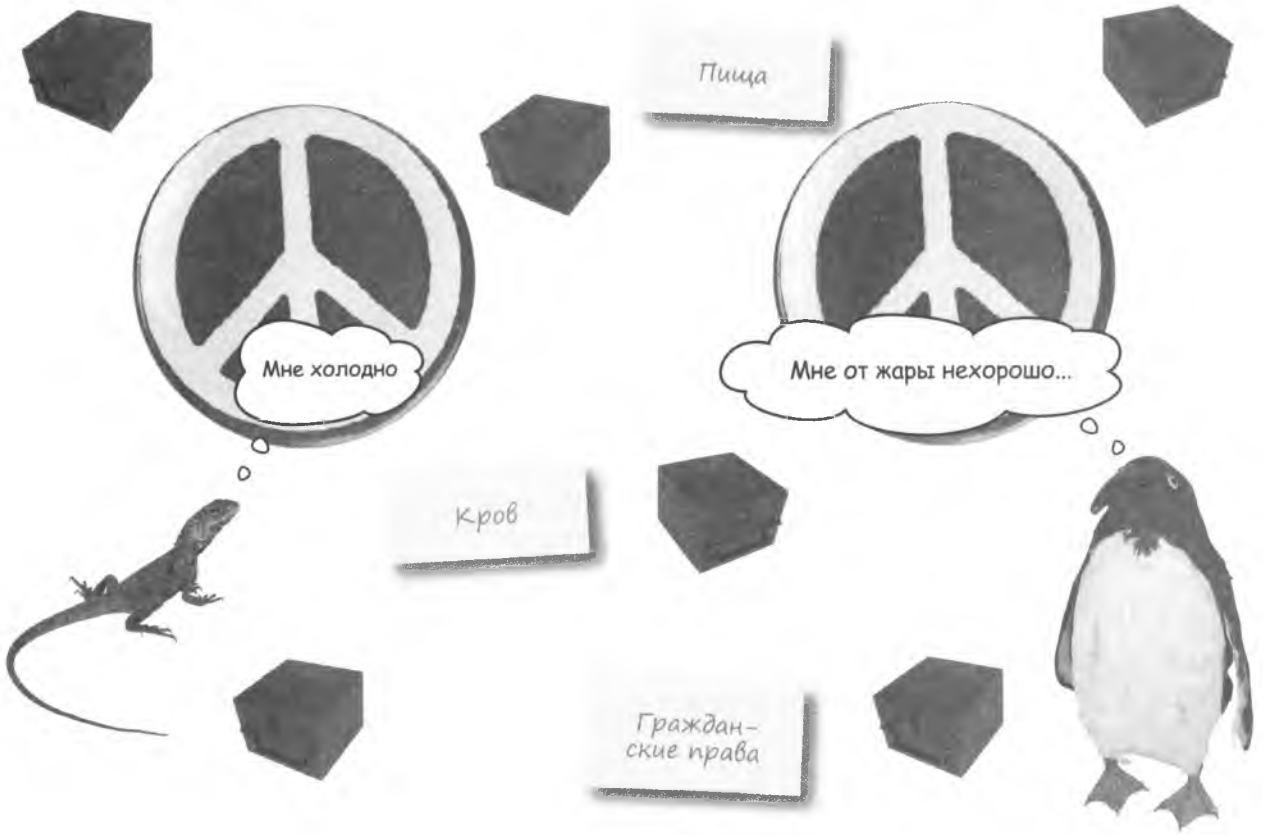


Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Что дали вам функции?





Установить мир во всем мире всегда сложно. Даже в JavaScript только самый систематизированный код даст вам спокойствие и уверенность. Комфортная жизнь дается нелегко, по крайней мере, в JavaScript.

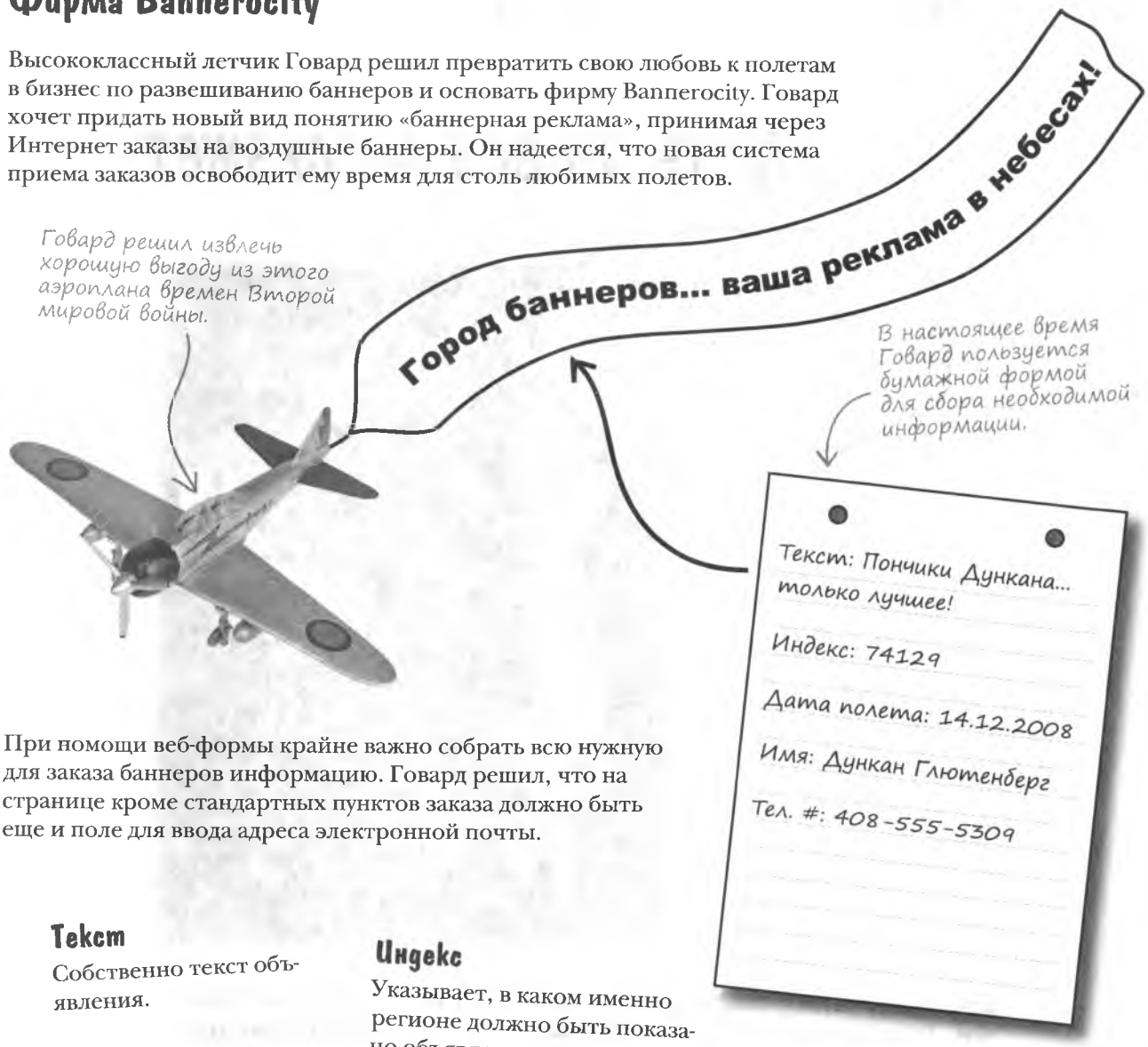
7 Формы и проверка данных

Пусть он все расскажет

Для получения информации от пользователей при помощи JavaScript вам не потребуется быть джентльменом. Но вы должны быть аккуратны. Люди часто делают ошибки, а это означает, что данные, полученные при помощи веб-форм, далеко не всегда ***корректны***. **Проверяя вводимые данные при помощи JavaScript**, вы увеличиваете **надежность веб-приложений** и снимаете дополнительную нагрузку с серверов. Полоса пропускания нам пригодится для восхитительных видеороликов и чудесных фотографий.

Фирма Bannerocity

Высококлассный летчик Говард решил превратить свою любовь к полетам в бизнес по развешиванию баннеров и основать фирму Bannerocity. Говард хочет придать новый вид понятию «баннерная реклама», принимая через Интернет заказы на воздушные баннеры. Он надеется, что новая система приема заказов освободит ему время для столь любимых полетов.



При помощи веб-формы крайне важно собрать всю нужную для заказа баннеров информацию. Говард решил, что на странице кроме стандартных пунктов заказа должно быть еще и поле для ввода адреса электронной почты.

Текст
Собственно текст объявления.

Индекс
Указывает, в каком именно регионе должно быть показано объявление.

Дата полета
Определяет, когда будет вывешен баннер.

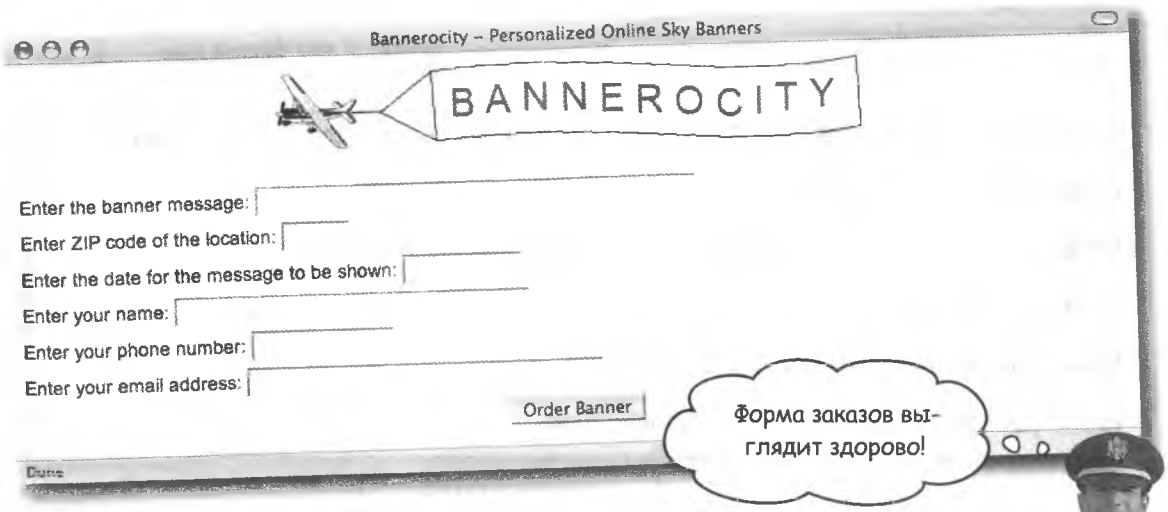
Email
Адрес электронной почты для связи с заказчиком.

Телефон
Контактная информация заказчика.

Имя
Имя заказчика.

HTML-форма

При помощи HTML первая форма Говарда приобрела вот такой вид:



Эта форма для сбора заказов имеет все необходимые поля и, кажется, готова к работе даже без JavaScript. Так ли это?

Говард столько лет отработал летчиком, что до сих пор не может расстаться с красивой формой.

Скачать код формы можно по адресу http://www.headfirstlabs.com/books/hfjs/

Возьми в руку карандаш

Давайте попробуем заполнить форму от руки. Не волнуйтесь, денег за рекламу нам платить не придется.

Enter the banner message:

Enter ZIP code of the location:

Enter the date for the message to be shown:

Enter your name:

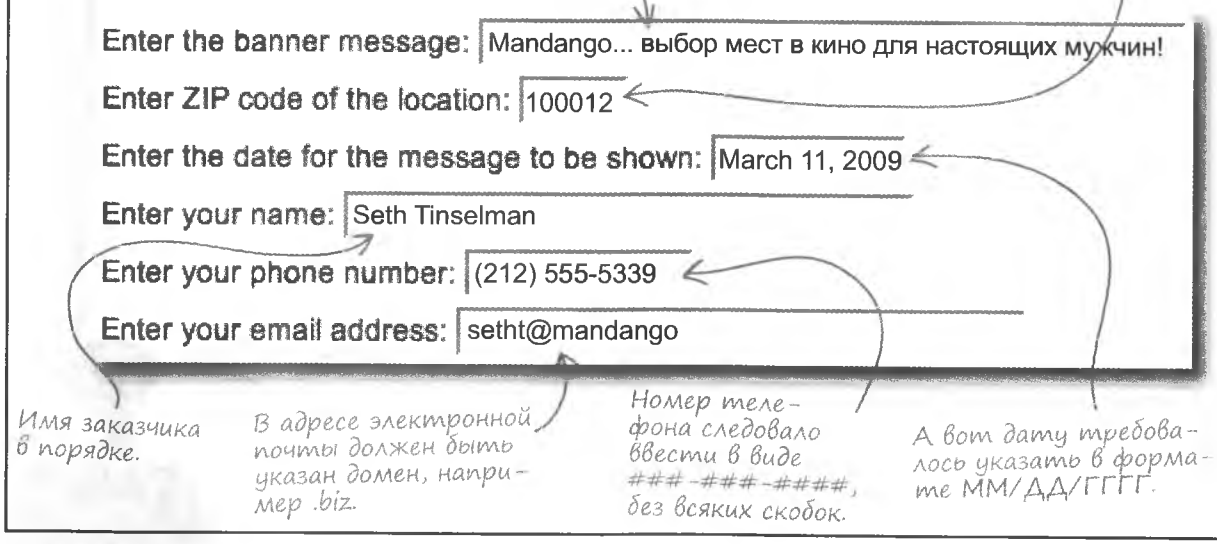
Enter your phone number:

Enter your email address:

Возьми в руку карандаш

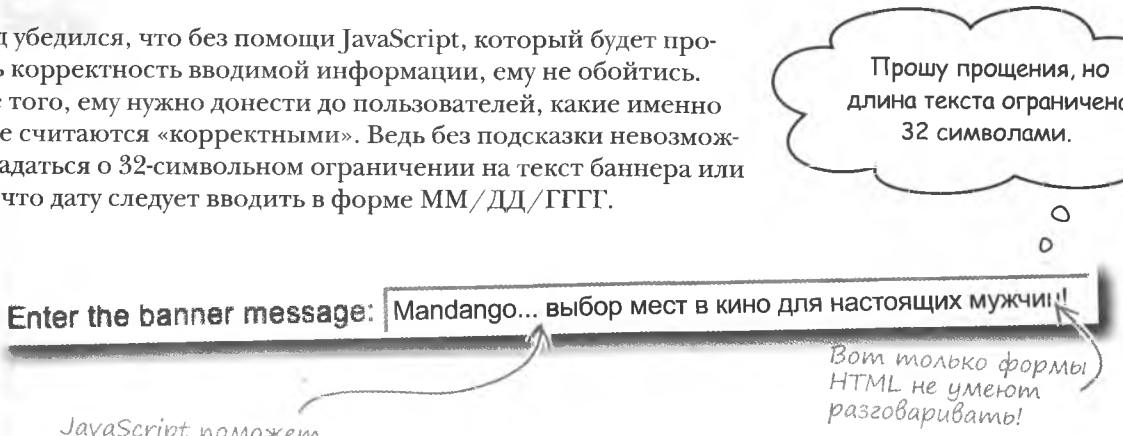
Решение

Вот каким образом мы заполнили предоставленную Говардом форму.



Когда языка HTML недостаточно

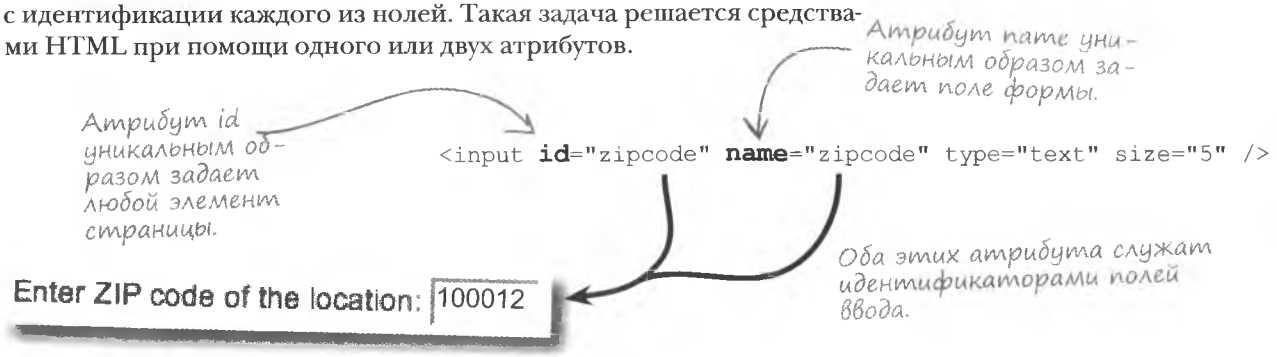
Говард убедился, что без помощи JavaScript, который будет проверять корректность вводимой информации, ему не обойтись. Кроме того, ему нужно донести до пользователей, какие именно данные считаются «корректными». Ведь без подсказки невозможно догадаться о 32-символьном ограничении на текст баннера или о том, что дату следует вводить в форме ММ/ДД/ГГГГ.



Основная проблема состоит в том, что все умение JavaScript управляться с данными не поможет Говарду, пока он не поймет, каким образом предоставить JavaScript доступ к вводимой в форму информации...

Доступ к данным формы

Получение доступа к введенным в форму данным нужно начать с идентификации каждого из полей. Такая задача решается средствами HTML при помощи одного или двух атрибутов.

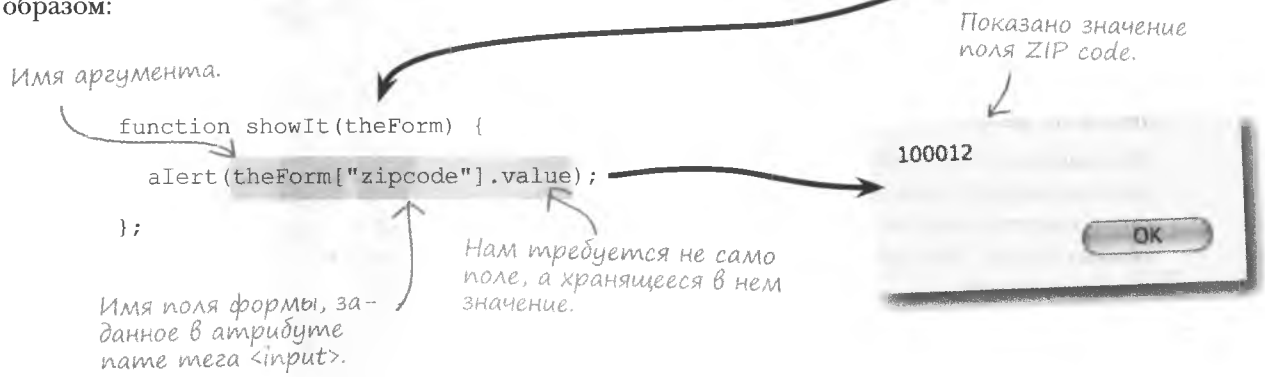


Наличие двух идентификаторов объясняется существованием нескольких способов получения доступа к полю. Во-первых, с помощью функции `getElementById()`. Этот способ прекрасно работает, но есть и более простая альтернатива.

Каждому полю формы соответствует объект `form`, который можно передать функции, проверяющей корректность введенных данных.

```
<input id="zipcode" name="zipcode" type="text" size="5" onclick="showIt(this.form)"/>
```

Объект `form` представляет собой массив, содержащий все поля формы. При этом индексы массива не являются числами; их роль играют уникальные значения атрибутов `name`! Поэтому при передаче функции объекта `form` в виде аргумента с именем `theForm` доступ к полю ZIP code осуществляется следующим образом:



Данный способ доступа к данным формы не лучше и не хуже, чем доступ с помощью функции `getElementById()`, но он проще для восприятия и требует меньшего количества кода.

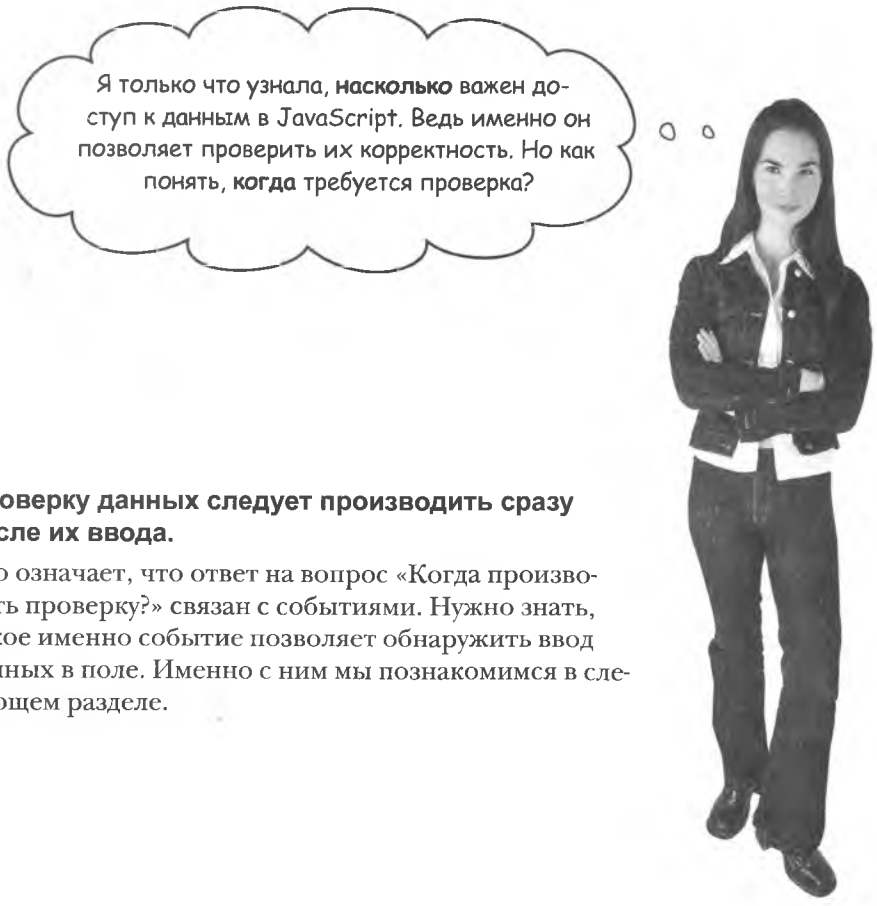
Часто Задаваемые Вопросы

В: Почему даже отдельное поле формы имеет доступ к объекту `form`?

О: Иногда это не так, но следует помнить, что поля формы могут вызывать функцию проверки, которой требуется доступ к данным. В этом случае удобный доступ к остальным полям формы обеспечивает именно объект `form`. Этот объект обычно передается функции проверки, поэтому она может быстро получить данные из любого нужного ей поля. Вы в этом еще убедитесь на примере с формой Bannerocity.

В: То есть значение является свойством поля формы? То есть все поля формы являются объектами?

О: Да. Каждое поле формы в коде JavaScript представлено объектом `form`, который предоставляет быстрый и легкий способ доступа к введенной в поля информации. Запись вида `form["objectname"]` дает доступ к значению при помощи свойства `value`. Но об этом мы поговорим в главах 9 и 10.



Проверку данных следует производить сразу после их ввода.

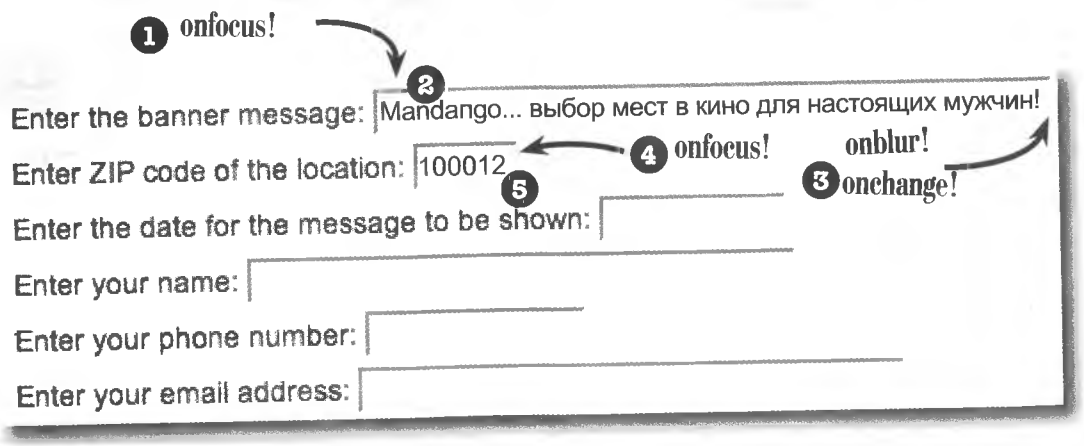
Это означает, что ответ на вопрос «Когда производить проверку?» связан с событиями. Нужно знать, какое именно событие позволяет обнаружить ввод данных в поле. Именно с ним мы познакомимся в следующем разделе.

Цепочка событий

Ввод данных в форму сопровождается целым рядом событий. Эти события можно использовать как точку входа для операции проверки данных. Рассмотрим типичную последовательность действий при вводе информации, чтобы понять, какие именно события при этом возникают и в какой именно момент.

1. Выделите поле (`onfocus`).
2. Введите данные.
3. Перейдите к следующему полю (`onblur/onchange`).
4. Выделите его (`onfocus`).
5. Введите в поле данные...

Ввод данных в поля формы сопровождается целой цепочкой событий JavaScript.



При выделении поля появляется событие `onfocus`, в то время как снятие выделения сопровождается событием `onblur`. Событие `onchange` в отличие от события `onblur` появляется в случае, когда с поля снимается выделение, но его содержимое уже было изменено.

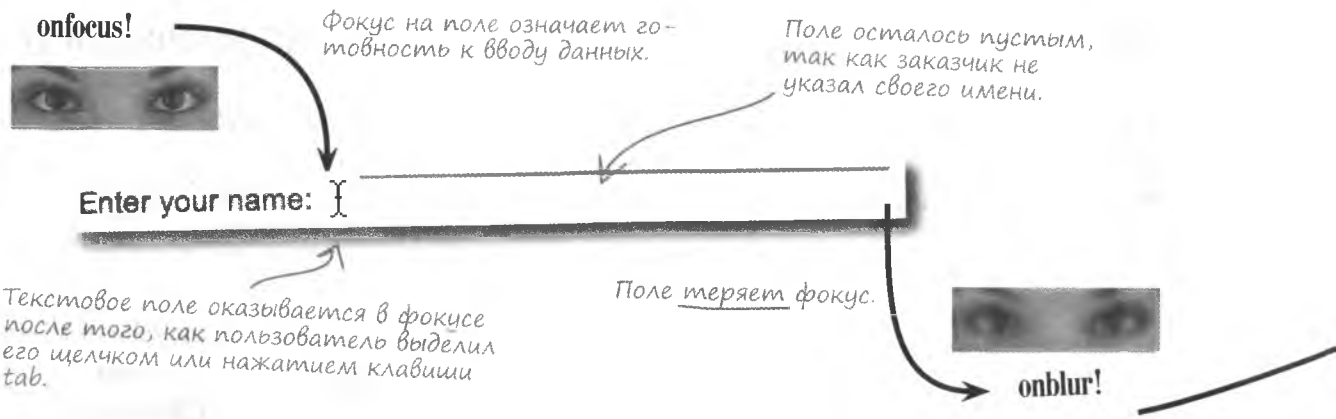
МОЗГОВОЙ ШТУРМ

Какие из вышеупомянутых событий нам потребуются для проверки данных?

Событие onblur

Было бы логично начинать проверку данных после события `onchange`, но непонятно, как при этом поступать с пустыми полями. Ведь для них ничего не изменилось, даже если пользователь и выделял такое поле в процессе заполнения формы. Проблема решается при помощи события `onblur`, возникающего при потере объектом фокуса.

Событие onblur является замечательным сигналом к началу проверки данных.



В отличие от события `onchange`, событие `onblur` возникает вне зависимости от того, были ли введены в поле данные. Таким образом, событие `onblur`, конечно, является полезным инструментом, но нужно внимательно отслеживать, когда и как вы применяете его для уведомления пользователей о результатах проверки данных.

Часто Задаваемые Вопросы

В: Я правильно понимаю, что ряд событий возникает после ввода данных пользователем?

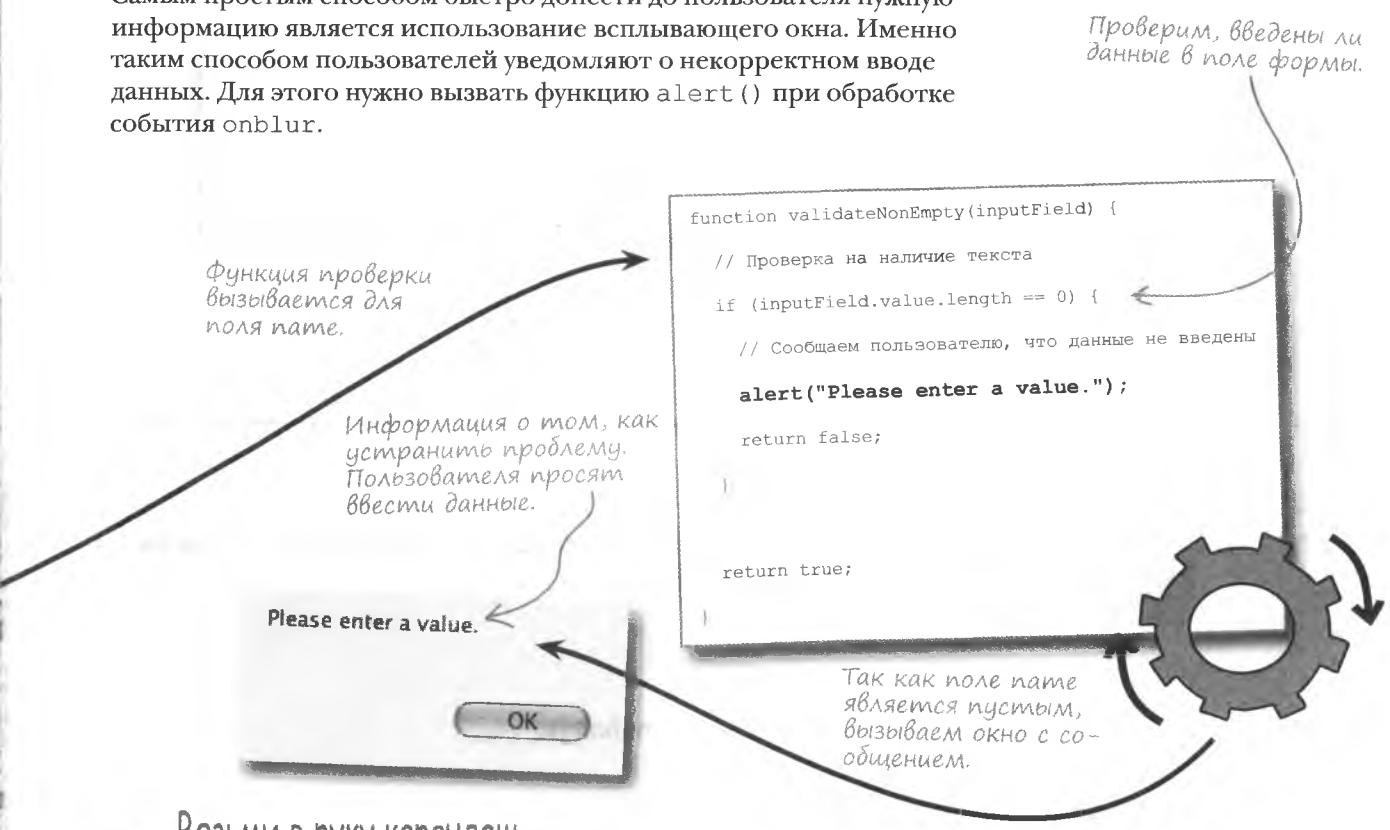
О: Да. В ответ на нажатие клавиш появляются события `onkeypress`, `onkeyup`, `onkeydown` и т. п. Иногда имеет смысл написать реагирующий на них код, но для проверки информации они не подходят, так как окно с предупреждением будет появляться еще до завершения ввода. Фактически это приведет к появлению уведомлений о каждой опечатке и каждом незавершенном фрагменте информации. Поэтому лучше дождаться момента, когда пользователь покинет поле. Именно в этот момент появляется событие onblur.

В: Что означает это странное название `onblur`?

О: Событие `onblur` является противоположным по смыслу событию `onfocus`. То есть если событие `onfocus` возникает в момент **фокусировки** на поле, то время события `onblur` наступает после **потери** полем фокуса. Даже слово «фокус» в данном контексте неточно отражает смысл происходящего, но именно от него пошло слово «blur» (размывание), указывающее на отсутствие фокуса.

Сообщение проверки

Самым простым способом быстро донести до пользователя нужную информацию является использование всплывающего окна. Именно таким способом пользователей уведомляют о некорректном вводе данных. Для этого нужно вызвать функцию `alert()` при обработке события `onblur`.



Возьми в руку карандаш

Сколько событий `onblur` появляется при такой последовательности ввода данных? А сколько событий `onchange`?

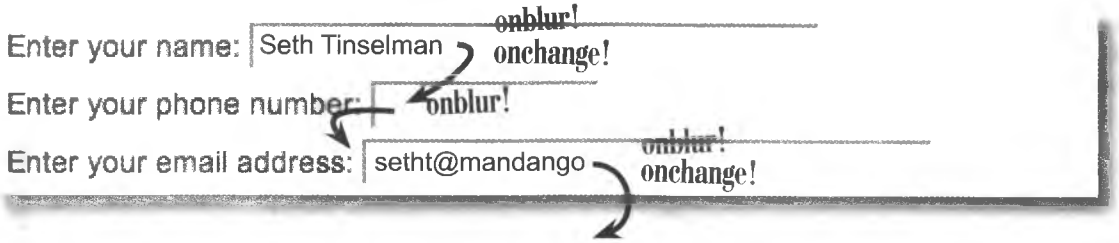
Событий `onblur`:.....

Событий `onchange`:.....

Возьми в руку карандаш

Решение

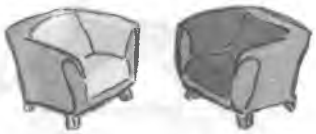
Вот сколько событий `onblur` и `onchange` появляется при таком способе ввода данных.



События `onblur`: 3

События `onchange`: 2

Беседа у камина



События onblur и onchange решают, когда следует реагировать на некорректные данные.

onblur:

Кажется, в наши дни сценарии только и заботятся об отслеживании действий пользователей. Вот тут-то я и пригождаюсь.

Именно об этом я и хотел с тобой поговорить. Ходят слухи о каких-то пустых полях. И все показывают пальцем на тебя.

Так никто и не сомневался в твоей способности реагировать на изменения. Но вот как ты поступаешь в случае, когда в поле так ничего и не ввели?

onchange:

Это да. Мы всегда даем знать, если элемент формы вдруг оказался измененным или потерял фокус... или все сразу!

Честно говоря, твой намек меня шокирует. Ты же знаешь, что я всегда оповещаю о любых внесенных в форму изменениях.

onblur:

Вот именно. Это не имеет смысла. Точно так же, как здравый смысл отсутствует у многих пользователей, но при этом они заполняют формы.

Не волнуйся, ты тут ни при чем. В конце концов, данные, оставшиеся без изменений, не твоя забота. Тебя зовут onchange.

Я уже сказал, что это не твоя забота. Просто, если для сценария важно, чтобы все поля были заполнены, ему не следует прибегать к твоим услугам.

Не стоит так нервно реагировать. Ты, конечно, не очень подходишь для запуска кода проверки, но это не означает, что сценарию не нужно узнавать об изменении данных. Ведь есть еще и формы, позволяющие редактировать хранящиеся в других местах данные. Ты можешь запускать процедуру сохранения, только если изменения были действительно внесены.

Конечно! Так что не переживай.

Пожалуйста. Рад был с тобой поболтать, но мне нужно проверить достоверность вон тех данных.

onchange:

Ты утверждаешь, что пользователь может попытаться отправить форму с незаполненными полями? Но это же не имеет никакого смысла.

Хорошо. Рассмотрим форму, в которой пользователь не заполнил некоторые поля, но тем не менее решил ее отправить... черт... мне страшно!

Но как же быть с предложенным тобой сценарием, при котором отправляется форма с незаполненными полями?

Рад это услышать, даже если твое утверждение и означает, что со мной вообще не надо работать. О боже, мне опять плохо...

Да, действительно. Получается, что от меня все-таки есть польза?

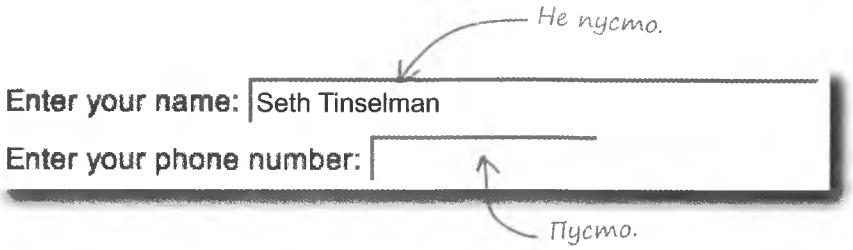
Спасибо. Ты меня обнадежил.

Ищем... что-нибудь

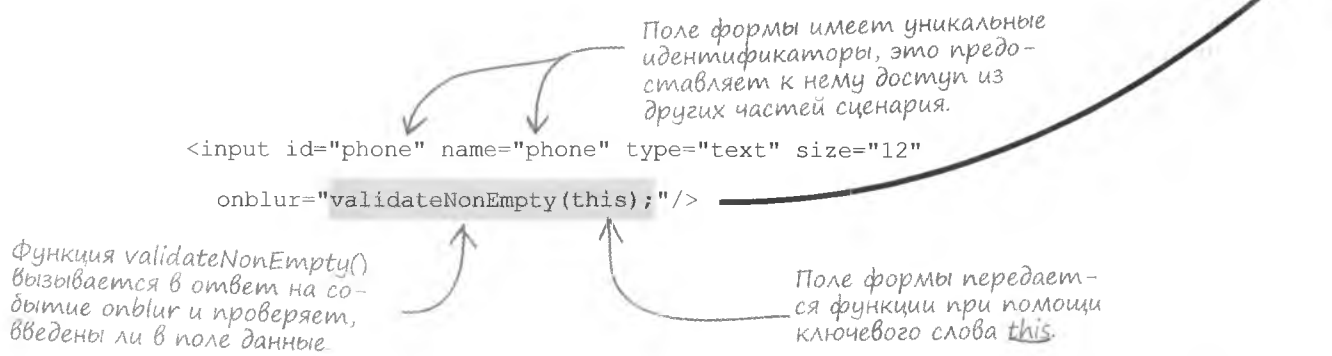
Вернемся к странице Bannerocity. Говард понимает, что ему для начала следует проверить, были ли введены данные в поля его формы. С точки зрения JavaScript эта задача выглядит немного по-другому. Лучше проверять не отсутствие данных в поле, а их наличие. Другими словами, «что-то» — это «не ничего».

Что-то = Не ничего

Причиной такого парадоксального на первый взгляд подхода является тот факт, что намного проще проверить отсутствие данных.



Функция проверки должна реагировать на событие `onblur` для каждого из полей, проверяя, не осталось ли поле пустым. Например:



Ключевое слово `this` является ссылкой на поле формы. Передавая поле в виде объекта функции проверки, оно предоставляет доступ к введенному в это поле значению. Ведь все поля формы содержат именно объект `form`.

Проверка полей на наличие данных

Для каждого из полей формы существует код, связывающий событие `onblur` с функцией `validateNonEmpty()`. Это обеспечивает проверку всех полей.

```
<input id="name" name="name" type="text" size="32"
  onblur="validateNonEmpty(this);"/>
```

Поле name функция проверяет на наличие имени.

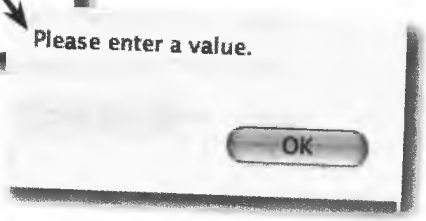
Функция проверяет поле phone number на наличие телефонного номера.

```
function validateNonEmpty(inputField) {
  // Проверка на наличие текста
  if (inputField.value.length == 0) {
    // Сообщаем пользователю, что данные не введены
    alert("Please enter a value.");
    return false;
  }

  return true;
}
```

Свойство length задает количество символов в строке.

Телефонный номер не введен, поэтому функция возвращает значение false и открывает окно с предупреждением.

Имя в поле обнаружено, поэтому функция возвращает значение true.



В данном примере пока никак не задействованы значения, возвращаемые функцией `validateNonEmpty()`. Они сообщают вызвавшему функцию коду о результатах проверки: `true`, если данные обнаружены, и `false` в противном случае. Немного позже вы увидите, каким образом эти значения используются для обеспечения корректности данных перед отправкой их на сервер для дальнейшей обработки.

Функция проверки гарантирует заполнение всех полей формы.

Какие недостатки имеет всплывающее окно с оповещением о вводе некорректных данных?

Проверка без предупреждающих всплывающих окон

Говард быстро сообразил, что всплывающие окна не самый лучший способ информирования о вводе некорректных данных. Слишком много жалоб поступало от потенциальных заказчиков. Ведь рано или поздно любого начинает раздражать, когда ввод данных прерывается появлением окна. Пусть даже в этом окне и содержится нужная пользователю информация.

Говард решил перейти к «пассивным подсказкам», которые не прерывают процесс ввода. Он предпочел добавить к форме несколько элементов HTML.

Новый HTML-элемент предоставил место для информационного сообщения.

Enter your phone number: Please enter a value.

Новые HTML-элементы позволяют донести до пользователя полезную информацию без участия всплывающих окон. Достаточно добавить тег `<span>`, задающий контейнер для внутреннего текста. Этот тег появляется в коде веб-страницы сразу под полем input.

Второй аргумент функции validateNonEmpty() теперь передается с вспомогательным текстом.

```
<input id="phone" name="phone" type="text" size="12"
  onblur="validateNonEmpty(this, document.getElementById('phone_help'))" />
<span id="phone_help" class="help"></span>
```

Тег изначально является пустым, но он имеет ID, связанное с полем формы phone number.

Совпадение этих двух идентификаторов обеспечивает появление вспомогательного текста в поле ввода.

Для форматирования вспомогательного текста используется стиль class. В результате он пишется наклонным шрифтом красного цвета, хотя в книге этого и не видно!

Итак, мы вставили элемент `span`, содержащий вспомогательный текст, и теперь нам требуется код, который будет его отображать. Сделаем ответственной за эту операцию функцию `validateNonEmpty()`.

Усложняем наш валидатор

Новый подход к вспомогательным сообщениям заставляет нас отредактировать функцию `validateNonEmpty()`. Теперь она должна еще и отображать и убирать текст из полей формы.

Объект helptext передается функции в качестве второго аргумента.

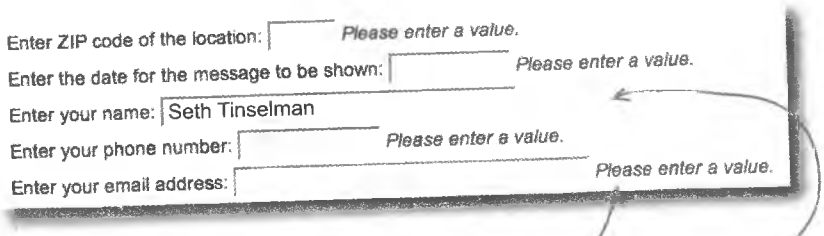
```
function validateNonEmpty(inputField, helpText) {
  // Проверка на наличие текста
  if (inputField.value.length == 0) {
    // Сообщаем пользователю, что данные не введены
    if (helpText != null)
      helpText.innerHTML = "Please enter a value.";
    return false;
  }
  else {
    // Данные обнаружены, убираем вспомогательный
текст
    if (helpText != null)
      helpText.innerHTML = "";
    return true;
  }
}
```

Убеждаемся, что элемент helptext существует (helpText != null), и присваиваем свойству innerHTML вспомогательный текст.

Нужно сделать так, чтобы после ввода данных вспомогательный текст исчезал из поля.

Проверка данных на странице Bannerocity стала заметно лучше благодаря новому подходу. Мы все еще используем для этой цели JavaScript, но уже более деликатным способом.

Теперь вам не будут надоедать всплывающие окна.



Если данные отсутствуют, в поле отображается просьба их ввести.

Так как все введено, вспомогательный текст не появляется.

Хорошенького понемножку

Наша проверка на наличие данных работает отменно, но задумывались ли вы о том, что избыток может быть так же вреден, как и недостаток? Чтобы понять суть проблемы, достаточно посмотреть на последний заказанный Говарду баннер.

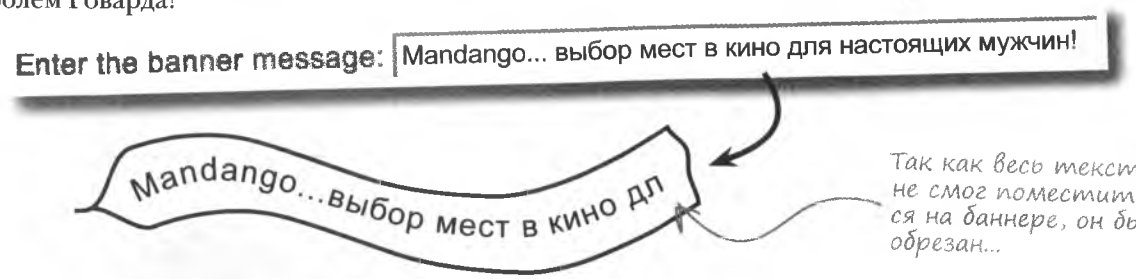
МОЗГОВОЙ ШТУРМ

Что же случилось с баннером и каким образом можно решить данную проблему?

Размер имеет значение...

Проблема в том, что на воздушном баннере помещается всего 32 символа, в то время как в соответствующее поле можно вводить фразы произвольной длины. Пока что появляется только вспомогательное сообщение о необходимости ввода данных, но нигде не написано об ограничениях на ввод. Вот и причина проблем Говарда!

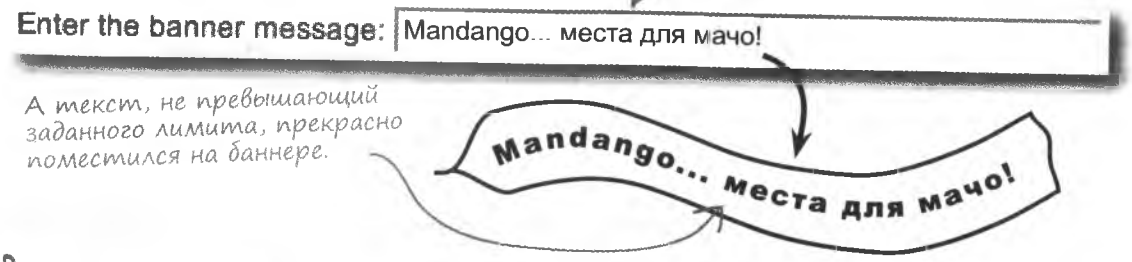
Был введен слишком длинный текст, но пользователя не проинформировали об этом.



Так как весь текст не смог поместиться на баннере, он был обрезан...

Невозможно поместить неограниченное количество символов на ограниченное место, а в результате мы имеем недовольных клиентов. Значит, требуется проверять еще и длину вводимых фраз. Разумеется, снабдив форму еще одним вспомогательным сообщением для пользователей.

Новый текст ограничен 32 символами.



А текст, не превышающий заданного лимита, прекрасно поместился на баннере.

Возьми в руку карандаш

Напишите псевдокод, демонстрирующий работу функции, которая проверяет длину сообщений. Не забудьте проверить не только максимальную, но и минимальную длину.

.....

.....

.....

.....

Возьми в руку карандаш

Решение

Вот принцип работы функции, проверяющей длину вводимых в поле сообщений.

Аргументам функции minLength и maxLength следует присвоить значения 1 и 32 соответственно.

If (fieldValue меньше, чем minLength OR fieldValue длиннее, чем maxLength)

Показать вспомогательный текст

Else

Убрать вспомогательный текст

Проверка длины

Новая функция `validateLength()` должна проверять, не выходит ли длина сообщения за отведенные границы. В случае с Bannerocity она должна проводить ограничения сверху, хотя проверка на минимальное количество символов в нее тоже встроена. На всякий случай. Причина в том, что Говард сомневается в существовании клиентов, которые захотят написать на баннере всего одну букву.

```
validateLength(minLength,
  maxLength, inputField,
  helpText);
```

Новой функции потребуется также аргумент для поля ввода и вспомогательный текст, призванный помочь пользователю правильно составить фразу. То есть всего у функции будет четыре аргумента.

maxLength
Максимальная длина вводимого в поле текста.

Enter the banner message: Mandango...выбор мест в кино для настоящих Please enter a value 1 to 32 characters in length.

minLength
Минимальная длина вводимого в поле текста.

inputField
Поле, количество символов в котором проверяется.

helpText
Элемент, отображающий вспомогательный текст.

```
<input id="message" name="message" type="text" size="32"
  onblur="validateLength(1, 32, this, document.getElementById('message_help'))" />
<span id="message_help" class="help"></span>
```

Объект, представляющий собой поле ввода для текста баннера.

Функция `validateLength()` берет значение аргумента `inputField` и проверяет, что это значение имеет (как минимум) длину, заданную параметром `minLength`, но не превышает параметра `maxLength`. В случае слишком короткого или слишком длинного значения в элементе `helpText` появится подсказка.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Все поля формы являются объектами JavaScript.
- Каждое поле формы имеет свойство `form`, представляющее **всю форму** в виде массива полей.
- Событие `onblur` возникает при потере объектом фокуса, и именно оно запускает функцию проверки данных.
- Всплывающие окна — не самый лучший способ информирования пользователей о проблемах с вводом данных.
- Пассивный подход к проверке данных меньше раздражает пользователей.
- Свойство `length` указывает на количество символов в строке.

Возьми в руку карандаш

Напишите код функции `validateLength()`, уделяя особое внимание передаваемым ей аргументам.

```
function validateLength(minLength, maxLength, inputField, helpText) {
  // Смотрим, содержит ли текст как минимум minLength, но не больше maxLength символов.

  // Сообщаем пользователю о вводе некорректных данных.

  // С данными все в порядке, убираем подсказку.

}
```

Возьми в руку карандаш

Решение

Вот как выглядит код функции `validateLength()`.

```
function validateLength(minLength, maxLength, inputField, helpText) {
  // Смотрим, содержит ли текст как минимум minLength, но не больше maxLength символов
  if (inputField.value.length < minLength || inputField.value.length > maxLength) {
    // Сообщаем пользователю о вводе некорректных данных
    if (helpText != null)
      helpText.innerHTML = "Please enter a value " + minLength + " to " + maxLength +
        " characters in length.";
    return false;
  }
  else {
    // С данными все в порядке, убираем подсказку
    if (helpText != null)
      helpText.innerHTML = "";
    return true;
  }
}
```

Проверяем как максимальную, так и минимальную длину вводимого в поле текста.

Уведомление о том, что вводимый текст ограничен сверху и снизу по количеству символов.

При вводе корректного текста убираем подсказку.

Проблема с текстом решена

Говард может вздохнуть с облегчением. Ведь у него пока нет денег на покупку большего баннера, поэтому он может решить проблему только на уровне JavaScript. По крайней мере теперь пользователи будут узнавать об ограничении на длину текста до того, как сделали заказ.

Вспомогательный текст появляется при вводе слишком большого количества символов.

Часто Задаваемые Вопросы

В: Чем так плохи всплывающие окна? Разве большинство пользователей не понимает, что это не реклама?

О: К сожалению, раздражающими являются любые действия, которые заставляют человека прервать свое занятие, чтобы закрыть появившееся окно. Именно поэтому мы крайне не рекомендуем использовать всплывающие окна при проверке данных.

В: Поясните назначение ключевого слова `this` в коде для события `onblur`. Объектом является поле или сама форма?

О: Оба ответа верны. В HTML ключевое слово `this` ссылается на элемент как на объект. То есть в случае поля `this` это ссылка на объект «поле». Этот объект обладает свойством `form`, дающим доступ к форме. То есть запись `this.form` в коде, связанном с событием `onblur`, означает ссылку на форму, как на объект.

Написав `this.form` в коде Bannerocity, мы получим доступ к элементу «вспомогательный текст», связанному с определенным полем ввода. Запомните, что `this.form` — это ссылка на объект `form`, который представляет собой массив со всеми полями формы. Поэтому для быстрого доступа к полю с именем `my_field` достаточно написать `this.form["my_field"]`. Аналогичный результат можно получить и с помощью функции `getElementById()`, но продемонстрированный выше подход является более лаконичным.

В: Какую роль играют атрибуты `name` и `id` при связывании вспомогательного текста с полем ввода?

О: Атрибут `id` элемента `helptext` основан на атрибутах `id`/`name` связанного с ним поля ввода, но это вовсе не одно и то же. Идентификатор вспомогательного текста получается из ID поля ввода добавлением `_help`. Подобная система именования позволяет создать прозрачную связь между полем и элементом, отображающим в этом поле текст подсказки. Хотя выбирать имя для идентификатора элемента `helptext` можно и самостоятельно, главное, чтобы оно было уникальным и корректно передавалось в функцию проверки.

В: Зачем удалять вспомогательный текст, если проверка показала корректность введенных данных?

О: Назначением вспомогательного текста является помощь пользователю при возникновении проблем. Если же данные введены корректно, проблемы нет и нет причины для отображения подсказки. А так как подсказка может оказаться видимой из-за более ранних проверок данного поля, в случае когда пользователь все ввел правильно, имеет смысл ее удалить.

В: Что произойдет, если элемента `helptext` не окажется в числе аргументов функции проверки?

О: Сценарий будет снова и снова искать отсутствующий элемент, раскалит страницу до красна и оставит от браузера обуглившиеся остатки. Шучу, на самом деле просто исчезнет система пассивной помощи пользователям на странице Bannerocity. И вспомогательный текст перестанет появляться. Это говорит о необязательности данной функции. То есть вы можете по желанию убрать подсказки или вывести их только для отдельных полей формы.

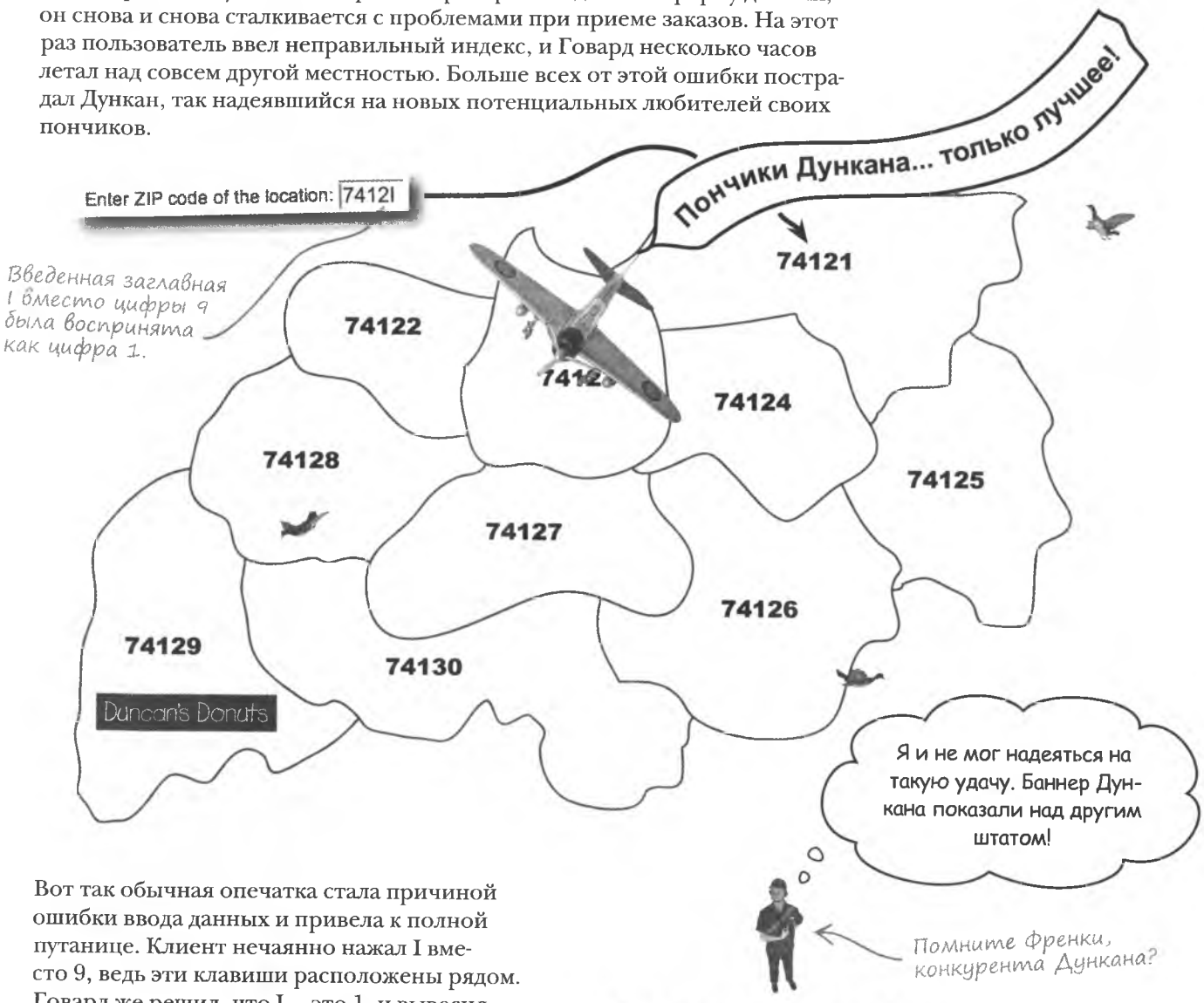
Если код проверки увидит, что аргумент `helpText` имеет значение, отличное от `null`, значит, нужный нам элемент существует и может быть отображен.

В: Ограничивает ли атрибут `size` поля формы длину вводимого сообщения?

О: Нет, HTML-атрибут `size` ограничивает только физический размер поля формы на странице — на количество вводимых символов он никак не влияет. К примеру, для поля ZIP code на странице Bannerocity атрибут `size` имеет значение 5, что означает возможность показа только пяти символов. Но, чтобы ограничить длину вводимого текста, потребуется уже атрибут `maxlength`. Функция проверки позволяет достаточно гибко управлять количеством вводимых символов. Так, в случае с индексами имеет смысл не только ограничить максимальное количество пятью, но и проверять, являются ли вводимые символы цифрами. Подозреваю, что Говард в один прекрасный момент захочет это реализовать.

Некорректное местоположение

Несмотря на все усилия Говарда по проверке вводимых в форму данных, он снова и снова сталкивается с проблемами при приеме заказов. На этот раз пользователь ввел неправильный индекс, и Говард несколько часов летал над совсем другой местностью. Больше всех от этой ошибки пострадал Дункан, так надеявшийся на новых потенциальных любителей своих пончиков.



Вот так обычная опечатка стала причиной ошибки ввода данных и привела к полной путанице. Клиент нечаянно нажал I вместо 9, ведь эти клавиши расположены рядом. Говард же решил, что I — это 1, и вывесил баннер совсем в другом месте.

МОЗГОВОЙ ШТУРМ

Как проверить правильность ввода индекса?

Проверка индексов

Итак, очередная проблема Говарда связана с неверно введенным индексом. В США индексы состоят из пяти цифр. Поэтому для начала следует проверить, чтобы клиент ввел именно пять цифр — ни больше, ни меньше.

Возьми в руку карандаш

Закончите код функции `validateZIPCode()`, проверяющей правильность ввода индексов.

```
function validateZIPCode(inputField, helpText) {
  // Проверяем количество вводимых символов. Их должно быть ровно 5

  if (                                    ) {
    // Если их количество не совпадает, показываем подсказку
    if (helpText != null)
      helpText.innerHTML = "Введите ровно пять цифр.";


    .....
  }
  // Проверяем, что все символы являются числами

  else if (                                    ) {
    // Если данные введены некорректно, показываем подсказку
    if (helpText != null)
      helpText.innerHTML = "Пожалуйста, введите число";


    .....
  }
  else {
    // Данные введены корректно, убираем подсказку
    if (helpText != null)
      helpText.innerHTML = "";


    .....
  }
}
```

Возьми в руку карандаш

Решение

Вот как выглядит функция `validateZIPCode()`, проверяющая корректность ввода индексов.

```
function validateZIPCode(inputField, helpText) {
  // Проверяем количество вводимых символов. Их должно быть 5.
  if ( inputField.value.length != 5 ) {
    // Если их количество не совпадает, показываем подсказку
    if (helpText != null)
      helpText.innerHTML = "Введите ровно пять цифр.";
    return false;
  }
  // Проверяем, что все символы являются числами
  else if ( isNaN(inputField.value) ) {
    // Если данные введены некорректно, показываем подсказку
    if (helpText != null)
      helpText.innerHTML = "Пожалуйста, введите число";
    return false;
  }
  else {
    // Данные введены корректно, убираем подсказку.
    if (helpText != null)
      helpText.innerHTML = "";
    return true;
  }
}
```

Длина введенной строки должна составлять 5 символов.

Так как количество введенных в поле символов отличается от 5, возвращается значение false.

Функция isNaN() проверяет, равен ли аргумент значению NaN (не-число).

Так как в поля введены символы, отличные от чисел, возвращается значение false.

Значение true возвращается, когда данные введены корректно.

Будьте осторожны!

Показанная выше функция не универсальна.

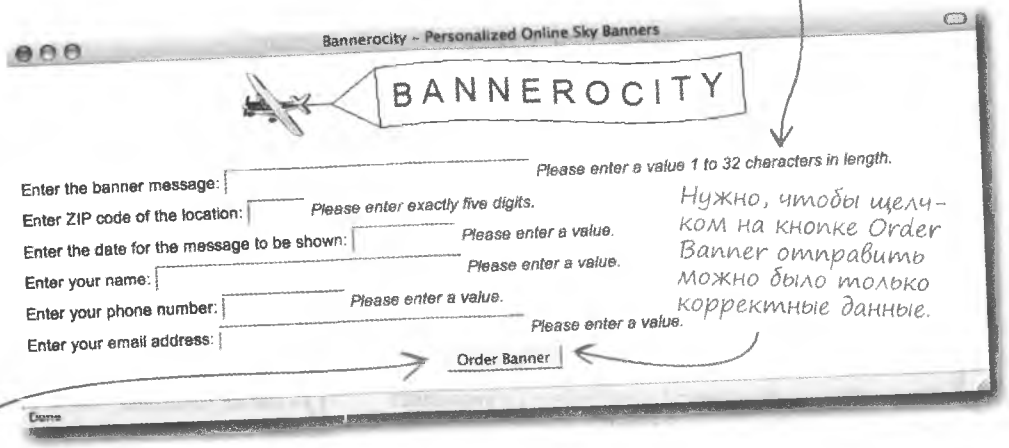
Например, она не подходит для проверки индексов других стран. Существуют страны, в которых индексы составлены не только из цифр, но и из букв. Более того, американские индексы в полной форме состоят из 9 цифр и имеют вид #####-####. Наличие дефиса делает индекс нечисловым.

А что произойдет, если клиент не будет обращать внимания на вспомогательные сообщения? Будет ли заполненная им форма с данными все равно отправлена на сервер?

Некорректные данные отправляться не должны.

В конце концов, зачем нужен код проверки, если пользователь все равно может отправить неверно введенные данные? На странице Bannerocity пока что отсутствует процедура проверки информации перед отправкой содержимого формы, а значит, не исключена вероятность дальнейших ошибок.

Проверка корректности не имеет смысла, если пользователь имеет возможность проигнорировать все предупреждения и все равно отправит данные.



Нужно, чтобы щелчком на кнопке Order Banner отправить можно было только корректные данные.

Самые устойчивые приложения на всякий случай проверяют данные еще и на сервере.

Таким образом, странице Bannerocity требуется еще одна функция, которая будет проверять все поля формы перед отправкой данных на обработку. Эту функцию `placeOrder()` мы свяжем с кнопкой Order Banner, и именно она будет осуществлять финальную проверку.

```
<input type="button" value="Order Banner" onclick="placeOrder(this.form);" />
```

Подробно о функции placeOrder()

```
function placeOrder(form) {
  if (validateLength(1, 32, form["message"], form["message_help"]) &&
    validateZIPCode(form["zipcode"], form["zipcode_help"]) &&
    validateNonEmpty(form["date"], form["date_help"]) &&
    validateNonEmpty(form["name"], form["name_help"]) &&
    validateNonEmpty(form["phone"], form["phone_help"]) &&
    validateNonEmpty(form["email"], form["email_help"])) {
    // Отправка заказа на сервер
    form.submit();
  } else {
    alert("Простите, но форма заполнена неверно");
  }
}
```

В качестве аргумента функции передается объект form, что дает ей доступ к полям формы.

Доступ к полям формы и элементам подсказок осуществляется через объект form по индексам массива.

Большую часть тела функции занимает большой оператор if/else, вызывающий функцию проверки для каждого из полей формы.

Метод submit() вызывается для отправки данных на сервер только при положительном результате проверки.

Ввод некорректных данных является достаточно значимой проблемой, чтобы оправдать появление всплывающего окна.

Часто Задаваемые Вопросы

В: Каким образом функция `placeOrder()` управляет отправкой данных формы на сервер?

О: Оператор `if/else` в теле функции структурирован таким образом, что при обнаружении некорректных данных в любом из полей будет запущен оператор `else`, вызывающий функцию `alert()`. Если же проверка всех полей покажет корректность введенных данных, будет вызван метод `submit()` объекта `form`, который и отправит данные на сервер. То есть отправка информации зависит от того, будет ли вызван метод `submit()`. Он является эквивалентом кнопки submit в языке HTML.

В: Вы же говорили, что не стоит пользоваться всплывающими окнами при проверке данных...

О: В большинстве случаев это действительно так, потому что пользователю в результате приходится прервать процесс заполнения формы, чтобы прочитать сообщение и щелкнуть на кнопке OK. Но щелчок на кнопке Order Banner осуществляется только после завершения ввода данных. И попытка отправить некорректно заполненную форму является достаточно веской причиной для появления всплывающего окна с предупреждением.

Проверка времени

К сожалению, проверка корректности ввода индексов принесла Говарду только временное облегчение, потому что практически сразу он столкнулся с новой проблемой. Теперь он показал баннер в нужном месте, но в другой день, так как что-то произошло с введенной датой...

Кажется, никто не в состоянии ввести цифру 9 правильно. На этот раз вместо нее была введена буква o...

Enter the date for the message to be shown: 05/1o/2008

Говард решил, что буква o — это ноль, и показал баннер 10 числа.

Stick Figure Adventures начинаются!

Воскресенье	Понедельник	Вторник	Среда	Четверг	Пятница	Суббота
				1	2	3
4	5	6	7	8		10
11			14	15	16	
18	19	20	21	22	23	24
25	26	27				31

Уже понедельник! И где, спрашивается, мой баннер?

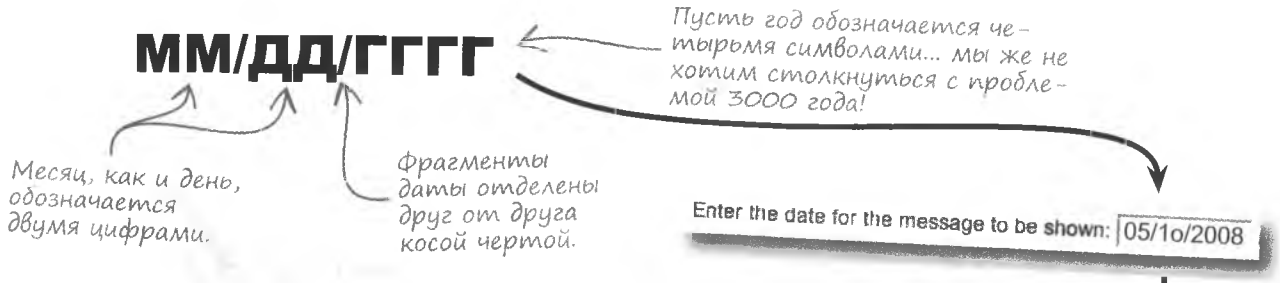
Еще один недовольный клиент.

МОЗГОВОЙ ШТУРМ

Каким образом можно удостовериться, что дата введена в форме ММ/ДД/ГГГГ?

Проверка даты

Говард понял, что недостаточно проверить, ввел ли пользователь дату, следует также удостовериться, что дата была введена корректно. Для этого нужно выбрать формат даты. Например, пусть сначала двумя цифрами обозначается месяц, потом двумя цифрами — день и наконец четырьмя цифрами — год.



Выбор формата даты — это самая простая часть задачи. Намного сложней реализовать процедуру проверки на соответствие этому формату. Существуют функции, позволяющие разделить строку на части, использовав в качестве разделителя какой-либо заранее выбранный символ, например косую черту. Но такая задача слишком сложна. Ведь придется сначала поделить строку на фрагменты, а затем удостовериться, что каждый фрагмент содержит только цифры и имеет заданную длину.

Вот как эта процедура будет выглядеть пошагово:

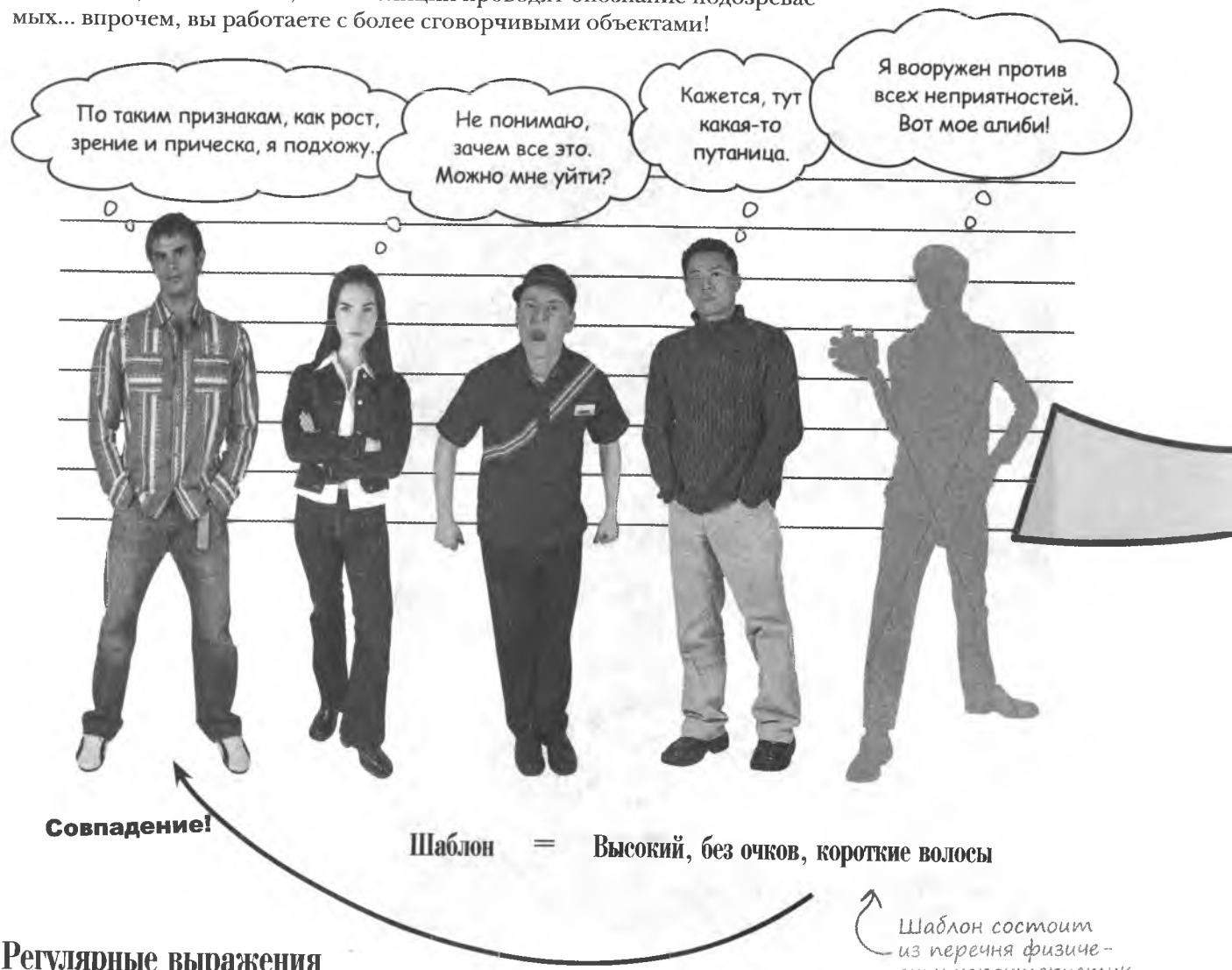
1. Разобьем значение поля на набор строк, использовав в качестве разделителя косую черту.
2. Убедимся, что первая строка состоит из двух символов, и это — цифры.
3. Убедимся, что вторая строка состоит из двух символов, и это — цифры.
4. Убедимся, что третья строка состоит из четырех символов, и это — цифры.
5. Проигнорируем все данные после второй черты.

С точки зрения написания кода такая последовательность выглядит достаточно просто, но проверить повторяющийся несложный шаблон можно и еще более простым способом.


А вот если бы даты можно было проверять без разделения на строки... Мечтать ведь не запрещено?

Регулярные выражения не «регулярны»

В JavaScript имеется такой мощный инструмент, как **регулярные выражения**, созданный специально для поиска в текстах совпадений с выбранным шаблоном. Вы создаете шаблон и применяете его к строке, ища совпадение, точно так же, как в полиции проводят опознание подозреваемых... впрочем, вы работаете с более сговорчивыми объектами!

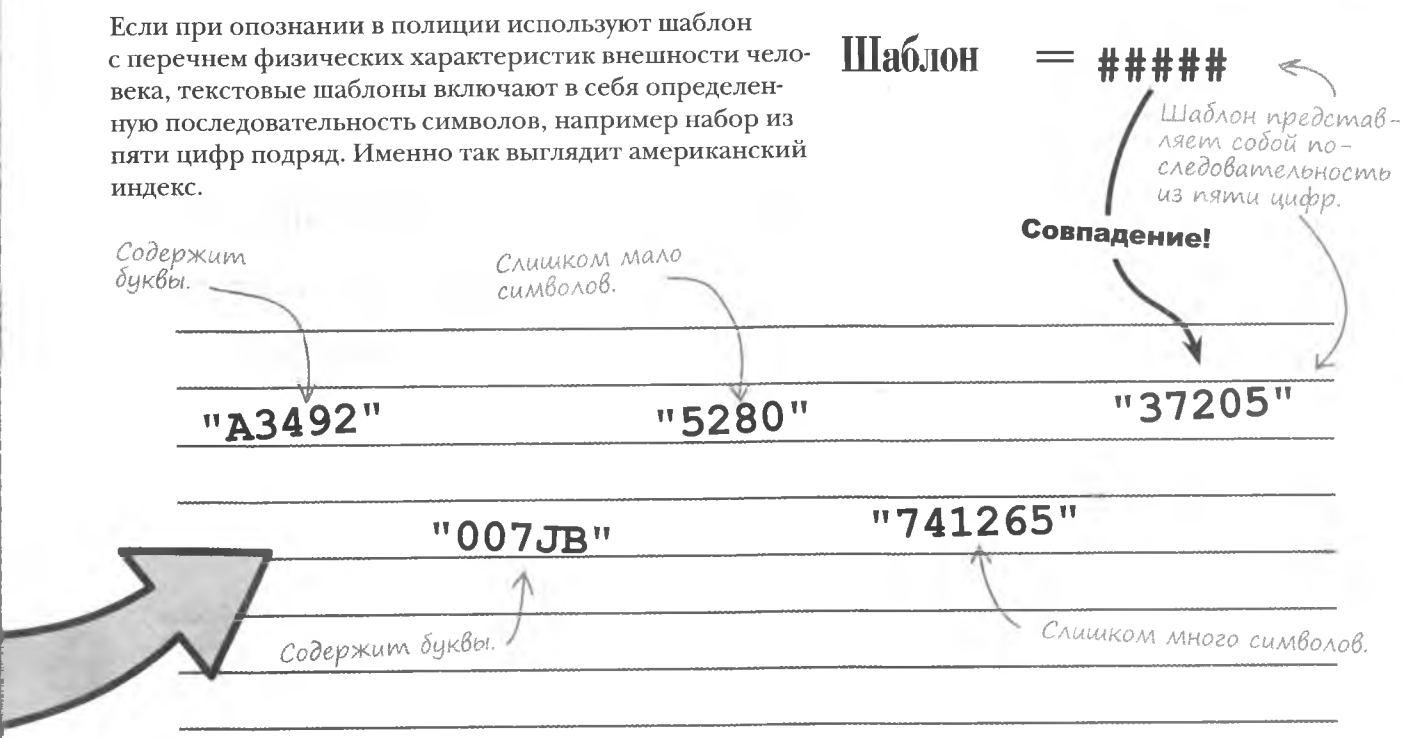


Регулярные выражения используются для поиска совпадений в тексте.

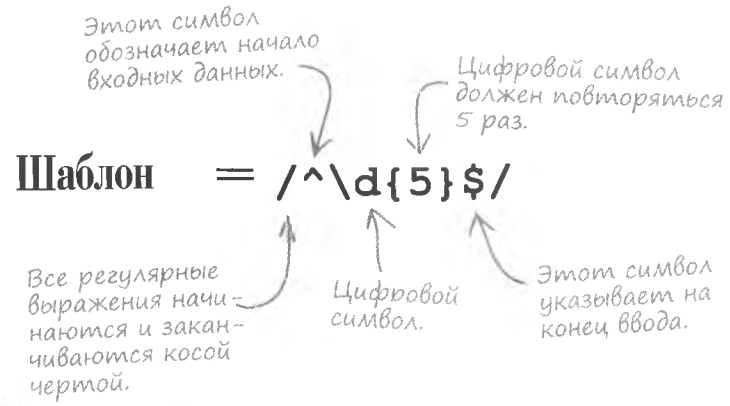
Шаблон описывает физические свойства, которые затем рассматриваются в приложении к реальным людям. Регулярные выражения позволяют делать то же самое со строками.

Задание шаблона

Если при опознании в полиции используют шаблон с перечнем физических характеристик внешности человека, текстовые шаблоны включают в себя определенную последовательность символов, например набор из пяти цифр подряд. Именно так выглядит американский индекс.



К сожалению, превратить состоящий из пяти цифр индекс в регулярное выражение не так-то просто. Ведь для описания шаблонов используется компактный и на первый взгляд совершенно непонятный синтаксис. Сразу и не догадаешься, что такое выражение означает всего лишь состоящий из пяти цифр индекс:

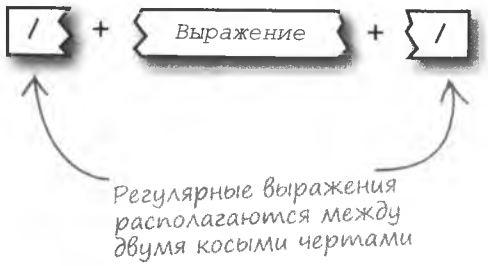


Не волнуйтесь, если вам кажется, что вы ничего не понимаете.

Мы еще не раз будем возвращаться к этому шаблону.

Структура регулярного выражения

Регулярное выражение напоминает строковый литерал, но в отличие от последнего не заключается в кавычки или в апострофы, а начинается и заканчивается косой чертой (/ /).



Регулярное выражение начинается с косой черты и заканчивается ею.

Тело регулярного выражения состоит из набора так называемых **метасимволов**, которые вместе с буквами и цифрами формируют лаконичный, но исчерпывающий шаблон. Для создания рабочих шаблонов вовсе не обязательно знать все нюансы «языка». Достаточно запомнить самые распространенные метасимволы:

Да, это всего лишь точка.

`.`

Обозначает любой символ, кроме перевода строки.

`\d`

Обозначает любую цифру.

Многие метасимволы начинаются с обратной косой черты.

`\w`

Обозначает любой символ латинского алфавита.

К таким символам относятся пробелы, табуляция, знаки перевода строки и т. п.

`\s`

Обозначает символы пробела.

`^`

Символ обозначает начало ввода данных.

Перед этим символом может располагаться только косая черта.

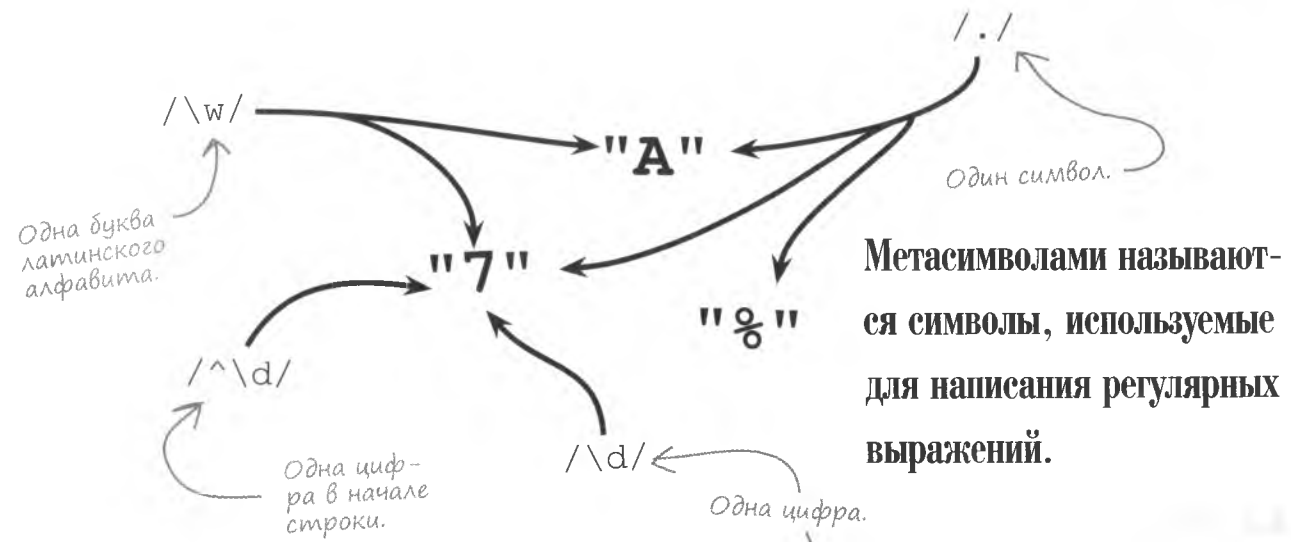
`$`

Символ конца ввода данных.

После этого символа может находиться только косая черта.

Мы постарались дать как можно более исчерпывающие описания метасимволов, но их назначение станет еще более понятным в процессе практического составления шаблонов...

Метасимволы



Метасимволами называются символы, используемые для написания регулярных выражений.

В регулярных выражениях один символ можно обозначить разными способами. А как быть со строками из нескольких символов? Давайте посмотрим на примерах таких строк:

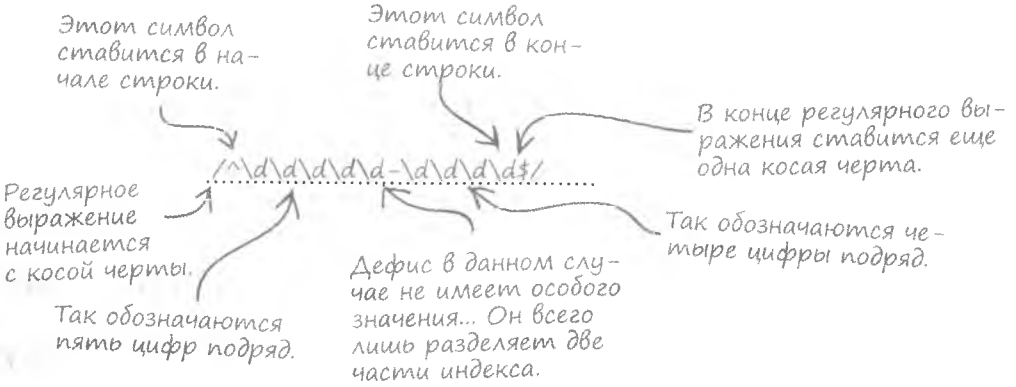
Упражнение

Напишите регулярное выражение для полного американского индекса, который имеет форму `#####-####` и должен появляться отдельно.

.....

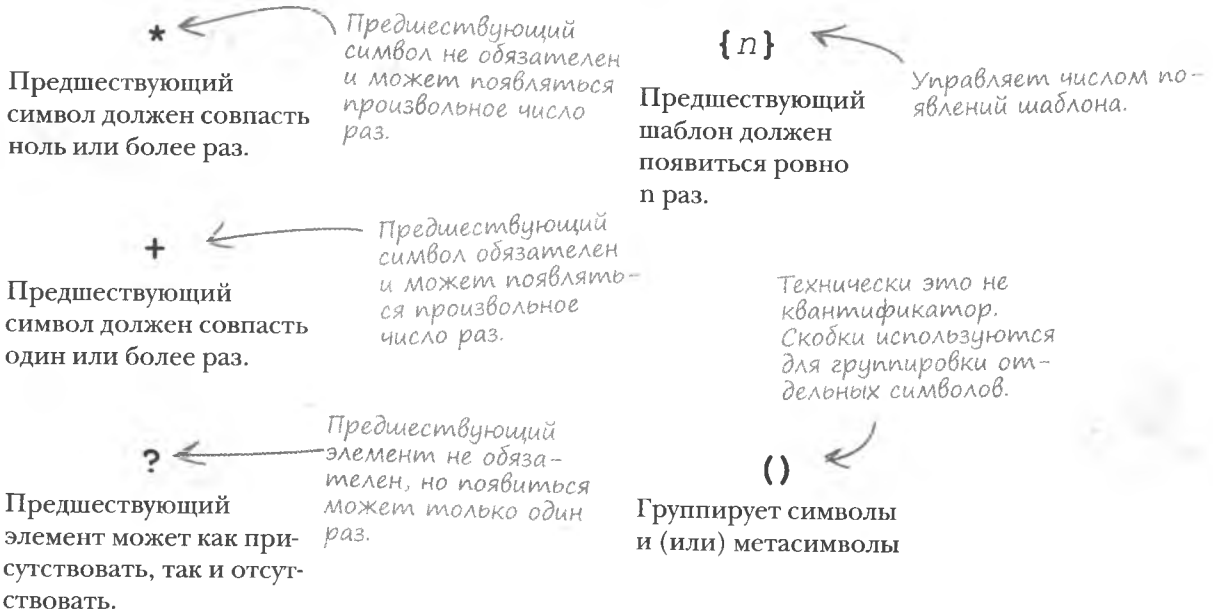
Упражнение
Решение

Вот как выглядит регулярное выражение для полной формы американского индекса #####-####.



Количество повторений

В регулярных выражениях любой текст, не относящийся к метасимволам, проверяется на совпадение. Это означает, например, что последовательность символов `/howard/` совпадает с текстом «howard» в произвольной части строки. Кроме того, существуют метасимволы, задающие количество повторений. Их называют **квантификаторами**.



Повторение шаблона

Квантификаторы позволяют сделать регулярные выражения более краткими. Теперь вам не требуется в явном виде писать все символы, достаточно указать, сколько раз их следует повторить. Вот как будет выглядеть шаблон для полной формы индекса после записи с использованием квантификаторов:

Квантификаторы задают количество повторений шаблона.

```
/^\d{5}-\d{4}$/
```

Квантификатор {} избавляет от необходимости писать все цифры.

Метасимволы и квантификаторы позволяют создавать регулярные выражения для любых сочетаний, которые могут появляться в строках.

```
/\w*/
```

Совпадает с произвольным количеством алфавитно-цифровых символов, включая пустую строку.

```
/.+/
```

Любой символ должен появиться один или более раз... совпадает с непустой строкой.

```
/(Hot)? ?Donuts/
```

Совпадает с выражением «Donuts» или «Hot Donuts».

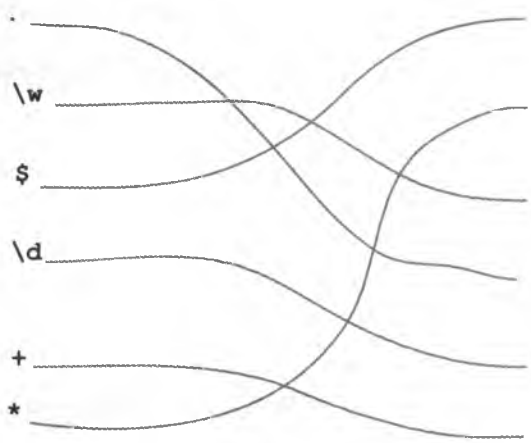
Кто и что делает?

Укажите назначение перечисленных слева метасимволов и квантификаторов.

.	Символ указывает на конец строки.
\w	Предшествующий символ **обязателен** и может появиться произвольное число раз.
$	Означает любой алфавитно-цифровой символ.
\d	Означает любой символ, кроме переноса строки.
+	Означает любую цифру.
*	Предшествующий символ **не обязателен** и может появиться произвольное число раз.

Кто и что делает?

Вот какое описание соответствует каждому из указанных в левом столбце метасимволов и квантификаторов.



Часто Задаваемые Вопросы

В: Регулярное выражение — это строка?

О: Нет. Регулярное выражение можно считать **описанием** строки или ее части. Регулярные выражения тесно связаны со строками и ищут в них указанные пользователем текстовые шаблоны, но строками они при этом не являются.

В: Можно ли применять регулярные выражения к другим типам данных?

О: Нет, регулярные выражения были созданы специально для поиска заданных шаблонов в текстовых строках. Но это не умаляет значения данного инструмента, позволяющего искать в текстовых строках самые разные варианты шаблонов.

В: Как найти в тексте знак доллара?

О: Как и в случае со строками, в JavaScript специальные символы выделяются обратными косыми чертами. Соответственно, для указания на знак доллара в регулярном выражении нужно написать символ `$` как `\$`. Аналогичное правило действует для прочих специальных символов, например таких, как `^`, `*` и `+`. Все прочие символы помещаются в регулярное выражение в своем обычном виде, без всякого дополнительного форматирования.

В: Имеют ли регулярные выражения отношение к проверке данных? Ведь мы заговорили о них в теме про проверку ввода дат.

О: Немножко терпения. Скоро вы попрактикуетесь в применении регулярных выражений. Они действительно очень удобны для проверки данных сложных форматов, например дат или адреса электронной почты. В форме Bannerocity еще много нуждающихся в проверке полей, для части которых вам потребуются регулярные выражения.

В: А каким образом в JavaScript используются регулярные выражения?

О: Мы дойдем и до этого, честное слово! Регулярные выражения в JavaScript представляются в виде объектов, снабженных набором необходимых для функционирования методов.

МАСТЕР ШАБЛОНОВ

Интервью недели: Загадочные, но мощные регулярные выражения

Head First: Я много слышал о вашей способности находить в тексте указанные шаблоны. Это действительно так?

Регулярное выражение: Да, я своего рода расчленитель кода, способный, взглянув на строку, сразу обнаружить нужный шаблон. Меня можно использовать даже в ЦРУ... Но они не отвечают на мои звонки.

Head First: Хотите ловить шпионов?

Регулярное выражение: Нет, я просто люблю отыскивать шаблоны. Дайте мне какие угодно параметры, и я тут же их найду, ну или, по крайней мере, сообщу, что совпадений не обнаружено.

Head First: А разве метод `indexOf()` объекта `String` не делает то же самое?

Регулярное выражение: Этот любитель даже не представляет, что такое работа с шаблонами. Нет, если вам нужен простейший механизм поиска, который, к примеру, ищет в строке слово «lame», метод `indexOf()` вам поможет. Но в более серьезных случаях он бессилен, так как не умеет анализировать строки.

Head First: Но разве поиск по строкам не является поиском шаблонов?

Регулярное выражение: Разумеется. Дойти до почтового ящика — это тоже физическое упражнение, но что-то его не включают в программу Олимпийских игр, по крайней мере пока. Так и обычный поиск. Вы ищете совпадения простейших шаблонов — статических слов или фраз. А теперь представьте, что требуется найти дату или адрес URL. Такие запросы хотя и имеют строковый формат, но не являются статическими.

Head First: Кажется, я понял, что вы имеете в виду. Шаблон — это описание текста, которое может появиться в строке, но не сам текст?

Регулярное выражение: Именно так. Представьте, что я прошу вас дать мне знать, когда пройдет человек, высокий, с короткими волосами и без очков. Это же описание человека, а не он сам. Как только появится парень Ален, подходящий под это описание, мы скажем, что шаблон совпал. Но под описание могут попасть и другие люди. Без шаблонов мы не смогли бы проверять, подходит ли человек под описание. Так что разница между методом `indexOf()` и мной в том, что он ищет Алена, а я высокого человека, с короткой стрижкой и без очков.

Head First: А каким образом ваши способности могут применяться для проверки данных?

Регулярное выражение: Такая проверка выполняется именно на соответствие определенному, заранее заданному формату. Так что я беру шаблон и смотрю, подходит ли он к данным. Положительный результат означает, что данные введены корректно.

Head First: Регулярных выражений должно быть очень много?

Регулярное выражение: Конечно. И именно эта часть моей работы требует наибольшей сосредоточенности — написание регулярного выражения, описывающего очередной формат.

Head First: Большое спасибо, что рассказали про вашу роль в проверке данных.

Регулярное выражение: Нет проблем. Я привык объяснять... Думаю, это мой поведенческий шаблон.

Проверка данных при помощи регулярных выражений

Метод test( ) объекта RegExp проверяет наличие в строках заданного шаблона.

Давайте перестанем создавать регулярные выражения просто для того, чтобы посмотреть, как они работают, вернемся к полю для ввода дат на странице Bannerocity и поможем Говарду с заказами. В JavaScript регулярные выражения представлены объектом `RegExp`, который снабжен методом `test()`. Именно этот метод и проверяет соответствие нужному нам шаблону.

Это регулярное выражение для индекса из пяти цифр.

Данный литерал автоматически создает объект RegEx.

В качестве аргумента методу передается значение, введенное в поле.

```
var regex = /^\d{5}$/;

if (!regex.test(inputField.value))

  // Индекс введен неверно!
```

Если метод test() возвращает значение false, значит, данные не прошли проверку.

Метод test() вызывается объектом RegEx.

Метод test() возвращает значение true при совпадении с шаблоном.

Метод `test()` можно вызвать для каждой из функций проверки, но мы предпочтем создать обобщенную функцию на основе регулярных выражений. Присвоим ей имя `validateRegEx()`. Вот как выглядит принцип ее работы:

1. Проверяет, совпадают ли переданные ей в качестве аргументов шаблон и строка.
2. При совпадении присваивает подсказке переданный ей в качестве аргумента текст и возвращает `false`.
3. В случае несовпадения удаляет подсказку и возвращает значение `true`.

```
validateRegEx(regex,
  inputStr, helpText,
  helpMessage);
```

Осталось только написать код этой функции. Большая его часть уже попадалась нам в составе других функций проверки, чем мы и воспользуемся.

```
function validateRegEx(regex, inputStr, helpText, helpMessage) {
  // Проверим корректность данных inputStr
  if (!regex.test(inputStr)) {
    // Данные некорректны, зададим текст подсказки и вернем false
    if (helpText != null)
      helpText.innerHTML = helpMessage;
    return false;
  }
  else {
    // Данные корректны, удалим текст подсказки и вернем true
    if (helpText != null)
      helpText.innerHTML = "";
    return true;
  }
}
```

В качестве аргументов передаются регулярные выражения, данные ввода, текст и элемент подсказки.

Регулярное выражение сравнивается с данными ввода.

Несоответствие условию означает некорректность данных, поэтому отобразим подсказку.

Если проверка пройдена, текст подсказки исчезает.

Возьми в руку карандаш

Напишите код функции `validateDate()`, которая для проверки введенной в поле на странице Bannerocity даты использует функции `validateNonEmpty()` и `validateRegEx()`. Подсказка: у этой функции два аргумента — данные ввода и элемент со вспомогательным текстом.

Возьми в руку карандаш

Решение

Вот как выглядит функция `validateDate()`, которая для проверки введенной в поле на странице Bannerocity даты использует функции `validateNonEmpty()` и `validateRegEx()`.

Сначала функция validateNonEmpty() проверяет, что поле не является пустым.

```
function validateDate(inputField, helpText) {
  // Проверяем, введены ли данные в поле
  if (!validateNonEmpty(inputField, helpText))
    return false;

  // Проверяем, являются ли эти данные датой
  return validateRegEx(/^\d{2}\/\d{2}\/\d{4}$/, inputField.value, helpText,
    "Пожалуйста, введите дату (например, 01/14/1975).");
}
```

Функции validateRegEx() передается регулярное выражение для даты.

Так как косая черта является метасимволом, ее заключают между обратными косыми чертами.

Для задания выражения формата ММ/ДД/ГГГГ используются как метасимволы, так и квантификаторы.

Интересно, будут ли пользоваться вашим сценарием в 2100 году?

А почему бы не допустить ввод года в виде только двух символов?

Проблема 2100 года еще так далеко...

Учитывая, что до следующего рубежа веков еще очень много времени, наверное, можно позволить пользователям вводить год в виде только двух цифр. Вряд ли какой-то из написанных в наши дни сценариев JavaScript проживет 90 лет и столкнется с проблемой. Говард сначала хотел для надежности в Bannerocity вводить год в виде четырех символов, но потом решил, что, если его детище и доживет до следующего века, тогда и можно будет внести в него исправления.

Часто Задаваемые Вопросы

В: Зачем вызывать функцию `validateNonEmpty()` внутри функции `validateDate()`? Разве регулярные выражения не умеют работать с пустыми полями?

О: Да, регулярные выражения умеют проверять наличие или отсутствие данных, поэтому упомянутую вами функцию можно удалить. Однако, проверяя, были ли введены данные, и отображая соответствующий вспомогательный текст, мы делаем страницу интуитивно понятной для пользователей. Отсутствие данных и ввод некорректных данных должны сопровождаться разными подсказками. Именно на этом принципе работает наша система помощи, позволяющая пользователю понять, как именно следует заполнять форму. И ради этого маленького усовершенствования можно написать несколько дополнительных строчек кода.

В: А что, если я хочу написать сценарий с учетом требований завтрашнего дня. Это сложно?

О: Вовсе нет. Предвосхищение будущих требований и написание кода с их учетом обычно не является проблемой. В случае с Bannerocity мы учтем будущие требования, оставив ввод года в виде четырех символов. Профессиональные программисты позволяют пользователям вводить только две последние цифры, добавляя первые две в тексте сценария. В результате данные, вводимые в форму, все равно будут храниться в виде четырех цифр.

Диапазон вхождений

Квантификатор {} используется для указания количества появлений шаблона в строке. Он может записываться и в другой форме, уже с двумя аргументами, определяющими минимально и максимально возможное количество повторений шаблона. Это дает еще один дополнительный инструмент настройки:

`{min,max}`

Указывает, сколько раз шаблон может появиться как минимум и как максимум.

Предшествующий шаблон должен появиться как минимум min раз подряд, но не больше, чем max раз.

```
/^\w{5,8}$/
```

Некоторые пароли позволяют вводить от пяти до восьми буквенно-цифровых символов, что замечательно описывается при помощи квантификатора {}.

Будьте осторожны!

ММ/ДД/ГГГГ — не единственный возможный формат дат.

В ряде стран более привычным является указание дня, а только затем месяца, то есть запись производится в формате ДД/ММ/ГГГГ.

Возьми в руку карандаш

Перепишите регулярное выражение из функции `validateDate()` таким образом, чтобы оно позволяло ввод года как в виде двух, так и в виде четырех цифр.

Возьми в руку карандаш

Решение

Вот как выглядит регулярное выражение из функции `validateDate()`, позволяющее вводить год как в виде двух, так и в виде четырех цифр.

```
/^\d{2}\/\d{2}\/\d{2,4}$/
```

В таком виде квантификатор {} задает минимальное и максимальное количество цифр при вводе года.

То есть получается, что код проверки позволяет вводить год тремя цифрами? Это же лишено смысла...

Регулярное выражение для даты действительно позволяет ввести в конце только три цифры!

Enter the date for the message to be shown: 03/01/200

Никакие ревизионисты не помогут вернуть JavaScript во времена до десятого века.

Соответственно, нам нет никакого смысла поддерживать ввод года в виде трех цифр. Убрав эту возможность, вы уменьшите количество потенциальных проблем в бизнесе Говарда.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Регулярное выражение совмещает **шаблон** с текстом строки.
- Кроме обычного текста в регулярных выражениях используются метасимволы и квантификаторы.
- В JavaScript регулярные выражения поддерживаются объектом `RegExp`, но обычно создаются в виде литералов.
- Метод `test()` объекта `RegExp` проверяет заданный регулярным выражением шаблон на соответствие строке текста.

Выбери это... или то

Другим полезным метасимволом для построения регулярных выражений является перечисление. По своему виду и функциональности перечисление напоминает оператор ИЛИ. Но, в отличие от этого оператора, для записи перечисления используется всего одна вертикальная черта |, разделяющая допустимые варианты. Другими словами, шаблон считается совпавшим при соответствии одному из двух указанных выражений.

```
ЭТО|ТО
```

Метасимвол перечисления позволяет реализовывать выбор из двух альтернатив.

Шаблон совпадает при совпадении с одним из указанных вариантов.

```
/small|medium|large/
```

Более сложные варианты выбора реализуются при помощи набора перечислений.

```
/(red|blue) pill/
```

Это выражение совпадает как со строкой «red pill», так и со строкой «blue pill».

Возьми в руку карандаш

Еще раз перепишите регулярное выражение из функции `validateDate()`, заставив его совпадать только со строками из двух или четырех символов.

.....

Возьми в руку карандаш

Решение

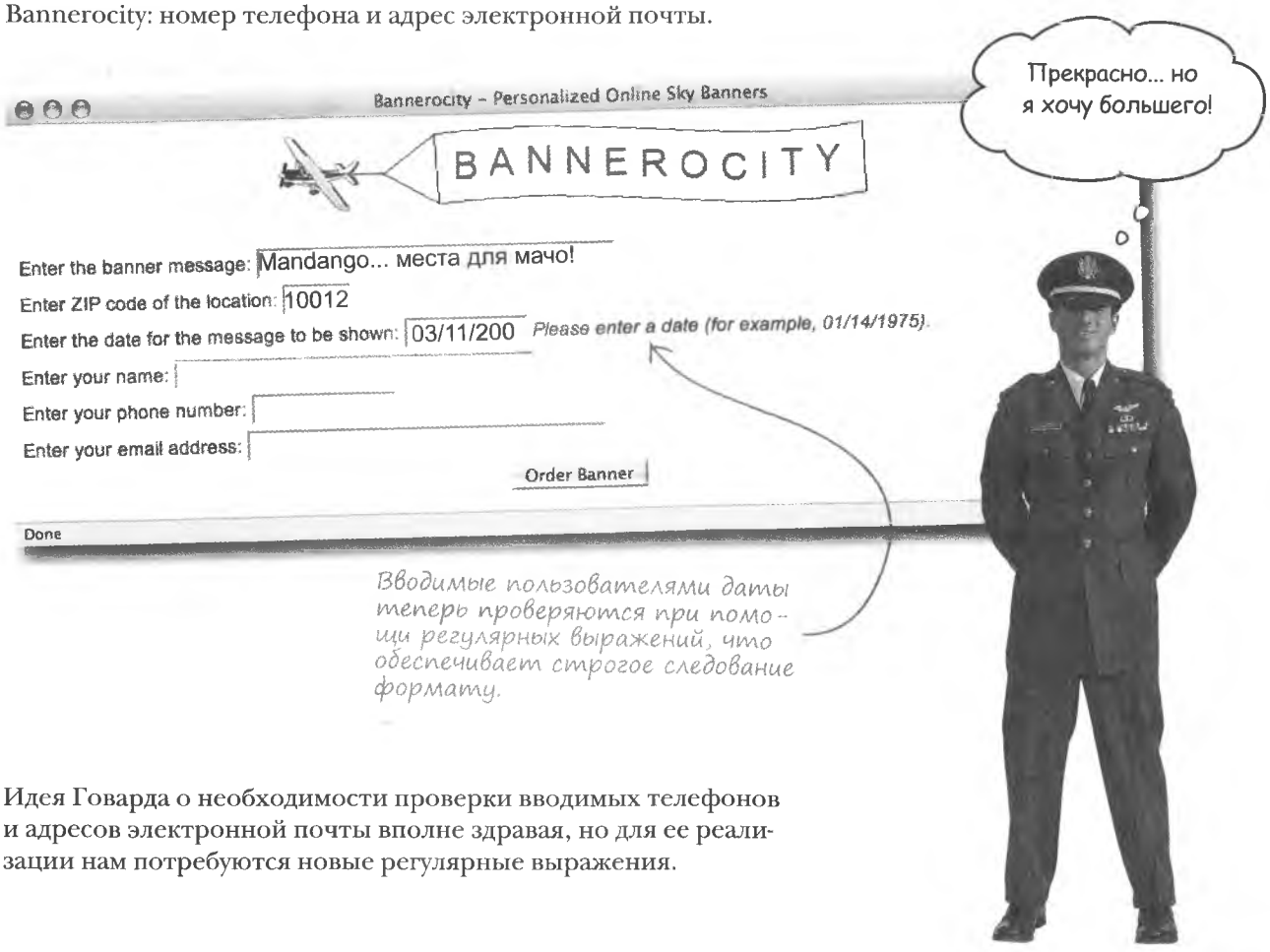
Вот как выглядит регулярное выражение из функции `validateDate()`, совпадающее только со строками из двух или четырех символов.

Метасимвол перечисления (|) задает шаблон для ввода года как в виде двух, так и в виде четырех цифр.

```
/^\d{2}\/\d{2}\/(\d{2}|\d{4})$/
```

Никаких случайностей

Говарду очень понравился новый, надежный способ проверки дат, обеспечивающий точное следование заданному шаблону. И он решил воспользоваться регулярными выражениями для проверки корректности ввода данных и в двух оставшихся полях формы Bannerocity: номер телефона и адрес электронной почты.



Идея Говарда о необходимости проверки вводимых телефонов и адресов электронной почты вполне здравая, но для ее реализации нам потребуются новые регулярные выражения.

Вы меня слышите?

Проверить корректность ввода телефонных номеров достаточно просто, так как они записываются в достаточно строгом формате. Разумеется, без регулярных выражений нам пришлось бы столкнуться с необходимостью деления строк, но нам этого делать не придется. Американские телефонные номера строятся по следующему шаблону:

Шаблон = ###-###-####

Так как Говард не собирается предлагать свои услуги иностранцам, можно ограничиться американским форматом телефонных номеров.

Заменим дефисы косыми чертами и укажем, из скольких цифр состоит каждый фрагмент. Получившееся регулярное выражение имеет практически такой же формат, как и в предыдущем случае.

Шаблон для даты был создан при помощи метасимвола \d и квантификатора {}.

```
/^\d{2}\/\d{2}\/\d{2,4}$/
```

Шаблон для номера телефона отличается от шаблона для даты наличием дефисов, разделяющих группы цифр.

```
/^\d{3}-\d{3}-\d{4}$/
```

Благодаря регулярным выражениям и функции `validateRegEx()` написать код функции `validatePhone()`, проверяющей номер телефона, несложно.

```
function validatePhone(inputField, helpText) {
  // Проверяем, были ли введены данные
  if (!validateNonEmpty(inputField, helpText))
    return false;

  // Проверяем, являются ли данные номером телефона
  return validateRegEx(/^\d{3}-\d{3}-\d{4}$/,
    inputField.value, helpText,
    "Пожалуйста, введите номер телефона (например, 123-456-7890).");
}
```

Вам письмо

Теперь, когда функция проверки номера телефона готова, у Говарда осталась только одна забота. Проверить корректность ввода адресов электронной почты в форму Bannerocity. Как и раньше, мы начнем решение этой задачи с написания регулярного выражения.

2, 3 или 4 алфавитно-цифровых символа.

Шаблон = *LocalName@DomainPrefix.DomainSuffix*

Здесь могут встречаться как буквы, так и цифры.

Все выглядит не так уж страшно — адрес электронной почты представляет собой всего лишь три текстовых фрагмента, которые могут быть составлены из букв латинского алфавита и цифр, а также символа (@) и набора точек.

Все эти адреса подходят под шаблон. Значит, дело сделано... или все-таки нет?

howard@bannerocity.com

sales@duncansdonuts.com

puzzler@youcube.ca

Создание шаблона для адреса электронной почты является несложным процессом.

Так как точка входит в число специальных символов, ее следует поместить между обратными косыми чертами.

Адрес должен начинаться с одного или нескольких буквенно-цифровых символов.

```
/^\w+@\w+\.\w{2,4}$/
```

В конце адреса электронной почты должны стоять 2, 3 или 4 алфавитно-цифровых символа.

После символа @ появляется еще один или несколько буквенно-цифровых символов.

Шаблон кажется вполне рабочим. Осталось понять, всегда ли адреса электронной почты записываются именно в таком формате.

МОЗГОВОЙ ШТУРМ

Возможны ли другие варианты шаблона для адреса электронной почты? Вспомните, какие еще адреса вы видели.

Исключение — это правило

Адреса электронной почты более сложны, чем это кажется на первый взгляд. При написании нами шаблона необходимо учесть существование других вариантов написания адреса электронной почты. Вот примеры совершенно корректных адресов:

cube_lovers@youcube.ca

Подчеркивание в имени почтового ящика.

aviator.howard@bannerocity.com

Точка в имени почтового ящика.

rocky@i-rock.mobi

Дефис в доменном имени.

Четыре символа в имени домена первого уровня.

i-love-donuts@duncansdonuts.com

Дефисы в имени почтового ящика.

Для проверки адресов электронной почты нам потребуются дополнительные символы.

seth+jason@mandango.us

Знак + в имени почтового ящика.

ruby@youcube.com.nz

Многоуровневое доменное имя.

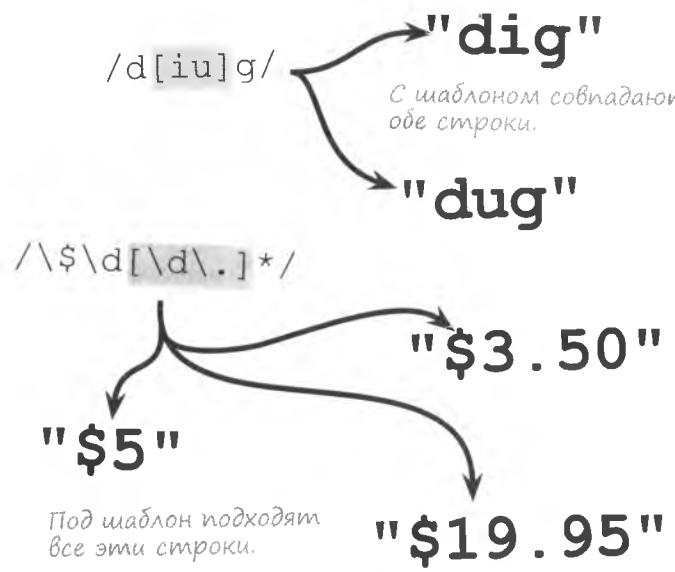
Чтобы описать все многообразие адресов электронной почты, нам не хватает символов.

Изначально мы считали, что фрагменты адреса электронной почты записываются только буквами и цифрами. Теперь в шаблон требуется вставить дополнительные символы...

Дополнительные символы

Еще одной функцией, которая непосредственно касается способа написания шаблонов для адресов электронной почты, являются **символьные классы**. Именно они позволяют задавать правила вставки дополнительных символов. Символьный класс можно представить в виде набора правил для представления одного символа.

Выбор подходящего символа осуществляется среди набора, указанного в квадратных скобках. Рассмотрим несколько примеров:



Символьные классы — это именно то, чего нам не хватало для приведения в порядок шаблона записи адресов электронной почты.

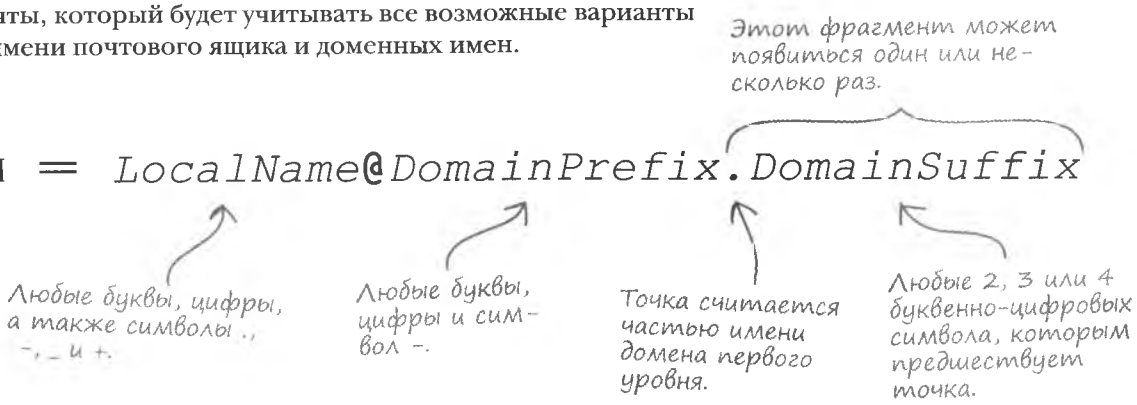
Символьные классы позволяют вставлять в шаблоны для регулярных выражений дополнительные символы.

Не забывайте про знаки перехода.

Если вам требуется включить в выражение символ, совпадающий с метасимволом, ставьте перед ним обратную косую черту. Такой чертой (\), в частности, предваряются символы: `[\^$.|?*+().`

Проверка адреса электронной почты

Теперь мы можем написать корректный шаблон для адресов электронной почты, который будет учитывать все возможные варианты написания имени почтового ящика и доменных имен.



Помните, что любой шаблон может быть создан несколькими способами. Это относится и к шаблону для адресов электронной почты. Ведь так сложно учесть все нюансы указанного формата данных. Впрочем, вы уже получили навыки конструирования шаблонов и убедились, насколько просто они преобразовываются в регулярные выражения.

Возьми в руку карандаш

Закончите код функции `validateEmail()`, предназначенной для проверки корректности ввода адресов электронной почты.

```
function validateEmail(inputField, helpText) {
  // Проверяем, были ли введены данные.
  if (!.....(inputField, helpText))
    return false;

  // Проверяем, являются ли данные адресом электронной почты
  return validateRegEx(.....,
    inputField.value, helpText,
    .....);
}
```

Возьми в руку карандаш

Решение

Вот как выглядит функция `validateEmail()`, проверяющая корректность ввода адресов электронной почты.

```
function validateEmail(inputField, helpText) {

  // Проверяем, были ли введены данные

  if (! validateNonEmpty (inputField, helpText))
    return false;

  // Проверяем, являются ли данные адресом электронной почты

  return validateRegEx( /^[\w\.-_\+]+@[\w-]+(\.\w{2,4})+$/ ,
    inputField.value, helpText,
    "Пожалуйста, введите адрес (например, johndoe@acme.com)." );
}
```

Мы продолжаем проверять отсутствие данных при помощи функции validateNonEmpty().

Шаблон для проверки адреса электронной почты был составлен с применением большинства методов, которые мы изучали в этой главе.

Доменное имя первого уровня может состоять как из 2, так и из 4 символов.

Имя почтового ящика может состоять из букв, цифр, таких символов, как ., -, _ и +, и должно стоять в начале строки.

Если проверка не будет пройдена, появится вспомогательное сообщение с образцом корректного адреса электронной почты.

Надежная форма

Благодаря усилиям, направленным на проверку вводимых в форму Bannerocity данных, заказы обрабатываются без проблем. Говард так счастлив, что по собственной инициативе показал вот такой баннер.

Проверка данных — это полезное изобретение!

Говард с восторгом возвращается к любимому занятию — полетам!

Вводимые номера телефонов и адреса электронной почты теперь проверяются на соответствие заданному формату данных.

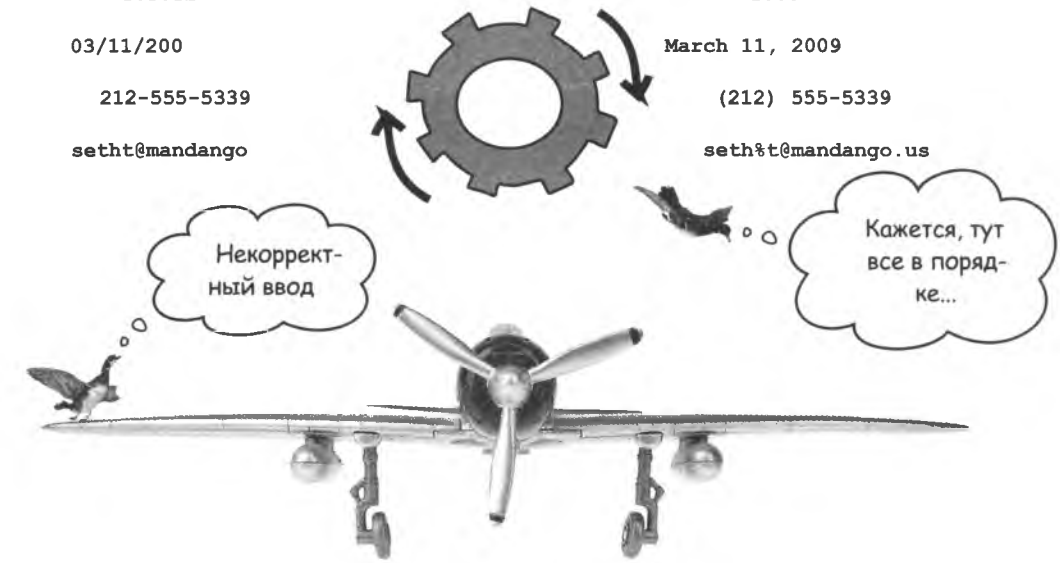
Enter the banner message: Mandango... места для
Enter ZIP code of the location: 10012
Enter the date for the message to be shown: 03/11/2009
Enter your name: Seth Tinselman
Enter your phone number: (212) 555-5339 *Please enter a phone number (for example, 123-456-7890).*
Enter your email address: setht@mandango *Please enter an email address (for example, johndoe@acme.com).*
Order Banner

Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Зачем JavaScript веб-формам?

Vи хорошо, а два лучше!




```
/^val(ley|ue|krie)/
```

```
/name|id$/
```

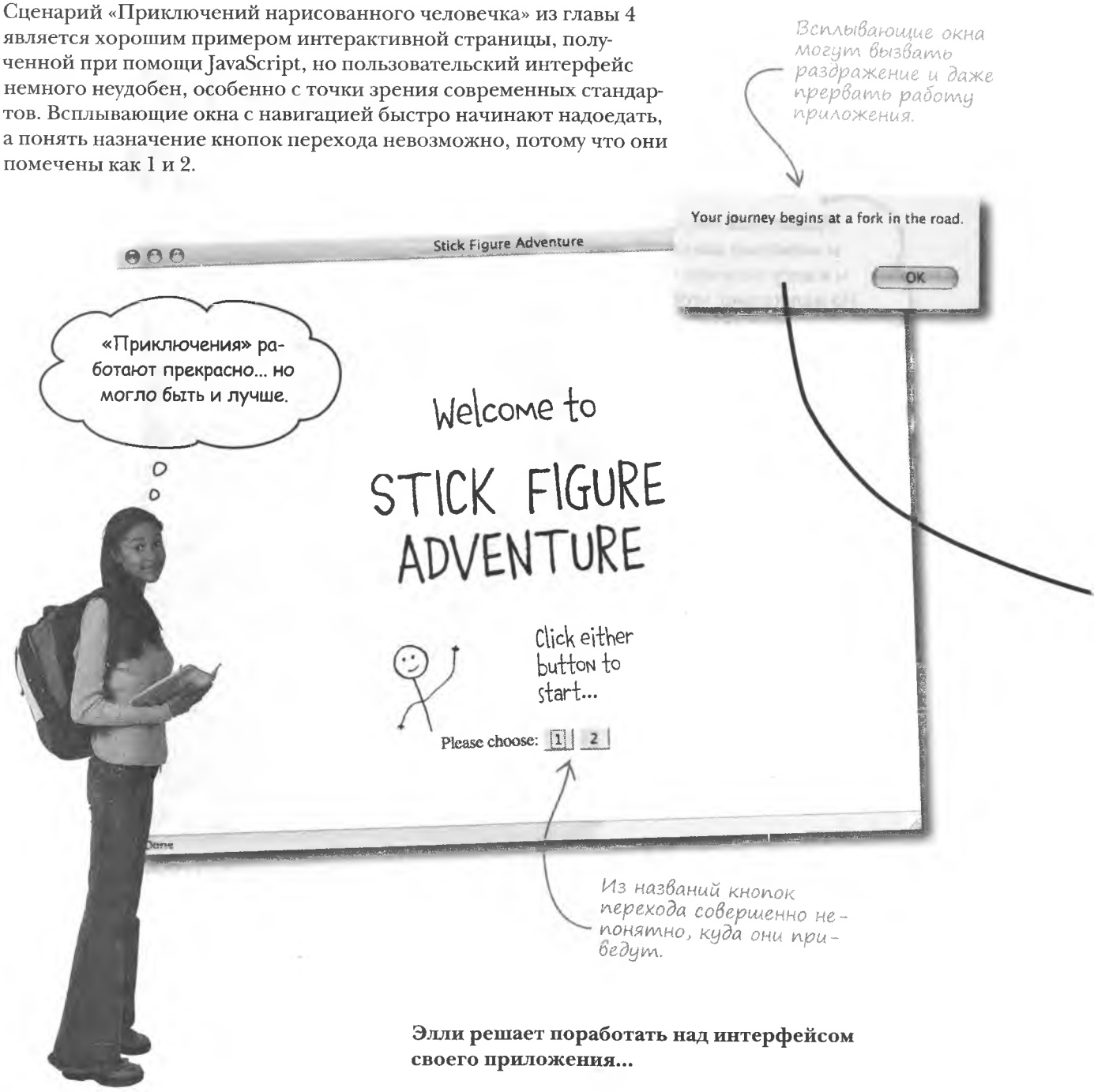
JavaScript может так много предложить веб-формам, что сложно даже выбрать что-то одно. Ответ на вопрос в заголовке почти всегда включает в себя какой-то уровень данных. Но как они используются?

Управление HTML с DOM

Управление содержимым веб-страницы при помощи JavaScript напоминает приготовление еды. Конечно, это не настолько грязное занятие... И, увы, вы не сможете съесть результат своих трудов. Тем не менее вы получаете ***полный доступ к HTML***-ингредиентам, из которых состоит веб-страница, и, что еще важнее, вы можете ***менять*** исходный рецепт. Ведь **JavaScript дает возможность управлять HTML-кодом веб-страницы**, что открывает для вас целый ряд интереснейших перспектив, которые реализуются посредством ***набора стандартных объектов*** **DOM**.

Функциональный, но неудобный

Сценарий «Приключений нарисованного человечка» из главы 4 является хорошим примером интерактивной страницы, полученной при помощи JavaScript, но пользовательский интерфейс немного неудобен, особенно с точки зрения современных стандартов. Всплывающие окна с навигацией быстро начинают надоедать, а понять назначение кнопок перехода невозможно, потому что они помечены как 1 и 2.

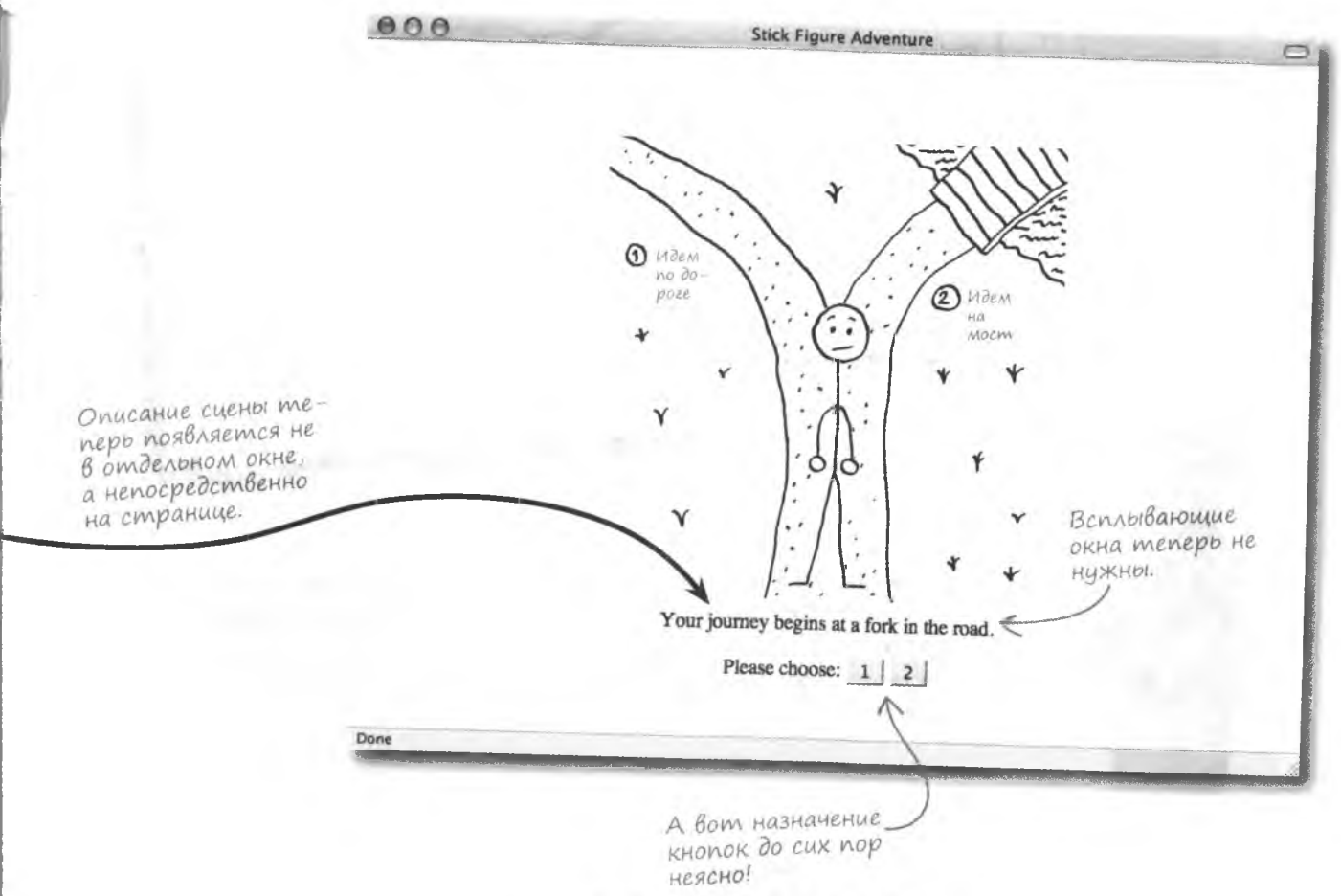


Элли решает поработать над интерфейсом своего приложения...

Без всплывающих окон

Всплывающие окна с описанием сцены плохи еще и тем, что текст исчезает после щелчка на кнопке OK. Лучше поместить описание непосредственно на страницу. Вот как после этого будут выглядеть «Приключения»:

Новый набор файлов для «Приключений нарисованного человечка» доступен по адресу http://www.headfirstlabs.com/books/hfjs/.



МОЗГОВОЙ ШТУРМ

Какими средствами JavaScript лучше всего реализовать новый подход к описанию сцен?

Тег div

Чтобы отобразить на странице описание сцены, сначала нужно выделить под него область страницы в виде HTML-элемента. Так как описание появляется в виде отдельного параграфа, задача решается при помощи тега .

Тег имеет атрибут ID, уникальным образом задающий элемент, в котором будет располагаться текст описания.

```
<body>

  <div style="margin-top:100px; text-align:center">
   <img id="sceneimg" src="scene0.png" alt="Stick Figure Adventure" /><br />
   <div id="scenetext"></div><br />
   Please choose:
   <input id="decision1" value="1" onclick="changeScene(1)" />
   <input id="decision2" value="2" onclick="changeScene(2)" />
```

Дает ли атрибут ID тега доступ к описанию сцены?

Именно при помощи атрибута ID осуществляется доступ к элементам страницы, в том числе и к описанию сцены.

И именно атрибут `id` тега даст коду JavaScript доступ к описанию. Кстати, подобную операцию вы уже проделывали...

Будьте осторожны!

Идентификаторы должны быть уникальными.

Не забудьте, что атрибут `id` *предназначен для уникальной идентификации элементов страницы.*

Доступ к HTML-элементам

С методом `getElementById()` стандартного объекта `document` мы уже не раз сталкивались. Именно он дает доступ к HTML-элементам страницы... При условии наличия у них уникального ID.

Это должно совпадать с атрибутом id элемента HTML, в нашем случае тега div.

```
var sceneDesc = document.getElementById("scenetext");
```

Доступ к элементу div осуществляется при помощи его атрибута id.

```
<img id="sceneimg" src="scene0.png" ...
<div id="scenetext"></div><br />
Please choose:
```

Имея в руках элемент с описанием сцены, мы можем управлять его содержимым. Но сначала познакомимся с методом `getElementsByTagName()`, коллекционирующим все элементы страницы с определенным тегом, например `div` или `img`, и помещающим их в массив в порядке появления на странице.

Имя тега, без <>.

```
var divs = document.getElementsByTagName("div");
```

Упражнение

Напишите код JavaScript, предоставляющий доступ к оранжевому изображению. Сначала при помощи метода `getElementById()`, а затем — при помощи метода `getElementsByTagName()`.

```
<body>
  <p>Выберите уровень сложности Приключений:</p>
  <img id="green" src="green.png" alt="Расслабленный" /><br />
  <img id="blue" src="blue.png" alt="Раздражающий" /><br />
  <img id="yellow" src="yellow.png" alt="Изматывающий" /><br />
  <img id="orange" src="orange.png" alt="Пугающий" /><br />
  <img id="red" src="red.png" alt="Сумасшедший" />
</body>
```

Методом `getElementById()`: .....

Методом `getElementsByTagName()`: .....

Упражнение Решение

Вот как выглядят варианты кода JavaScript, предоставляющие доступ к оранжевому изображению.

```
<body>
  <p>Before starting, please choose an adventure stress level:</p>
  <img id="green" src="green.png" alt="Relaxing" /><br />
  <img id="blue" src="blue.png" alt="Irritating" /><br />
  <img id="yellow" src="yellow.png" alt="Frazzled" /><br />
  <img id="orange" src="orange.png" alt="Panicked" /><br />
  <img id="red" src="red.png" alt="Maddening" />
</body>
```

Оранжевое изображение является четвертым элементом массива, поэтому его индекс 3.

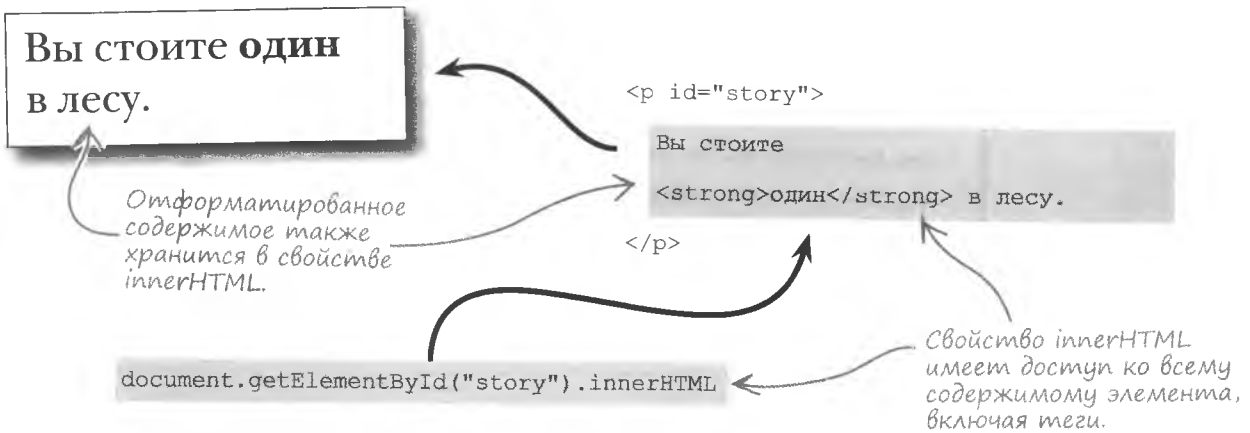
Методом `getElementById()`: *document.getElementById("orange")*

Методом `getElementsByTagName()`: *document.getElementsByTagName("img")[3]*

Внутренний код элемента

Под доступом к элементу HTML подразумевается доступ к содержащейся в нем информации. В случае HTML-элементов, содержащих текст, таких как `div` и `p`, это осуществляется при помощи свойства `innerHTML`.

Свойство innerHTML дает доступ к содержимому элементов.



Создается впечатление, что можно не только легко прочитать содержимое HTML-элемента, но и записать туда новые данные.

Свойство innerHTML применяется и для задания содержимого страницы.

Ведь оно позволяет осуществлять не только чтение, но и запись информации. Достаточно назначить этому свойству строку текста, и содержимое контейнера элемента будет заменено новым.

```
document.getElementById("story").innerHTML =
  "Вы <strong>не</strong> один!";
```

Содержимое элемента задается или, как в данном случае, замещается путем назначения строки свойству innerHTML.

Вы **не** один!

Возьми в руку карандаш

Воспользовавшись свойством `innerHTML`, напишите код, присваивающий текст элементу, содержащему описание сцены.

.....

Возьми в руку карандаш

Решение

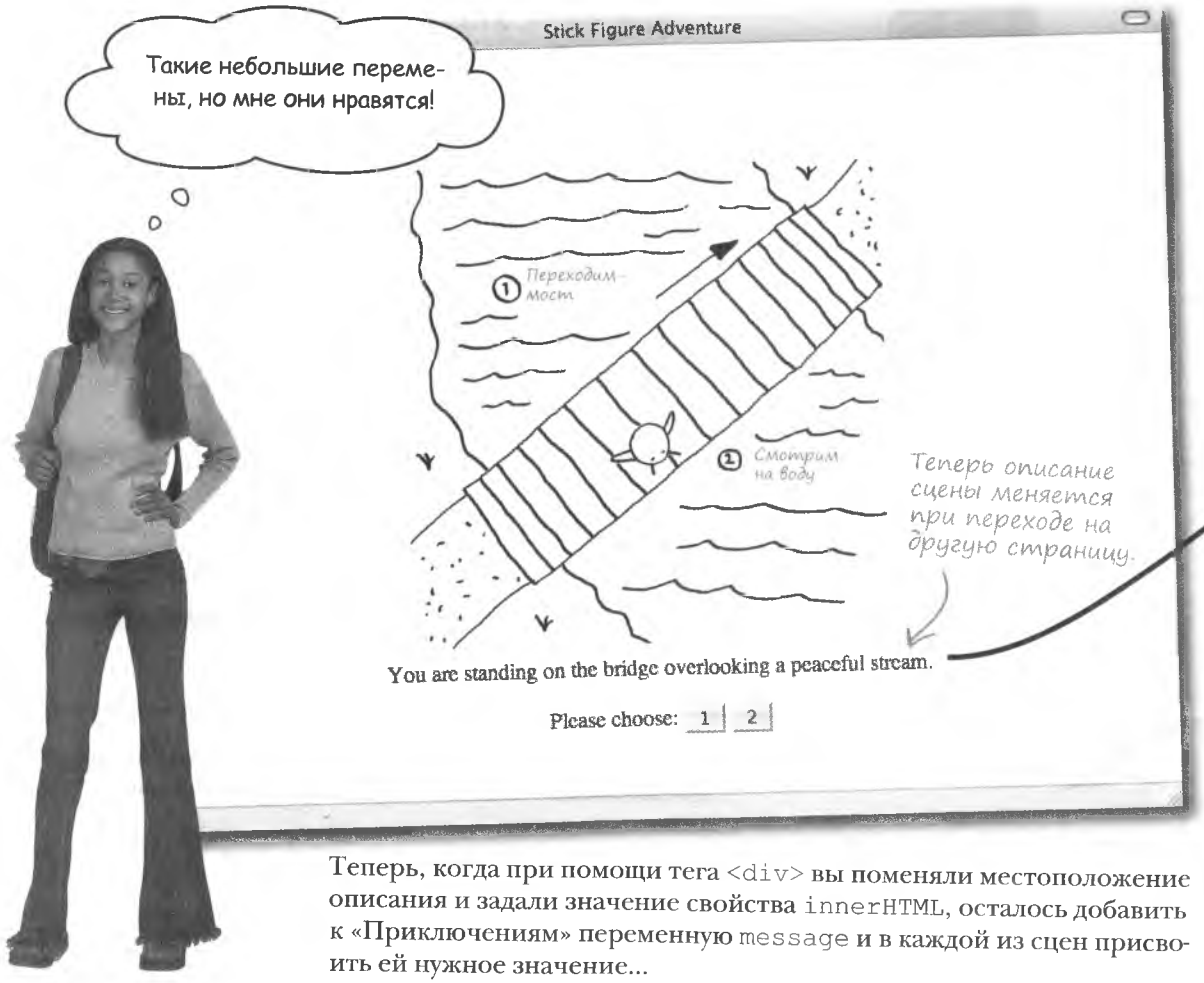
Вот как выглядит код, присваивающий текст элементу, содержащему описание сцены, написанный при помощи свойства `innerHTML`.

Атрибут ID тега с сообщением — это "sceneText".

```
document.getElementById("scenetext").innerHTML = message;
```

Непрерывное «Приключение»

Динамическое изменение описания сцен делает интерфейс «Приключений» более удобным и плавным. Ведь теперь вам не придется отвлекаться на всплывающие окна.



Теперь, когда при помощи тега `<div>` вы поменяли местоположение описания и задали значение свойства `innerHTML`, осталось добавить к «Приключениям» переменную `message` и в каждой из сцен присвоить ей нужное значение...

Подробно про код «Приключений»

```
<html>
  <head>
    <title>Приключения нарисованного человечка</title>

    <script type="text/javascript">
      // В качестве текущей выбираем сцену 0 (Введение)
      var curScene = 0;

      function changeScene(decision) {
        // Удаляем описание сцены
        var message = "";

        switch (curScene) {
        case 0:
          curScene = 1;
          message = "Your journey begins at a fork in the road.";
          break;
        case 1:
          if (decision == 1) {
            curScene = 2
            message = "You have arrived at a cute little house in the woods.";
          }
          else {
            curScene = 3;
            message = "You are standing on the bridge overlooking a peaceful stream.";
          }
          break;
        ...
        }

        // Обновляем изображение сцены
        document.getElementById("sceneimg").src = "scene" + curScene + ".png";

        // Обновляем описание сцены
        document.getElementById("scenetext").innerHTML = message;
      }
    </script>
  </head>

  <body>
    <div style="margin-top:100px; text-align:center">
      <img id="sceneimg" src="scene0.png" alt="Stick Figure Adventure" /><br />
      <div id="scenetext"></div><br />
      Please choose:
      <input type="button" id="decision1" value="1" onclick="changeScene(1)" />
      <input type="button" id="decision2" value="2" onclick="changeScene(2)" />
    </div>
  </body>
</html>
```

Локальная переменная message будет хранить описание каждой новой сцены.

Переменной message присваивается текст с описанием текущей сцены.

Текст описания присваивается переменной message при помощи свойства innerHTML.

Часто Задаваемые Вопросы

В: Предоставляет ли метод `getElementById()` доступ к элементам страницы?

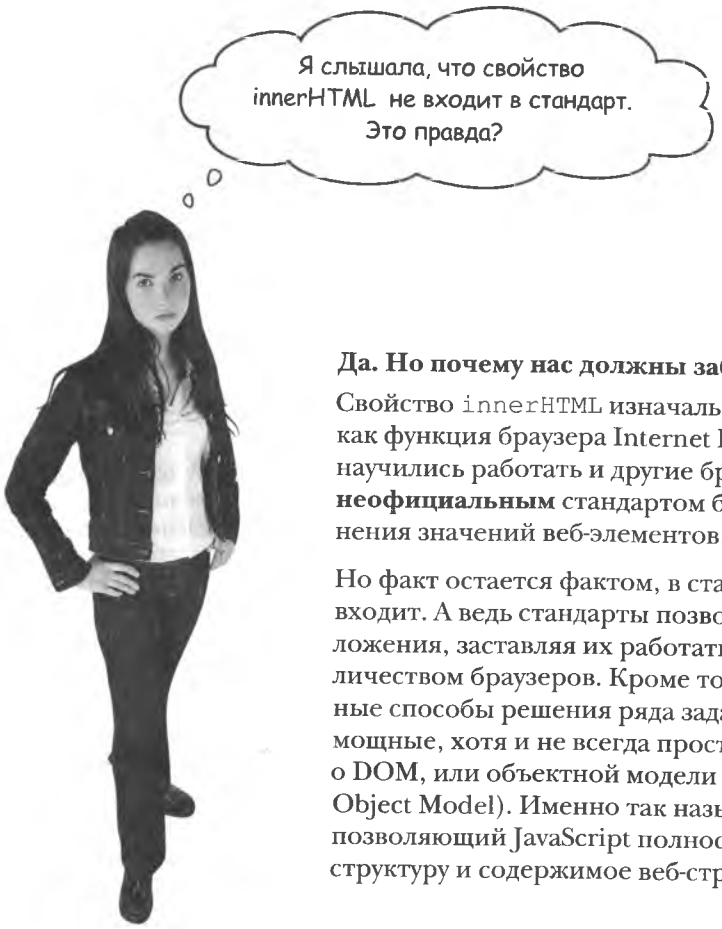
О: Да, но только при условии уникального значения атрибута `id`. Именно этот атрибут делает возможным применение метода `getElementById()`.

В: Позволяет ли свойство `innerHTML` задавать значение любых HTML-элементов?

О: Нет. Ведь далеко не все HTML-элементы имеют какое-то значение. Поэтому свойство `innerHTML` применяется только для задания значений таких элементов, как `div`, `span`, `p` и прочих контейнеров.

В: Что происходит со значением, когда оно задается свойством `innerHTML`?

О: Свойство `innerHTML` полностью переписывает предыдущее значение атрибута. То есть вы не можете **добавить** значение в `innerHTML`. Хотя можно осуществить соединение старого и нового значений и уже затем назначить его при помощи свойства `innerHTML`. Это может выглядеть, например, так: `elem.innerHTML += " Это предложение прибавляется.`



Да. Но почему нас должны заботить стандарты?

Свойство `innerHTML` изначально было создано Microsoft как функция браузера Internet Explorer. Постепенно с ним научились работать и другие браузеры, и свойство стало **неофициальным** стандартом быстрого и удобного изменения значений веб-элементов.

Но факт остается фактом, в стандарт `innerHTML` не входит. А ведь стандарты позволяют унифицировать приложения, заставляя их работать с как можно большим количеством браузеров. Кроме того, существуют стандартные способы решения ряда задач, более гибкие и более мощные, хотя и не всегда простые. Мы говорим сейчас о DOM, или объектной модели документа (Document Object Model). Именно так называется набор объектов, позволяющий JavaScript полностью контролировать структуру и содержимое веб-страниц.

Объектная модель документа (DOM)

Объектная модель документа предлагает удобный с точки зрения сценария взгляд на структуру и содержимое веб-страниц, благодаря которому возможно их динамическое редактирование средствами JavaScript. В DOM страницы представляются в виде иерархического дерева элементов. Каждый листок этого дерева называется **узлом** и непосредственно связан с каким-то элементом страницы. Узлы, расположенные снизу, считаются **дочерними** по отношению к вышестоящим.

Да, это дерево имеет странный вид, но именно так принято представлять набор узлов, формирующий страницу.

```
<html>
  <head></head>

  <body>
    <p id="story">
      Вы стоите <strong>один</strong> в лесу.
    </p>
  </body>
</html>
```

Верхний узел любого дерева DOM называется Document и располагается сразу над элементом HTML.

Это все узлы.

DOM «видит» страницу как иерархию узлов.

Пробелы, окружающие тег <p> на странице, интерпретируются как пустой текст.

Выделенный жирным текст «один» расположен под узлом тега .

Страница как набор узлов

В модели DOM существуют различные типы узлов. Основные типы соответствуют структурной части страницы и главным образом состоят из узлов `element` и `text`.

Узлы DOM классифицируются в соответствии с их типом.

DOCUMENT

Самый верхний узел дерева, представляющий собственно документ и стоящий над элементом `html`.

ELEMENT

Любой HTML-элемент, соответствующий HTML-тегу.

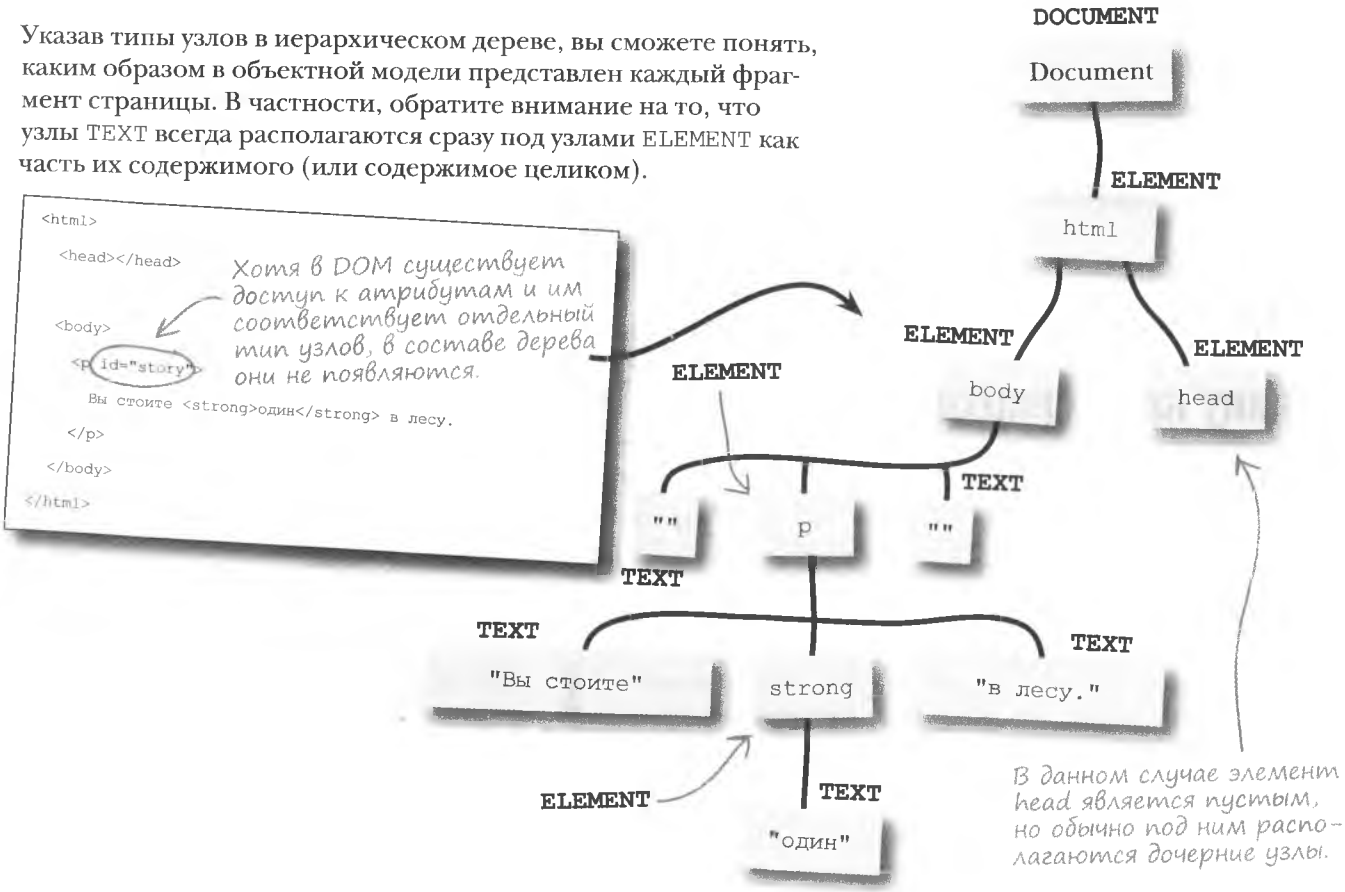
TEXT

Текстовое значение элемента, всегда хранящееся в дочернем для узла `element`-узле.

ATTRIBUTE

Атрибут элемента. В иерархическом дереве не фигурирует, но доступен через узел `element`.

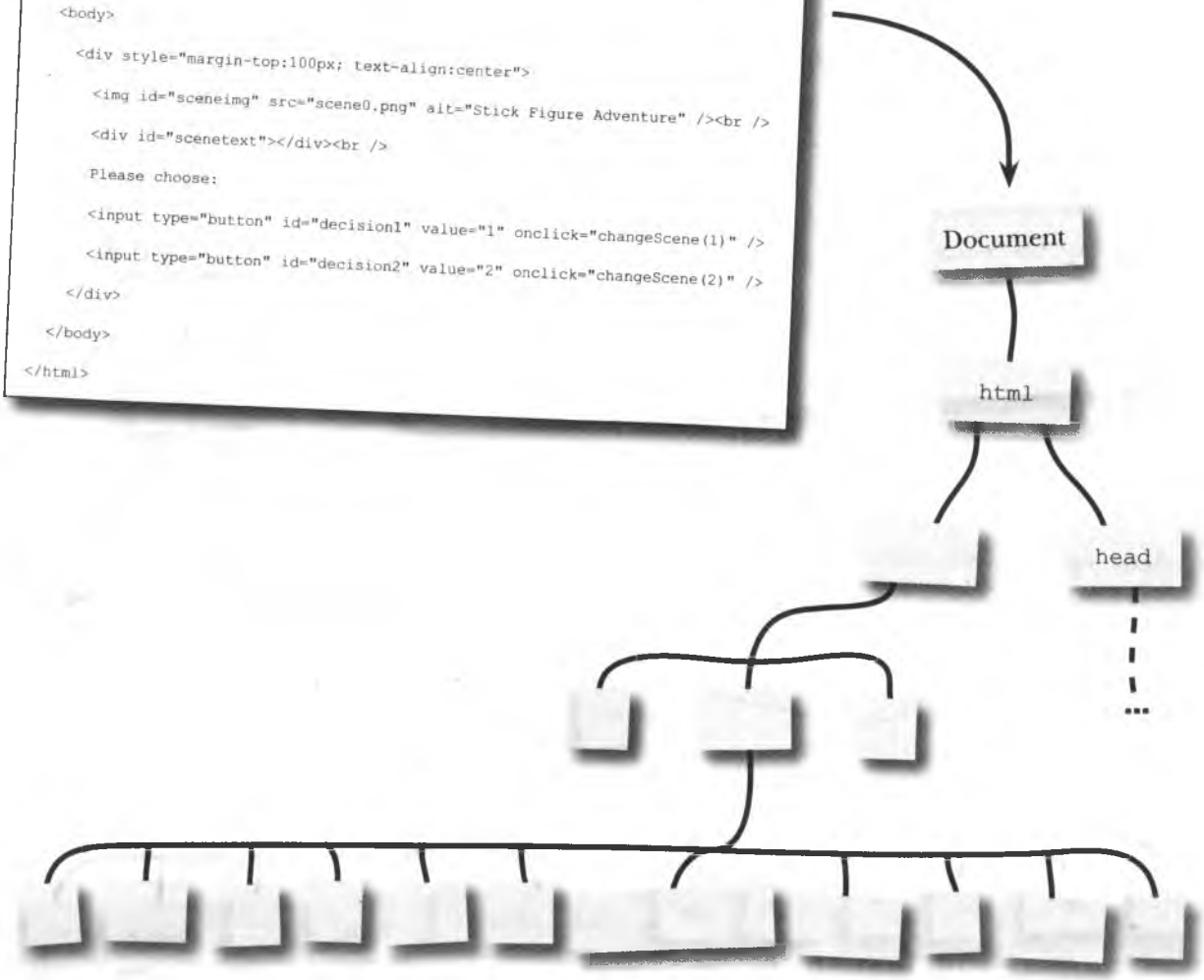
Указав типы узлов в иерархическом дереве, вы сможете понять, каким образом в объектной модели представлен каждый фрагмент страницы. В частности, обратите внимание на то, что узлы TEXT всегда располагаются сразу под узлами ELEMENT как часть их содержимого (или содержимое целиком).



Возьми в руку карандаш

Нарисуйте иерархическое дерево для показанного ниже HTML-кода, написав имя и тип каждого узла.

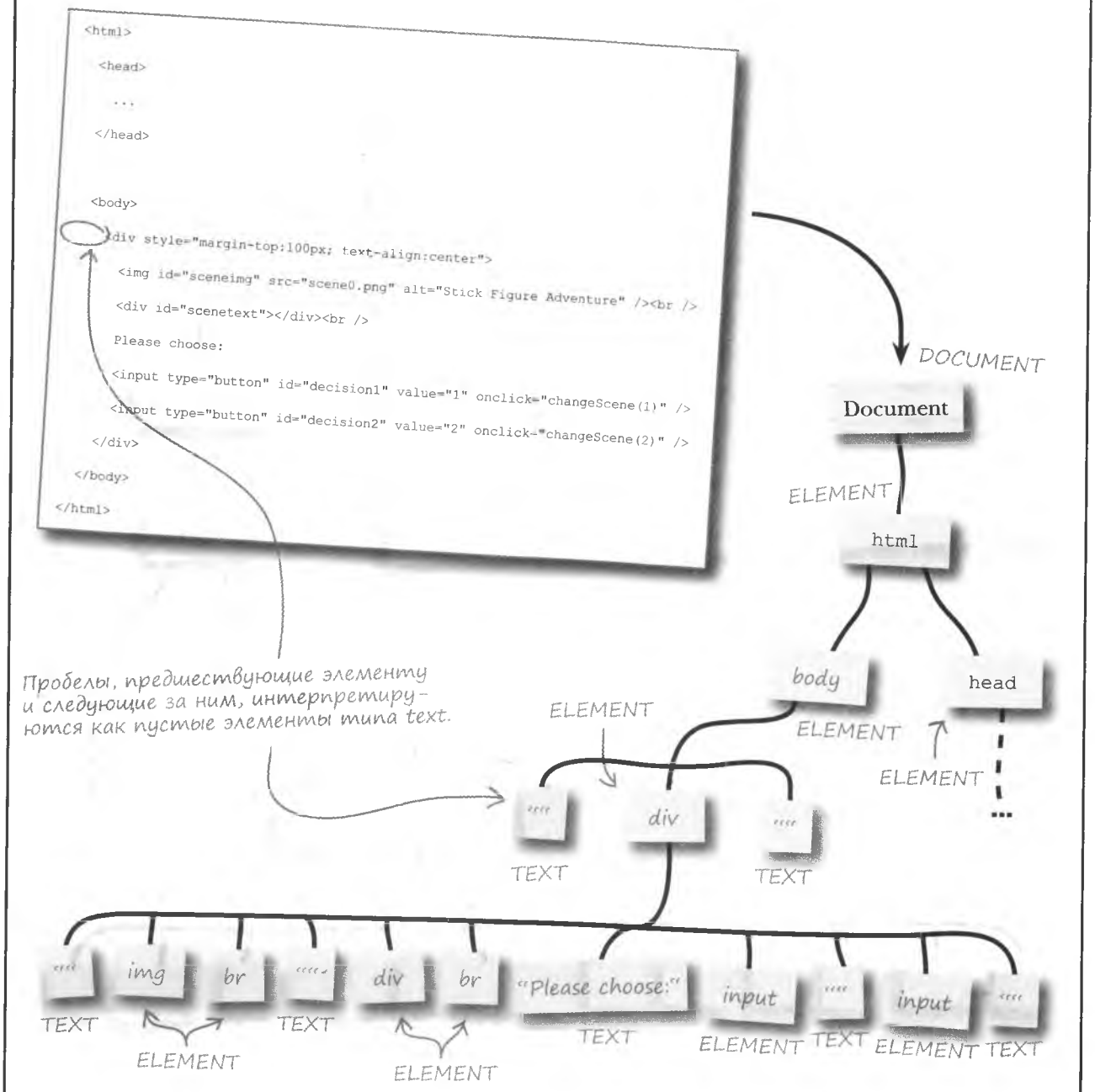
```
<html>
  <head>
    ...
  </head>

  <body>
    <div style="margin-top:100px; text-align:center">
      <img id="sceneimg" src="scene0.png" alt="Stick Figure Adventure" /><br />
      <div id="scenetext"></div><br />
      Please choose:
      <input type="button" id="decision1" value="1" onclick="changeScene(1)" />
      <input type="button" id="decision2" value="2" onclick="changeScene(2)" />
    </div>
  </body>
</html>
```



Возьми в руку карандаш

Решение

Вот как выглядит дерево иерархии узлов для показанного ниже кода.



Свойства узлов

Свойства узлов позволяют перемещаться по дереву DOM.

В большинстве случаев работа с DOM начинается с объекта `document`, который располагается на самом верху иерархического дерева. Этот объект имеет такие методы, как `getElementById()` и `getElementsByTagName()`, а также набор свойств. Доступ ко многим из этих свойств возможен из любого узла дерева. Некоторые из объектов допускают перемещения к другим узлам. То есть свойства узлов позволяют двигаться по их дереву.

nodeValue
Значения хранятся в текстовых узлах и узлах атрибутов, но не в узлах элементов.

nodeType
Тип узла, например `DOCUMENT` или `TEXT`, выраженный числом.

childNodes
Массив, содержащий перечень всех дочерних узлов в порядке их появления в коде страницы.

firstChild
Первый дочерний узел.

lastChild
Последний дочерний узел.

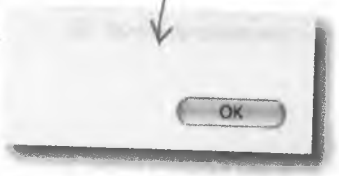
Эти свойства дают возможность перемещаться по иерархическому дереву и получать доступ к данным отдельных узлов. Например, для быстрого выделения нужного узла достаточно воспользоваться его свойствами и методом `getElementById()`.

```
alert(document.getElementById("scenetext").nodeValue);
```

Свойство nodeValue содержит только неформатированный текст.

Свойство nodeValue дает доступ к хранящемуся в узле тексту.

Текст описания сцены в «Приключении» изначально пустой.



Наверное, мы выбрали для демонстрации не лучший пример, ведь в исходной сцене «Приключений» тег `div` не содержит текста. Но уже для следующих сцен появится текст с описанием, и ситуация будет выглядеть намного лучше.

Упражнение

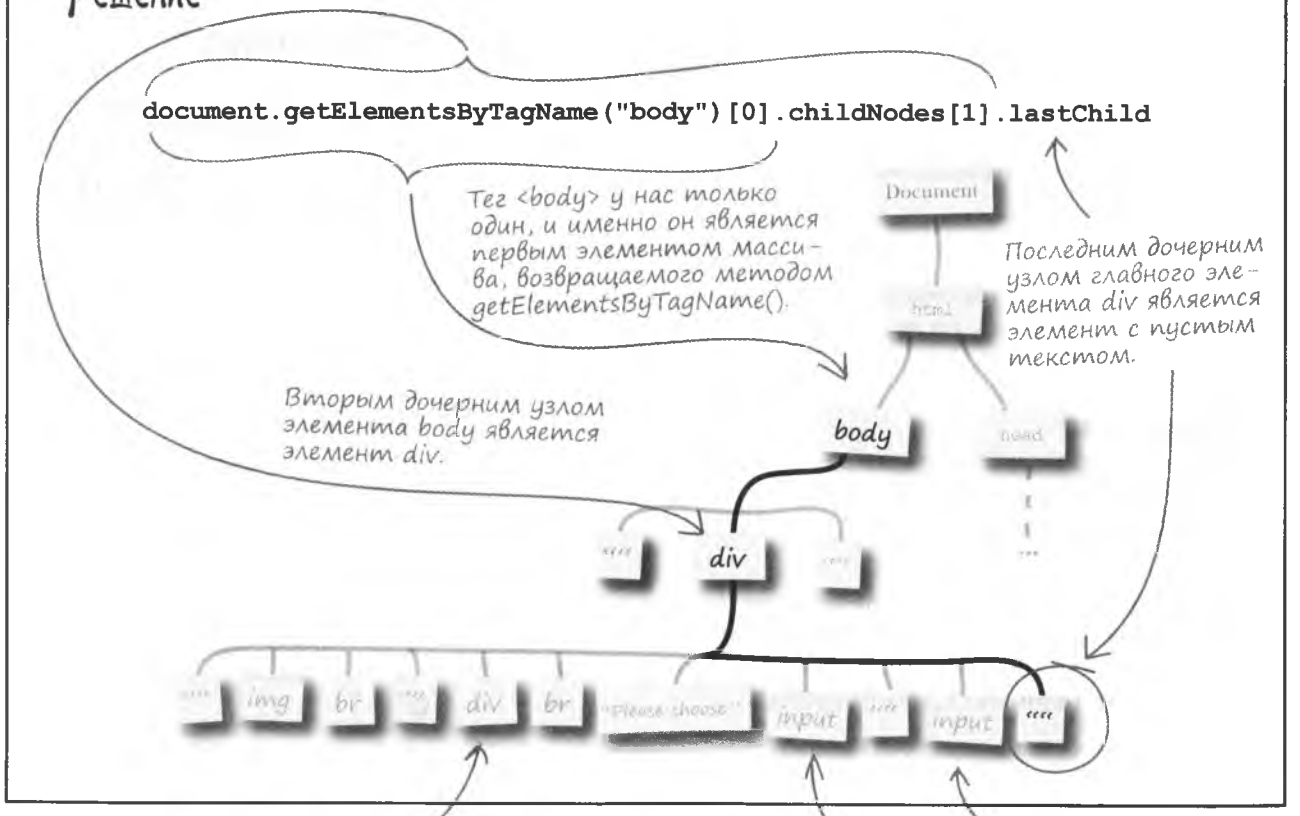
Вот код одного из узлов дерева со страницы 376. Рассмотрите его внимательно и укажите, какие узлы в нем упоминаются.

```
document.getElementsByTagName("body")[0].childNodes[1].lastChild
```

Упражнение Решение

Вот какие узлы нашего иерархического дерева упоминаются в показанной ниже строчке кода.

```
document.getElementsByTagName("body")[0].childNodes[1].lastChild
```



Часто Задаваемые Вопросы

В: В чем разница между методами `getElementById()` и `getElementsByTagName()` при работе с деревом DOM?

О: Выбор метода зависит от того, нужно ли вам выделить **один** элемент или же **группу** сходных элементов. В первом случае следует использовать метод `getElementById()` — достаточно указать идентификатор элемента.

Если же вашей целью является группа узлов, лучше воспользоваться методом `getElementByTagName()`.

Например, чтобы скрыть все изображения на странице средствами JavaScript, нужно сначала вызвать метод `getElementsByTagName()` и передать ему аргумент `"img"`, выделив таким образом все узлы изображений. Затем останется изменить свойство `visibility` языка CSS каждого из элементов. Но мы немножко забежали вперед, к DOM и CSS мы вернемся позднее. На данный момент вам достаточно знать, что, хотя метод `getElementsByTagName()` применяется реже метода `getElementById()`, в некоторых ситуациях без него не обойтись.

Свойства DOM позволяют редактировать содержимое веб-страниц и поддерживать соответствие стандартам.

Так как в DOM веб-страница представлена в виде набора узлов, именно их редактирование приводит к изменению ее содержимого. Текст в таких элементах, как `div`, `span` или `p`, всегда фигурирует в виде дочернего узла или узлов. Если текст содержится в едином узле, без дополнительных элементов HTML, значит, мы имеем всего один дочерний узел. Рассмотрим пример:

```
document.getElementById("story").firstChild.nodeValue
```

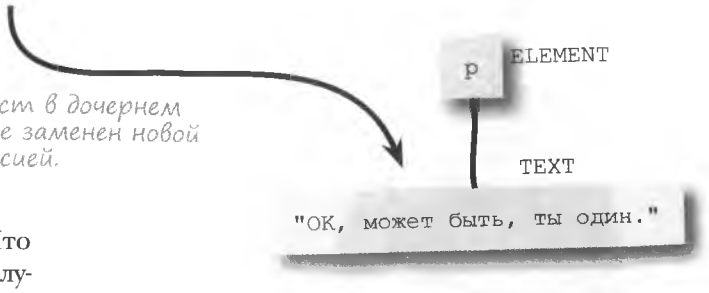
МОЗГОВОЙ ШТУРМ

Каким образом DOM позволяет менять текст?

Редактирование текста

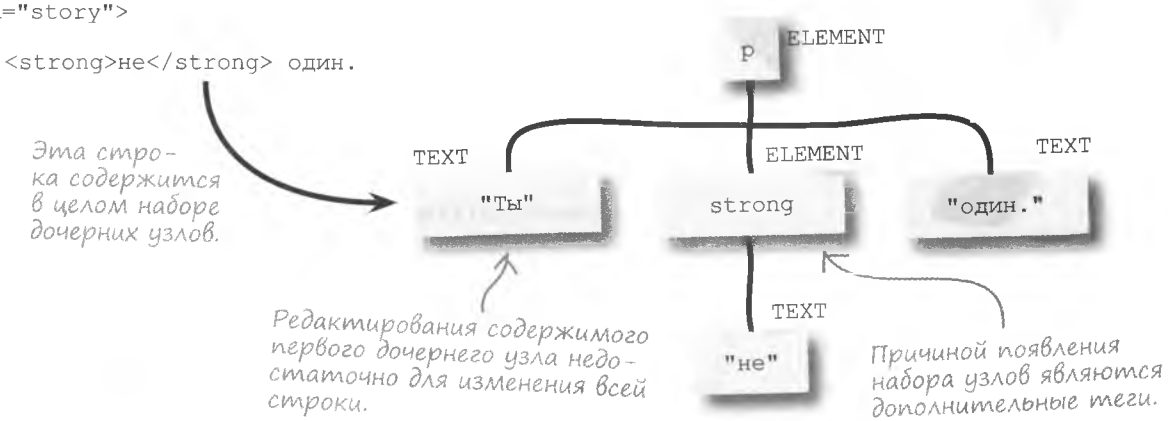
Если для простоты предположить, что существует только один дочерний узел, который и содержит текст, то ничто не помешает нам назначить этому узлу новый текст при помощи свойства `nodeValue`. Но еще раз напомним, что данный подход работает только при наличии единственного дочернего узла.

```
document.getElementById("story").firstChild.nodeValue = "OK, может быть ты один.";
```



Но проблема не всегда решается так просто. Что делать, если дочерних узлов несколько, как в случае вот такого кода?

```
<p id="story">
  Ты <strong>не</strong> один.
</p>
```



При замене содержимого первого дочернего узла остальные узлы не изменятся, и мы получим странный результат:

```
document.getElementById("story").firstChild.nodeValue = "OK, может быть ты один.";
```

Три шага для изменения

Итак, мы убедились, что замена значения одного узла не влияет на состояние остальных. Получается, что для редактирования информации в данном случае нужно сначала удалить все дочерние узлы, а затем добавить новые, уже с новыми значениями.

1. Удаляем все дочерние узлы.
2. Создаем текстовый узел с новым значением.
3. Добавляем его в качестве дочернего узла.

Для этого нам потребуются три метода:

1 removeChild()
Удаляет дочерний узел, переданный ему в качестве аргумента.

2 createTextNode()
Из строки создает текстовый узел.

3 appendChild()
Добавляет переданный ему узел в качестве последнего дочернего узла.

Для редактирования текста «ты не один» нам потребуются именно эти методы. Сначала убедимся, что все дочерние узлы удалены, затем создадим новый текстовый узел и, наконец, добавим его к строке.

```
var node = document.getElementById("story");
while (node.firstChild) 1
  node.removeChild(node.firstChild);
node.appendChild(document.createTextNode("OK, может быть ты один."));
```

Выделим элемент (узел) по его ID.

Удаляем дочерний узел, пока не очистим весь массив.

3 После удаления всех дочерних узлов добавим родителю новый текстовый узел.

2 Создадим новый текстовый узел.

OK, может быть, ты один.

СТРОИТЕЛЬНЫЕ ЭЛЕМЕНТЫ DOM

Интервью недели: Загадочные, но мощные регулярные выражения

Head First: Мне говорили, что вы являетесь самым маленьким кирпичиком в DOM, своего рода атомом в HTML-содержимом. Это так?

Узел: Не знаю, насколько я атомарен, но не могу не согласиться, что в дереве DOM я представляю именно дискретные фрагменты информации. Представим, что DOM разбивает каждую веб-страницу на байтовые кусочки... Вот таким кусочком я и являюсь!

Head First: А почему это имеет такое значение? Зачем нужна возможность деления страницы на фрагменты?

Узел: Это важно только в случаях, когда вам требуется доступ к информации или возможность ее редактирования.

Head First: А не рискуем ли мы в процессе такого деления потерять какой-нибудь фрагмент? Ведь часто люди разбирают что-то на части, а потом оказывается, что вещь банально сломана.

Узел: При работе с DOM такой проблемы не возникает, ведь вам не приходится в буквальном смысле разбирать страницу. DOM всего лишь представляет ее в виде иерархического дерева.

Head First: Даже так? А вы, я так понимаю, выходите на сцену, когда требуется что-то упростить?

Узел: Да. Но речь не идет только об упрощении, к дереву можно ведь и добавить данные.

Head First: Потрясающе! А как все это работает?

Узел: Помните, что каждый фрагмент информации является узлом? Вот через узлы и осуществляется доступ к содержимому страниц. Можно создать новый узел и добавить его к дереву. DOM — крайне гибкая структура.

Head First: А как вы связаны с элементами? Или это просто еще одно ваше имя?

Узел: По большому счету, да. Элемент — это всего лишь еще один способ восприятия тегов, например `<div>` или `<span>`. Все элементы страницы представлены на иерархическом дереве в виде узлов, так что можно сказать, что мы одно целое. Но я могу представлять не только элемент, но и хранящееся в нем содержимое. Так, тексту, хранящемуся в контейнере `<div>`, соответствует собственный узел, дочерний по отношению к узлу `div`.

Head First: А как не перепутать элемент с его содержимым?

Узел: Начнем с того, что в дереве DOM содержимое всегда представлено в виде дочерних узлов. Кроме того, каждый узел имеет собственный тип: для элементов это тип `ELEMENT`, в то время как их содержимое относится к типу `TEXT`.

Head First: То есть для доступа к хранящемуся в элементе тексту мне потребуется узел типа `TEXT`?

Узел: Да. Но следует помнить, что свойство `nodeType` возвращает лишь номера типов. Например, типу `TEXT` соответствует номер 3, а типу `ELEMENT` — номер 1. Впрочем, это можно не запоминать, ведь для доступа к содержимому элемента вам достаточно найти дочерний по отношению к нему узел.

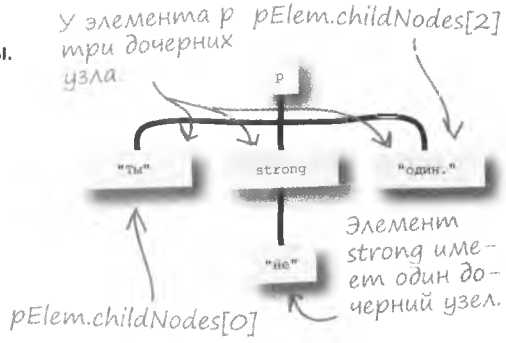
Head First: Понятно. Спасибо, что согласились к нам прийти и рассказать про чудеса дерева DOM.

Узел: Пожалуйста. И если когда-нибудь вы решите заняться приведением в порядок деревьев в вашем саду, не забудьте позвать меня.

Часто Задаваемые Вопросы

В: Я не совсем понимаю, как систематизированы дочерние узлы. Например, как работает свойство `childNodes`?

О: Узел, содержащий данные, считается родительским. Данные, представляющие собой больше, чем просто текстовую строку, формируют набор дочерних узлов. Этот набор указывается в свойстве `childNodes` родительского узла как массив. Порядок элементов массива определяется порядком появления узлов на странице. В результате для получения доступа к первому дочернему узлу достаточно написать `childNodes[0]`. Для поочередного доступа ко всем дочерним элементам используются циклы.

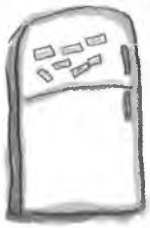


В: Каким образом выполняется проверка условия в цикле `while`, код которого удаляет все дочерние узлы?

О: Вот как выглядит это условие:

```
while(node.firstChild)
```

Мы проверяем, присутствует ли в родительском узле первый дочерний узел. В случае положительного ответа возвращается значение `true`, и цикл продолжает работу. Отрицательный ответ означает, что дочерних узлов больше не осталось. Выражение `node.firstChild` приобретает значение `null`, которое автоматически преобразовывается в `false`. После этого происходит выход из цикла.



Магниты JavaScript

В DOM-совместимой версии «Приключений» отсутствуют несколько важных фрагментов. Расставьте магниты таким образом, чтобы получить код, меняющий содержимое текстового узла. Каждый магнит может быть использован несколько раз.

```
// Обновление описания сцены

var ..... = document.getElementById("     ");

while (     .     )

    .     (     .     );

    .     (document.createTextNode(     ));
```

firstChild

appendChild

scenetext

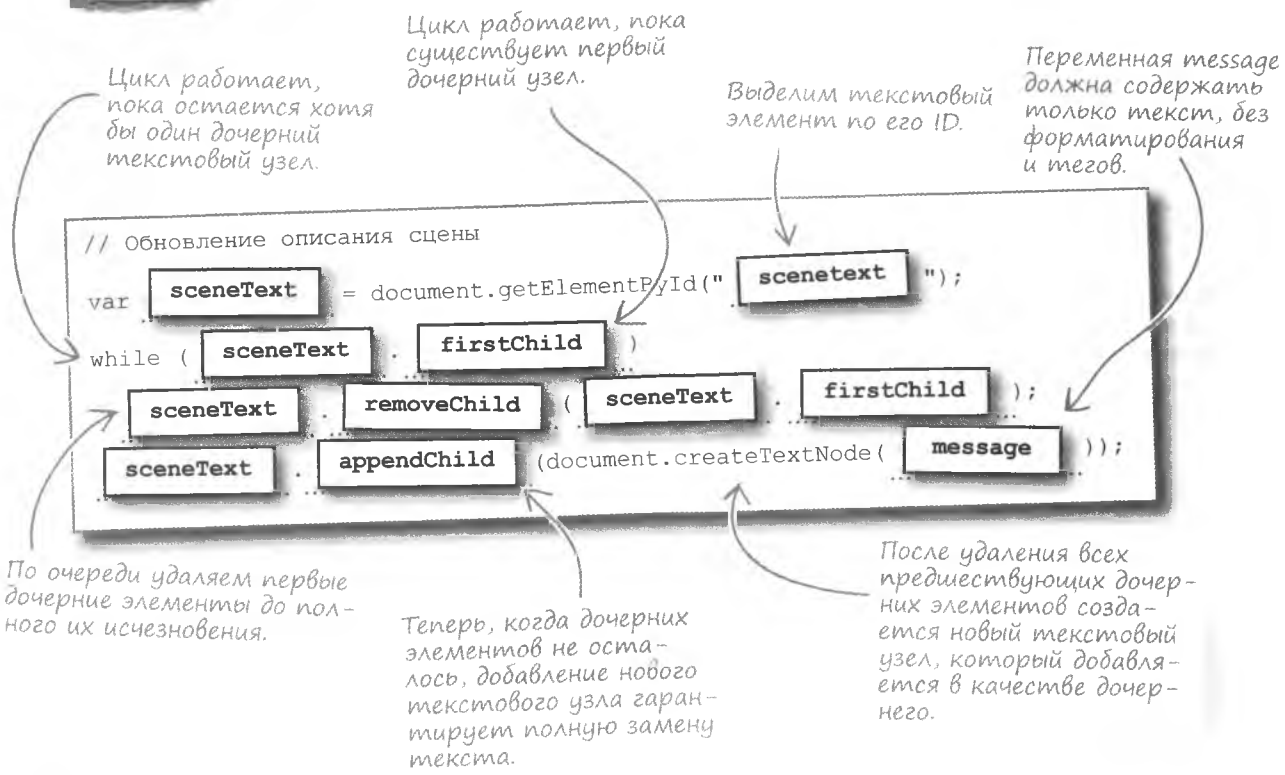
removeChild

sceneText

message

Решение задачи с магнитами

Вот как выглядит код, меняющий содержимое текстового элемента.

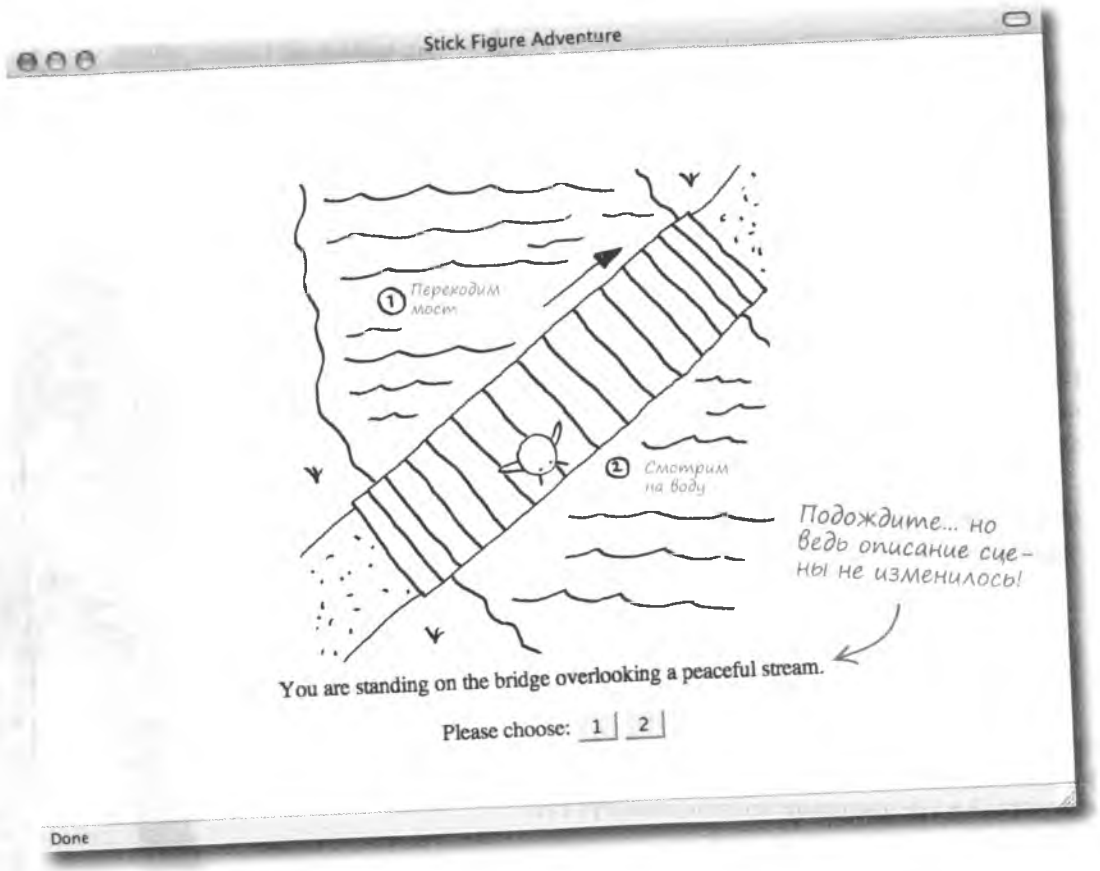


КЛЮЧЕВЫЕ МОМЕНТЫ

- Свойство `innerHTML`, не входящее в стандарт, дает доступ ко всему содержимому элементов.
- Объектная модель документа, или DOM, предоставляет **стандартизованный** механизм доступа к данным и их редактирования.
- В DOM веб-страницы представлены в виде **дерева** связанных узлов.
- В DOM редактирование содержимого страницы включает в себя удаление всех дочерних узлов элемента, с последующим созданием и добавлением дочернего узла с новым содержимым.

«Приключение», совместимое со стандартами

Разве это не здорово! Признаком хорошего «Приключения» стало соответствие стандартам. Вы согласны? В контексте современных веб-приложений это большое преимущество. Важнее всего значительные изменения, которые благодаря DOM произошли с описаниями сцен...



Да, внешний вид страницы не поменялся, но теперь, благодаря DOM, она сконструирована в соответствии с новейшими веб-стандартами. Далеко не всегда код JavaScript можно оценить визуально. Так и в данном случае, изменения коснулись аспектов, находящихся «за кулисами».

Стандарт DOM предоставляет больший контроль над редактированием HTML, чем свойство innerHTML.

В поисках лучших вариантов

К настоящему моменту текстовые описания сцен были дважды переделаны, а вот вид кнопок так и не поменялся. А ведь можно придумать более понятный способ перехода к следующей сцене, нежели выбор между цифрами 1 и 2!

Please choose: 1 | 2

Цифровые кнопки неудачны, так как они не дают представления о том, куда после их нажатия отправится пользователь.

Кнопки с цифрами прекрасно выполняют свою работу, но все-таки стоит их сделать более наглядными.

Нужно снабдить кнопки кратким описанием ситуации, в которую в результате нажатия одной из них попадет пользователь. То есть, если предстоит совершить выбор между «Перейти через мост» и «Остановиться на мосту и посмотреть на воду», кнопки могут выглядеть вот так:

Walk across Bridge | Gaze into Stream

Намного лучше! Теперь кнопки помогают в принятии решения.

Для решения нашей задачи не нужны кнопки формы — с этим справится любой элемент HTML, служащий контейнером для текста. А в оформлении кнопок помогут стили CSS.

МОЗГОВОЙ ШТУРМ

Каким образом реализовать отображение на кнопках текста с кратким описанием следующей сцены?

Проектирование лучше, варианты чище

Так как новые кнопки перехода в «Приключениях» представляют собой HTML-элементы, содержащие текст, для динамического изменения их содержимого воспользуемся средствами DOM. В результате для каждой сцены будет задаваться не только свое описание, но и свой вид кнопок. Значит, функции `changeScene()` потребуются две новые переменные для хранения еще двух текстовых фрагментов. Назовем их `decision1` и `decision2`.

Вот каким образом в функции `changeScene()` будет задан текст для кнопки перехода от сцены 1 к сцене 3:

```
curScene = 3;

message = "You are standing on the bridge overlooking a peaceful stream.";

decision1 = "Walk across bridge";

decision2 = "Gaze into stream";
```

Текст с описанием возможных вариантов перехода хранится в переменных decision1 и decision2.

Возьми в руку карандаш

Динамически меняющиеся надписи на кнопках в нашем «Приключении» требуют нового подхода. Напишите код для новых текстовых элементов, которые заместят кнопки `<input>`.

Подсказка: класс CSS-стиля для новых элементов называется «`decision`». В самой первой сцене на кнопке должен присутствовать текст «`Start Game`» (`Начать игру`).

Please choose: 1 2

```
<input type="button" id="decision1" value="1" onclick="changeScene(1)" />
<input type="button" id="decision2" value="2" onclick="changeScene(2)" />
```

Напишите код для кнопок, текст на которых меняется динамически!

Start Game

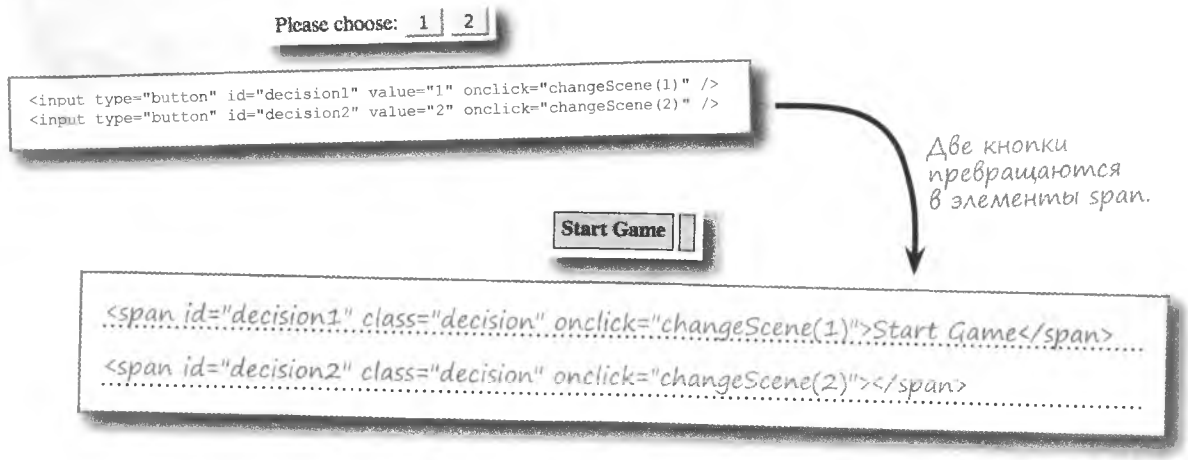
.....

.....

Возьми в руку карандаш

Решение

Итак, вот новый код для текстовых элементов, которыми мы заменим ранее использовавшиеся кнопки `<input>`:



А не создать ли нам функцию для замены текста в узлах?

И снова замена текста в узлах

Для того чтобы кнопки в «Приключениях» стали функционировать нужным образом, не хватает кода, задающего текст в элементах `span`. По большому счету он должен работать аналогично написанному ранее для динамической замены описания сцены коду DOM. Основная проблема в том, что теперь нам каким-то образом нужно решить одну и ту же задачу для трех элементов сразу: для описания сцены и двух кнопок перехода...

Функция, заменяющая текст узла

Функция, реализующая замену текста в узлах, пригодится нам не только для «Приключения». Она должна работать по принципу ранее написанного нами кода замены описания сцены. У этой функции будет два аргумента:

ID узла, текст которого нам нужно поменять.

```
function replaceNodeText(id, newText) {
  ...
}
```

Новый текст.

В качестве аргументов функции `replaceNodeText()` выступает, во-первых, идентификатор узла, содержимое которого следует заменить, во-вторых, новый текст. Функция позволит редактировать любые элементы, являющиеся текстовыми контейнерами. В «Приключениях» она будет динамически менять текст описания сцены и переходных кнопок. Но сначала нам нужно будет написать ее код.

Замена текста с описанием сцены.

Теперь мы вместо трехкратного написания одного и того же кода три раза вызываем функцию.

```
replaceNodeText("scenetext", message);
replaceNodeText("decision1", decision1);
replaceNodeText("decision2", decision2);
```

Замена текста двух кнопок перехода.

Возьми в руку карандаш

Напишите код функции `replaceNodeText()`, которая будет менять текст узла с указанным идентификатором.

Не забудьте, что у этой функции два аргумента: `id` и `newText`.

Возьми в руку карандаш

Решение

Вот код функции `replaceNodeText()`, меняющей текст в узле с указанным ID.

```
function replaceNodeText(id, newText) {
  var node = document.getElementById(id);
  while (node.firstChild)
    node.removeChild(node.firstChild);
  node.appendChild(document.createTextNode(newText));
}
```

Выделяем элемент по его ID.

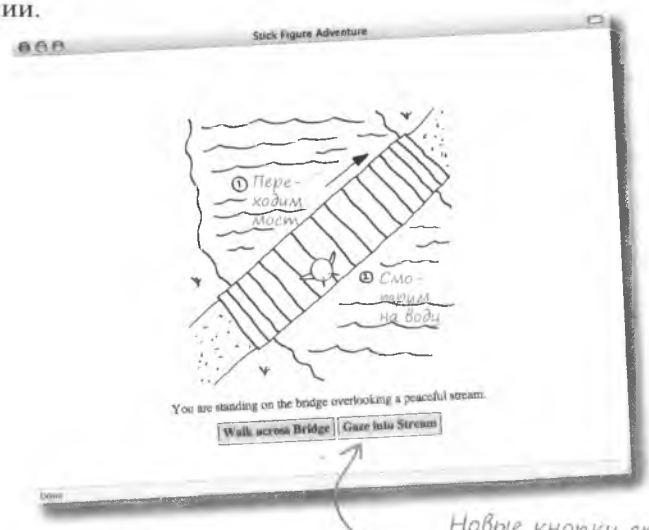
Удаляем все дочерние узлы.

Создаем новый дочерний элемент, используя переданный в качестве аргумента текст.

Функция createTextNode() доступна только в объекте document и не связана с отдельными узлами.

Динамические параметры

Новые варианты кнопок перехода с динамически меняющимся текстом более наглядны, чем кнопки с номерами из предыдущей версии.



Новые кнопки сразу дают понять, что нас ждет в следующей сцене.

Часто Задаваемые Вопросы

В: Почему вместо элементов `div` в «Приключении» используются элементы `span`?

О: Так как кнопки выбора должны располагаться рядом, для их задания не могут использоваться блочные элементы, такие как `div`. Соответственно, мы применяем для них строковой элемент `span`.

В: Где оказывается узел, созданный функцией `createTextNode()`?

О: Нигде. В первый момент созданный текстовый узел оказывается в неизвестности, по крайней мере с точки зрения дерева DOM. Вы вручную должны добавить его к уже существующему узлу в качестве дочернего, и после этого он оказывается в структуре дерева.

В: Созданный при помощи функции `createTextNode()` узел может содержать только текст?

О: Да. При работе со свойством `innerHTML` вы можете назначать текст прямо с тегами. А вот в DOM словосочетание «текстовый узел» означает только текст, без всякого форматирования.

Интерактивность

Новые варианты кнопок перехода намного лучше предыдущих, но даже их можно усовершенствовать. К примеру, можно сделать так, чтобы кнопка подсвечивалась при наведении на нее указателя мыши, как бы подсказывая, что на ней можно щелкнуть.

Подсвечивание элементов связано с CSS, но имеет непосредственное отношение и к DOM.

Подсвечивание содержимого веб-страницы реализуется средствами CSS, так как оно связано с небольшим изменением фонового цвета элемента. При этом DOM обеспечивает программный доступ к CSS-стилям элементов...

Значение стиля: CSS и DOM

CSS-стили связаны с HTML-элементами, в то время как DOM обеспечивает доступ к стилям посредством элементов (узлов). Применение DOM к редактированию CSS-стилей позволяет динамически менять вид страницы. Стили назначаются элементу через класс `style`.

```
<span id="decision1" class="decision" onclick="changeScene(1)">Start Game</span>
<span id="decision2" class="decision" onclick="changeScene(2)"></span>
```

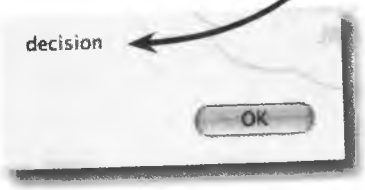
```
<style type="text/css">
  span.decision {
    font-weight:bold;
    border:thin solid #000000;
    padding:5px;
    background-color:#DDDDDD;
  }
</style>
```

Внешний вид кнопок перехода определяет их класс style.

DOM предоставляет доступ к классу стилей элемента при помощи свойства `className` узла.

```
alert(document.getElementById("decision1").className);
```

Свойство className дает доступ к классу стилей элемента.



Свойство className узла предоставляет доступ к классу стилей.

Будьте осторожны!

Не путайте классы стилей CSS с классами JavaScript.

Классы стилей в CSS и классы в JavaScript не имеют между собой ничего общего. В CSS классом называется набор стилей, который может быть применен к элементу страницы, в то время как в JavaScript классом называется шаблон создания объектов. Более подробно классы и объекты будут рассматриваться в главе 10.

Замена классов стилей

Чтобы поменять внешний вид элемента, достаточно указать для него имя другого класса стилей.

Класс стилей decisioninverse меняет цветовую схему кнопки перехода.

```
document.getElementById("decision1").className = "decisioninverse";
```

Вид той же самой кнопки перехода после замены класса стилей!

Новый класс стилей был назначен кнопке перехода при помощи свойства className.


```
<style type="text/css">
  span.decisioninverse {
    font-weight:bold;
    font-color:#FFFFFF;
    border:thin solid #DDDDDD;
    padding:5px;
    background-color:#000000;
  }
</style>
```

Замена класса стилей элемента в свойстве `className` немедленно отражается на внешнем виде элемента. Данная техника позволяет кардинальным образом менять внешний вид элементов страницы, не тратя практически никаких усилий.

Возьми в руку карандаш

Напишите код для элемента `<span>` кнопки перехода, который при помощи событий `onmouseover` и `onmouseout` меняет класс стиля при наведении на кнопку указателя мыши.

Подсказка: класс стиля, который используется при наведенном на кнопку указателе, называется `decisionhover`.

```
<span id="decision1" class="decision" onclick="changeScene(1)"
  .....
  .....>Start Game</span>
<span id="decision2" class="decision" onclick="changeScene(2)"
  .....
  .....></span>
```

Возьми в руку карандаш

Решение

Вот как выглядит код для элемента кнопки перехода, который при помощи событий onmouseover и onmouseout меняет класс стиля при наведении на кнопку указателя мыши.

Класс стиля decisionhover появляется в ответ на событие onmouseover.

Это событие возникает при наведении указателя мыши на элемент span.

```
<span id="decision1" class="decision" onclick="changeScene(1)"
  onmouseover="this.className = 'decisionhover'"
  onmouseout="this.className = 'decision'">Start Game</span>
<span id="decision2" class="decision" onclick="changeScene(2)"
  onmouseover="this.className = 'decisionhover'"
  onmouseout="this.className = 'decision'"></span>
```

Класс стиля unhighlighted возвращается в ответ на событие onmouseout.

Это событие возникает, когда указатель мыши убирается с элемента span.

Часто Задаваемые Вопросы

В: Можно ли создать кнопку, которая подсвечивается при наведении на нее указателя, только средствами CSS?

О: Да. В большинстве случаев именно так и следует поступать, потому что CSS поддерживается бóльшим количеством браузеров, чем JavaScript. Но наши «Приключения» — это приложение, написанное на JavaScript, и многие его функции нельзя реализовать средствами CSS. Соответственно, мы сочли возможным продемонстрировать другой подход к решению данной задачи.

Стильные варианты

Благодаря классам стилей кнопки перехода в наших «Приключениях» теперь могут быть представлены в двух вариантах: обычном и подсвеченном.

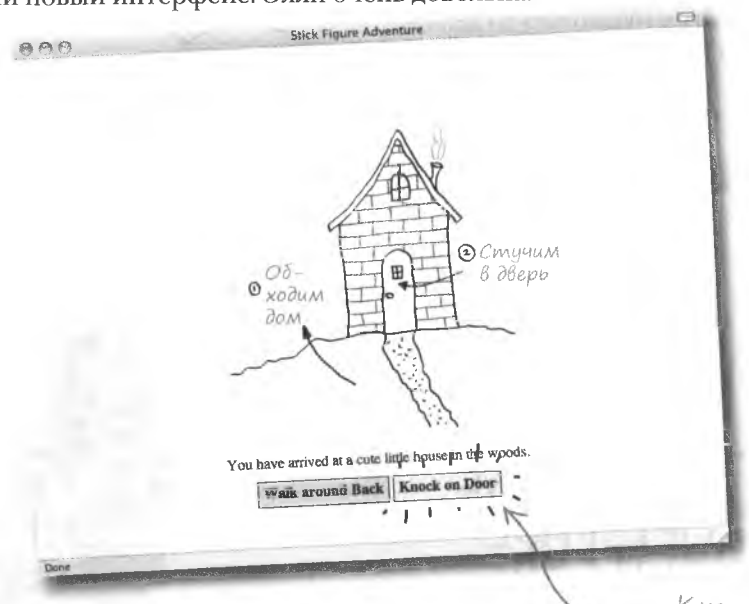
```
<style type="text/css">
  span.decision {
    font-weight:bold;
    border:thin solid #000000;
    padding:5px;
    background-color:#DDDDDD;
  }
</style>
```

Единственным различием между этими классами стилей является цвет фона.

```
<style type="text/css">
  span.decisionhover {
    font-weight:bold;
    border:thin solid #000000;
    padding:5px;
    background-color:#EEEEEE;
  }
</style>
```

Проверка работы приложения

Благодаря возможностям DOM менять класс стиля элемента по требованию наши «Приключения нарисованного человечка» получили новый интерфейс. Элли очень довольна.



Кнопки перехода теперь подсвечиваются при наведении на них указателя мыши.

Часто Задаваемые Вопросы

В: Я не помню событий `onmouseover` и `onmouseout`. Они входят в стандарт?

О: Да, на самом деле есть еще много стандартных событий JavaScript, с которыми вы пока не знакомы. Впрочем, важно только то, как вы реагируете на событие, даже ничего не зная о нем. В данном случае по именам событий можно понять, что первое возникает, когда указатель мыши оказывается **над** элементом, а второе — когда указатель убирается **с** элемента.

В: Почему для задания класса стиля кнопок перехода нам не потребовался метод `getElementById()`?

О: Каждый элемент JavaScript является объектом, а в HTML-коде элемента доступ к объекту осуществляется при помощи ключевого слова `this`. В коде «Приключений» это ключевое слово относится к узлу элемента `span`. Именно этот объект при помощи свойства `className` получает доступ к своему классу стилей. Поэтому для изменения класса стилей достаточно написать `this.className`.

В: Классы стилей — это здорово, но я бы предпочел менять всего одно свойство. Это возможно?

О: У вас замечательная интуиция! Именно над этой проблемой бьется сейчас автор «Приключений» Элли. И эта проблема включает в себя применение JavaScript и DOM для индивидуального управления свойствами стилей...

Пустая knonka

Данная проблема возникла давно, но до этого момента Элли старалась не обращать на нее внимания. Хотя из некоторых сцен возможен только один переход, отображаются все равно обе кнопки, как показано на снимке экрана. С этим нужно что-то делать, потому что интерактивные элементы, не содержащие информации, создают путаницу.

МОЗГОВОЙ ШТУРМ

В каких еще сценах присутствует данная проблема? Как ее можно решить?

Настройка «а ля style»

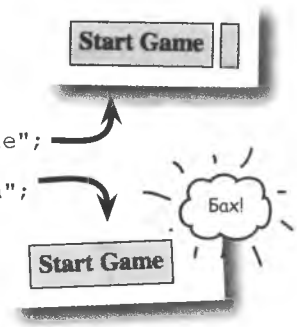
Свойство style узла предоставляет доступ к индивидуальным свойствам стиля.

Иногда не нужно менять весь класс стилей. Для случаев, когда требуется большая детализация, существует объект `style`. Объект `style` доступен как свойство узла и дает доступ к индивидуальным стилям как к свойствам. Так, свойство `visibility` применяется для управления видимостью элементов. В HTML-коде «Приключений» вторую кнопку можно изначально скрыть:

```
<span id="decision2" class="decision" onclick="changeScene(2)"
  onmouseover="this.className = 'decisionhover'"
  onmouseout="this.className = 'decision'"
  style="visibility:hidden"></span>
```

Таким образом, для изменения видимости элемента достаточно менять значение свойства `visibility` с `visible` на `hidden` и обратно.

```
document.getElementById("decision2").style.visibility = "visible";
document.getElementById("decision2").style.visibility = "hidden";
```



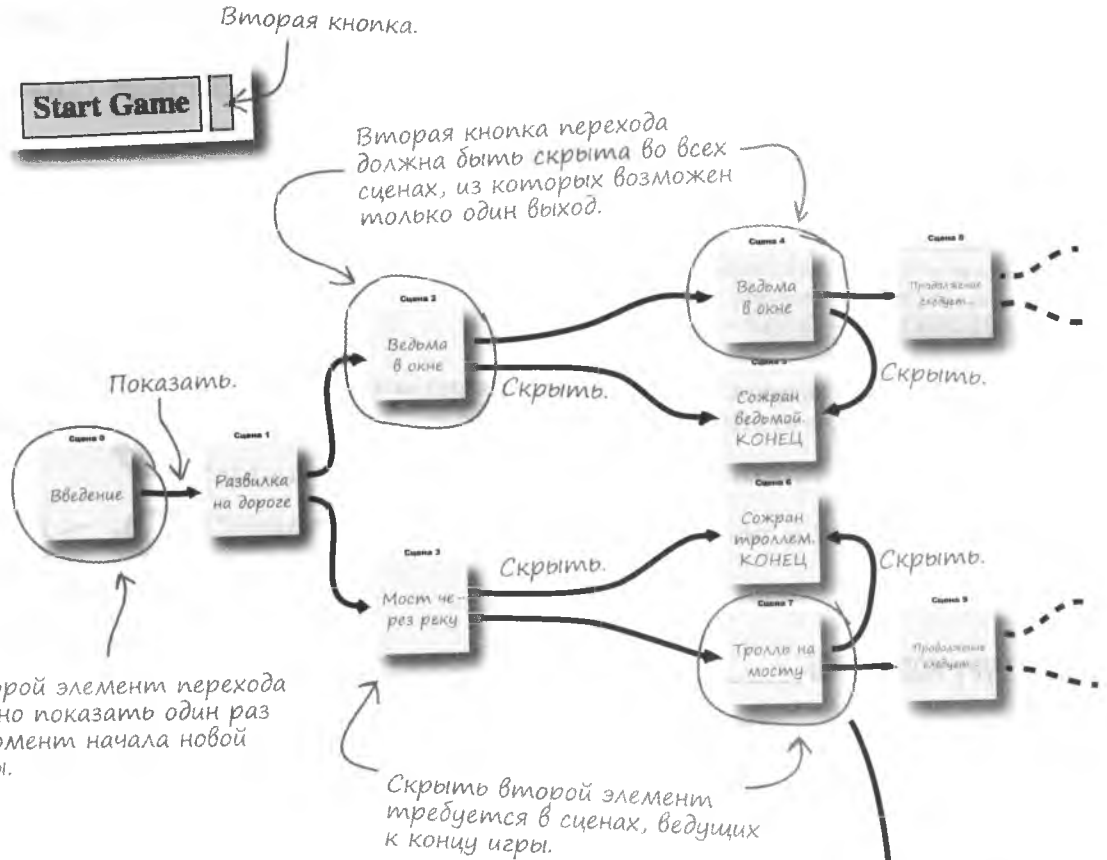
Возьми в руку карандаш

В некоторых сценах «Приключений» должна меняться видимость второй кнопки перехода. Пометьте эти сцены и укажите, каким образом должно осуществляться это изменение.

Возьми в руку карандаш

Решение

Вот каким образом в некоторых сценах «Приключений» должна меняться видимость второй кнопки перехода.



В каждой сцене изменение видимости второй кнопки перехода осуществляется при помощи свойства visibility объекта style.

```
...
case 7:
  if (decision == 1) {
    curScene = 6
    message = "Sorry, you became the troll's tasty lunch.";
    decision1 = "Start Over";
    decision2 = "";

    // Скрыть вторую кнопку перехода
    document.getElementById("decision2").style.visibility = "hidden";

  }
  else {
    curScene = 9;
    decision1 = "?";
    decision2 = "?";
  }
  break;
...
```

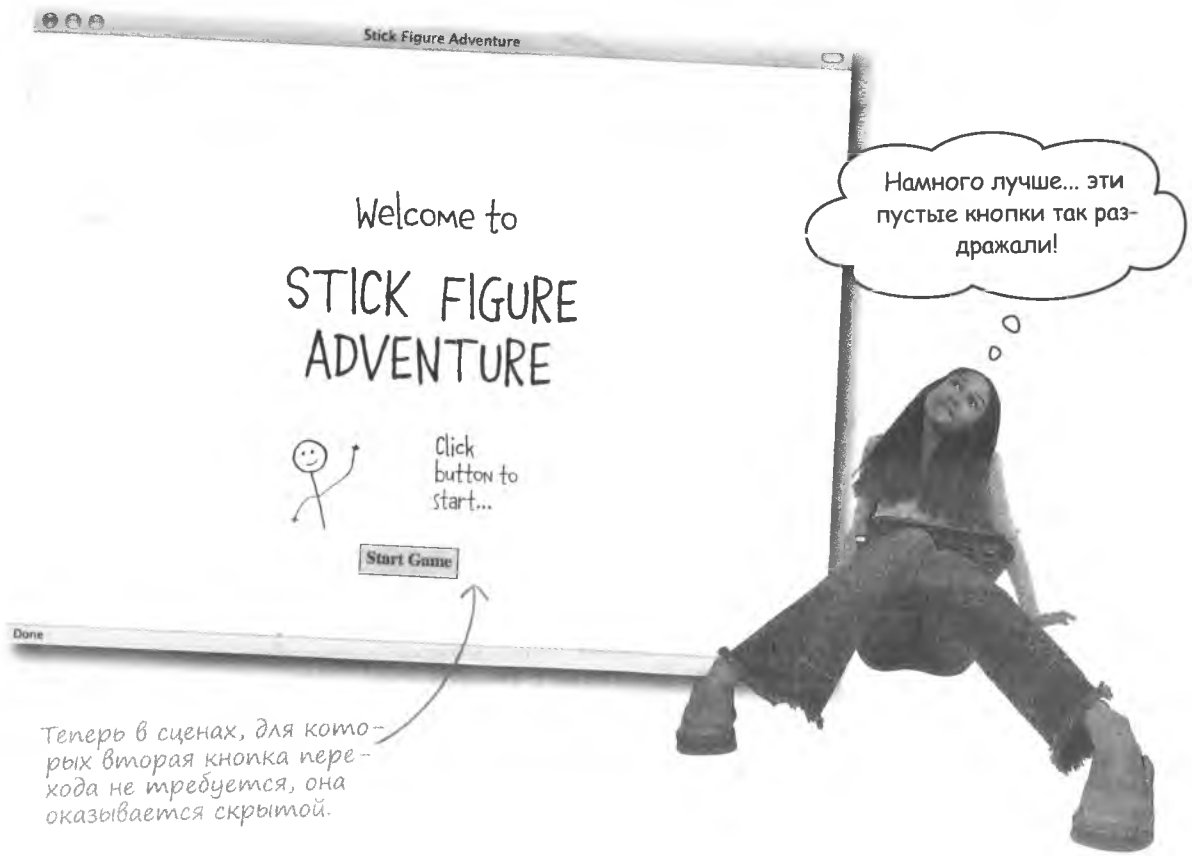
КЛЮЧЕВЫЕ МОМЕНТЫ

- Свойство `className` узла меняет **весь класс стиля** этого узла.
- Свойство `style` дает доступ к **отдельным свойствам стиля** узла.
- Классы стилей CSS не имеют ничего общего с классами JavaScript — это совсем разные вещи.
- Элементы страницы можно динамически прятать или снова делать видимыми при помощи свойства `visibility` элемента.

Аналогично действует свойство стиля display. Присвоив ему значение none, вы скроете элемент, а присвоив значение block, наоборот, сделаете видимым.

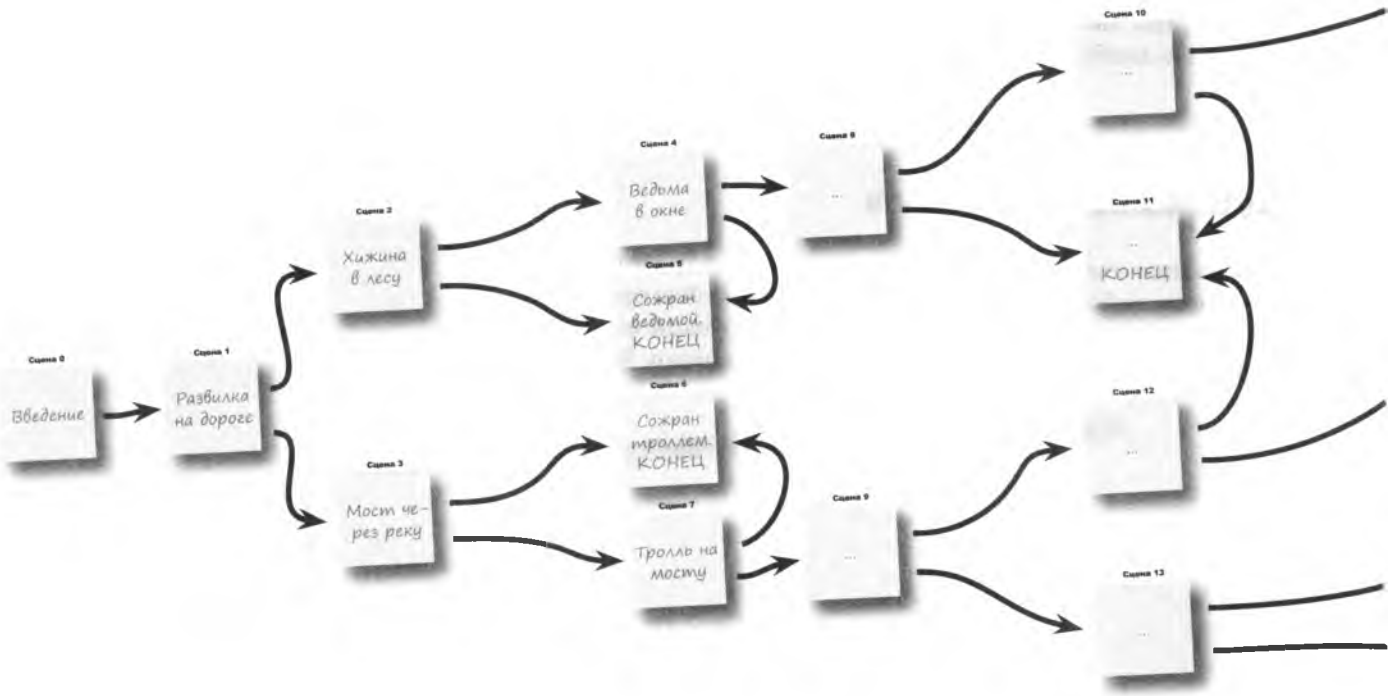
Без фиктивных кнопок

Управление отдельными стилями при помощи DOM позволяет по вашему выбору скрывать или делать видимой вторую кнопку перехода. В результате вы получаете более функциональный пользовательский интерфейс, ведь лишних кнопок больше нет.



Усложняем «Приключения»

Поскольку Элли собиралась написать длинную и интересную историю со множеством ветвлений, в управлении столь сложной структурой ей поможет DOM.



Новые «Приключения» = Рост дерева решений!

Самая последняя версия «Приключений нарисованного человечка» уже выложена в Интернете и ждет ваших дополнений. Если вы еще не скачали ее, сделайте это по адресу http://www.headfirstlabs.com/books/hfjs/.

Вряд ли вы напишете длинную историю без хорошего способа проверки условий.

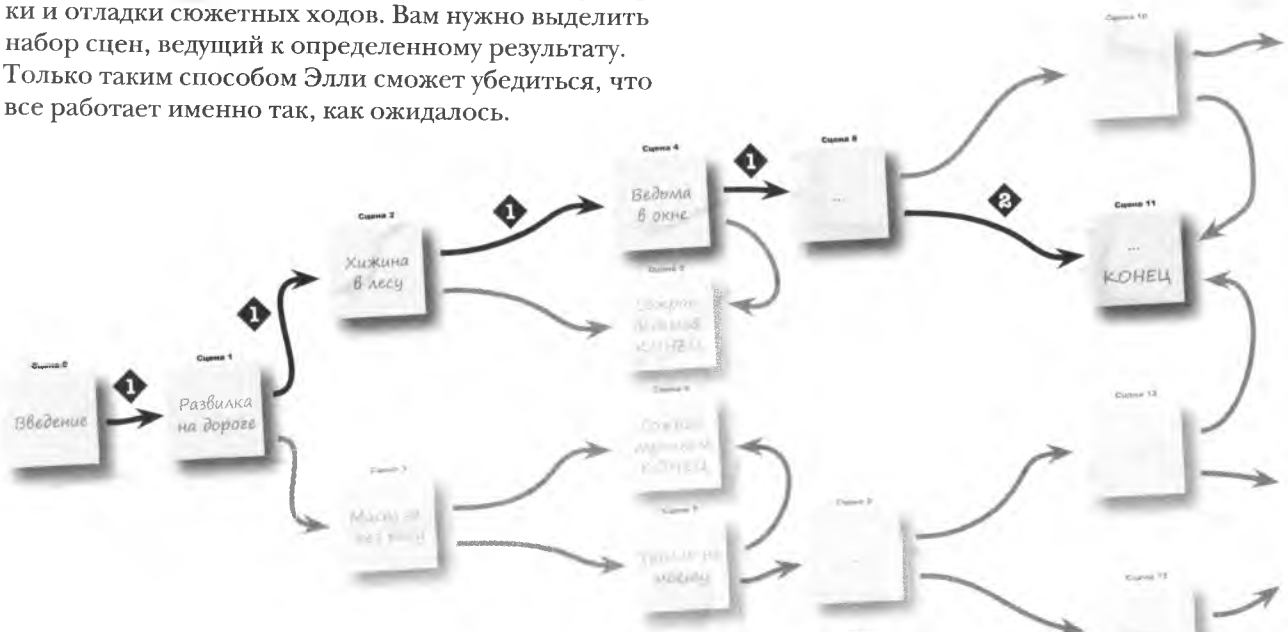
По мере добавления новых сцен и переходов становится все сложнее обычным способом проверять условия и гарантировать, что на каждой развилке пользователь будет попадать туда, куда нужно. Поэтому нам определенно требуется новый механизм проверки условий.

МОЗГОВОЙ ШТУРМ

Каким образом лучше всего осуществлять проверку условий при таком огромном количестве ветвлений?

Поход по дереву решений

Браузеры умеют запоминать список посещенных пользователем страниц. Аналогичную функцию можно использовать в «Приключениях» для проверки и отладки сюжетных ходов. Вам нужно выделить набор сцен, ведущий к определенному результату. Только таким способом Элли сможет убедиться, что все работает именно так, как ожидалось.



История переходов конструируется как список посещенных по мере продвижения по игре сцен. Именно она потом будет использоваться в качестве отладчика.

Начало Сцена 0 — Введение

Выбирается первый вариант...

1 Сцена 1 — Развилка на дороге.

1 Сцена 2 — Хижина в лесу.

1 Сцена 3 — Ведьма в окне. ...которая ведет к сцене 4.

1 Сцена 8 — ...

2 Сцена 11 — ...

Каждая пройденная сцена добавляется к истории вместе с решением, которое к ней привело.

Конец

МОЗГОВОЙ ШТУРМ

Каким образом следует изменить страницу «Приключений» для включения поддержки «истории пройденных сцен»?

Превратим историю в HTML

С точки зрения HTML код истории решений не является слишком сложным: вам понадобится элемент `div` и строка текста, описывающая следующую сцену.

Каждый элемент p содержит один из пунктов истории переходов.

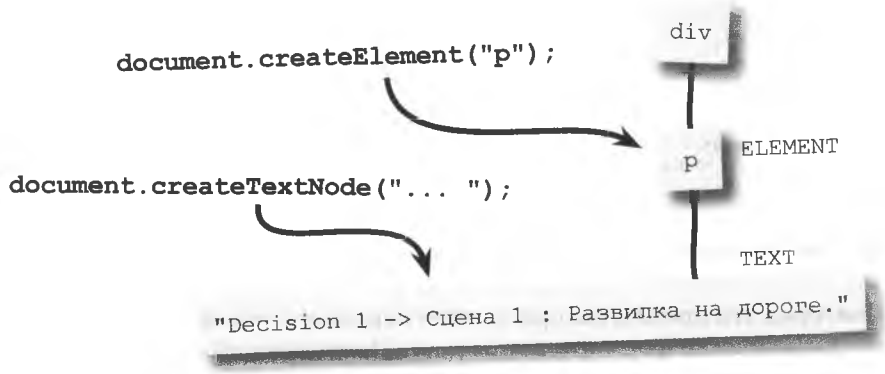
```
<div id="history">
  <p>Decision 1 -> Сцена 1 : Развилка на дороге.</p>
  <p>Decision 1 -> Сцена 2 : Хижина в лесу.</p>
  <p>Decision 1 -> Сцена 3 : Ведьма в окне.</p>
  ...
</div>
```

Осталось написать код JavaScript, который при помощи DOM превратит историю посещений в набор узлов.

Функция фиксации истории посещенных страниц может быть полезным инструментом отладки.

DOM позволяет создать любой HTML-элемент.

А значит, вы можете создать и строку. Для этого вам потребуется еще один метод объекта документа `createElement()`. Именно он создает новый контейнер, в который вы добавляете текст, формируя дочерний узел при помощи функции `createTextNode()`. В результате на дереве появляется новая ветка узлов.



Обработка HTML-кода

Для создания нового элемента методом `createElement()` достаточно указать имя тега. Так, чтобы создать элемент (`p`), достаточно вызвать метод с аргументом `"p"` и указать, куда присоединить конечный результат.

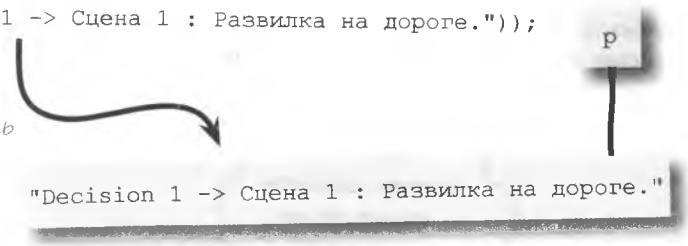
Мы начнем с нового элемента p, «парящего в пространстве».

```
var decisionElem = document.createElement("p");
```

Итак, у нас есть новый элемент без содержимого, пока еще не принадлежащий никакой странице. То есть нам нужно добавить к нему текст, создать текстовый узел и добавить его в качестве дочернего к узлу `p`.

```
decisionElem.appendChild("Decision 1 -> Сцена 1 : Развилка на дороге."));
```

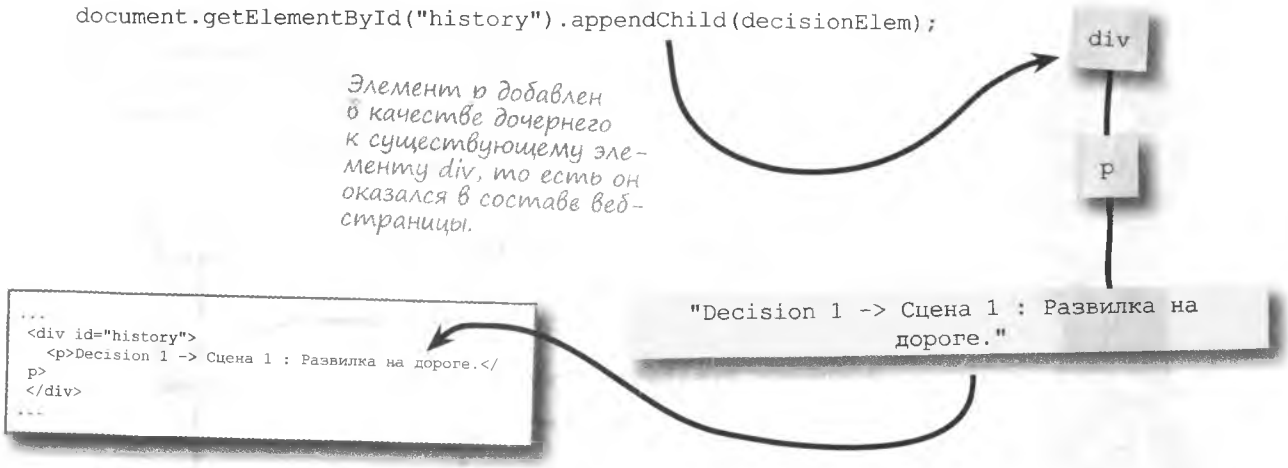
Элемент p до сих пор никуда не присоединен, но он перестал быть пустым благодаря новому дочернему текстовому узлу



Теперь новый элемент нужно добавить в качестве дочернего к элементу `div` истории переходов.

```
document.getElementById("history").appendChild(decisionElem);
```

Элемент p добавлен в качестве дочернего к существующему элементу div, то есть он оказался в составе веб-страницы.



Повторяя эти процедуры при каждом переходе к новой сцене, мы динамически формируем историю «Приключений».

КЛЮЧЕВЫЕ МОМЕНТЫ

- Все HTML-элементы можно создать методом `createElement()` объекта `document`.
- Для добавления текстового содержимого нужно создать текстовый узел и присоединить его к элементу.
- Аккуратно добавляя и удаляя узлы на дереве DOM, можно по желанию разбирать страницу на составные части и собирать обратно.

Возьми в руку карандаш

Напишите функцию `changeScene()`, фиксирующую историю пройденных сцен в «Приключениях». Подсказка: когда текущая сцена отлична от нулевой, нужно добавить новый параграф с дочерним текстовым узлом к элементу с историей переходов. Для сцены номер 0 нужно удалить историю.

```
function changeScene(decision) {

  ...

  // Обновление истории пройденных сцен
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

Возьми в руку карандаш

Решение

Вот код функции `changeScene()`, фиксирующей историю пройденных сцен в «Приключениях».

Добавим новый текстовый узел к новому параграфу.

Функция changeScene() уже снабжена локальной переменной decision, поэтому данной переменной присваивается другое имя.

Выделим связанные с историей элементы по идентификатору.

```
function changeScene(decision) {
  ...
  // Обновление истории пройденных сцен.
  var history = document.getElementById("history");
  if (curScene != 0) {
    // Добавим к истории последний переход
    var decisionElem = document.createElement("p");
    decisionElem.appendChild(document.createTextNode("decision " + decision +
      " -> Scene " + curScene + " : " + message));
    history.appendChild(decisionElem);
  }
  else {
    // Удалим историю переходов
    while (history.firstChild)
      history.removeChild(history.firstChild);
  }
}
```

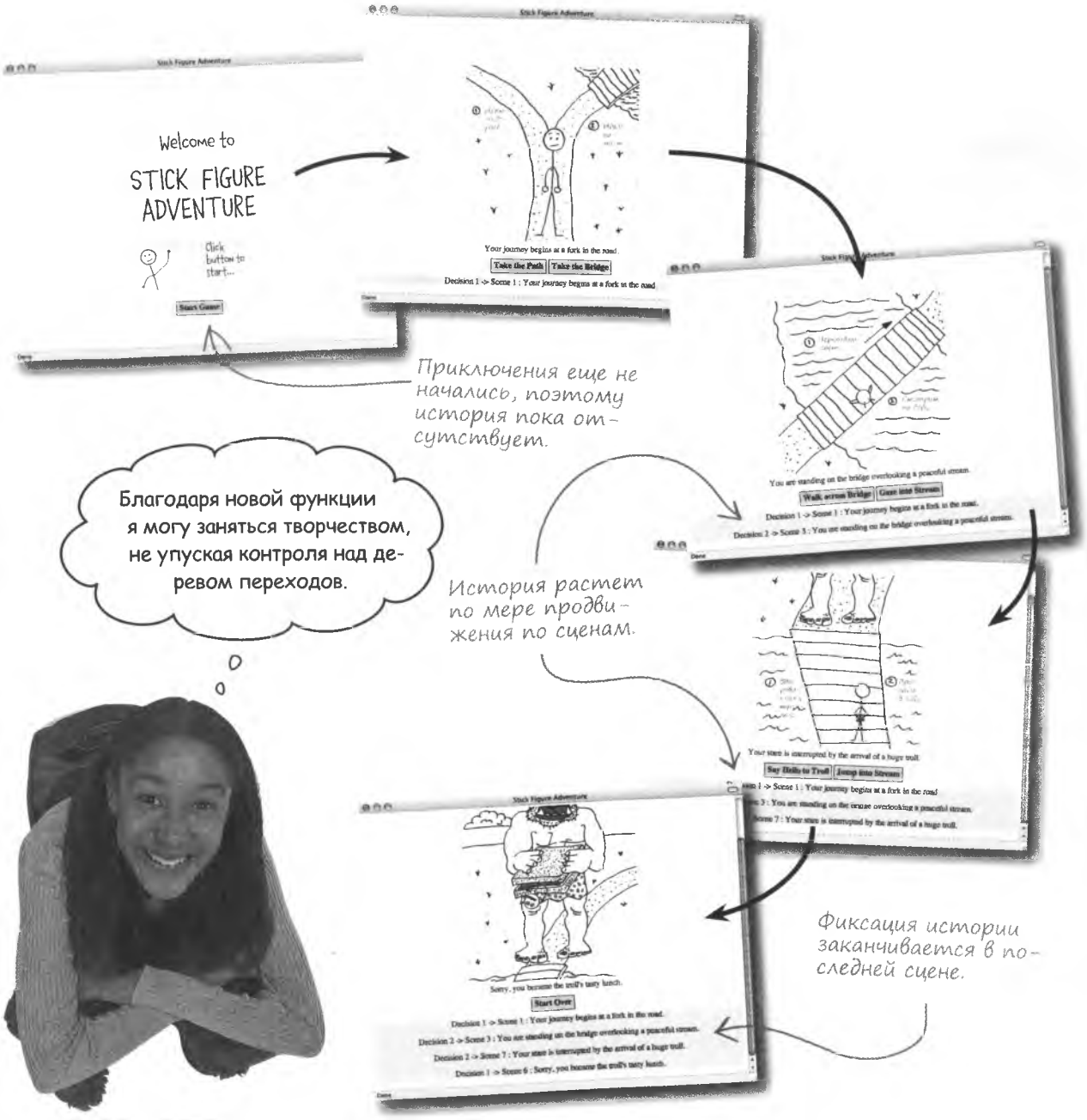
Добавим новый параграф к элементу div.

Очистим историю, удалив все дочерние узлы.

Создадим новый текстовый узел с информацией о переходе к очередной сцене.

Отслеживание «Приключений»

Функция фиксации истории в «Приключениях нарисованного человечка» дает возможность аккуратно отследить логику происходящего.



Длинное, странное путешествие...

Что ж, пришло время напрячь вашу фантазию и превратить «Приключения нарисованного человечка» в настоящую длинную игру, на примере которой можно будет осуществить отладку функции, фиксирующей историю. Ваш нарисованный друг ждет приключений...

Упражнение

Придумайте ваше собственное продолжение «Приключений нарисованного человечка» и добавьте соответствующий код.

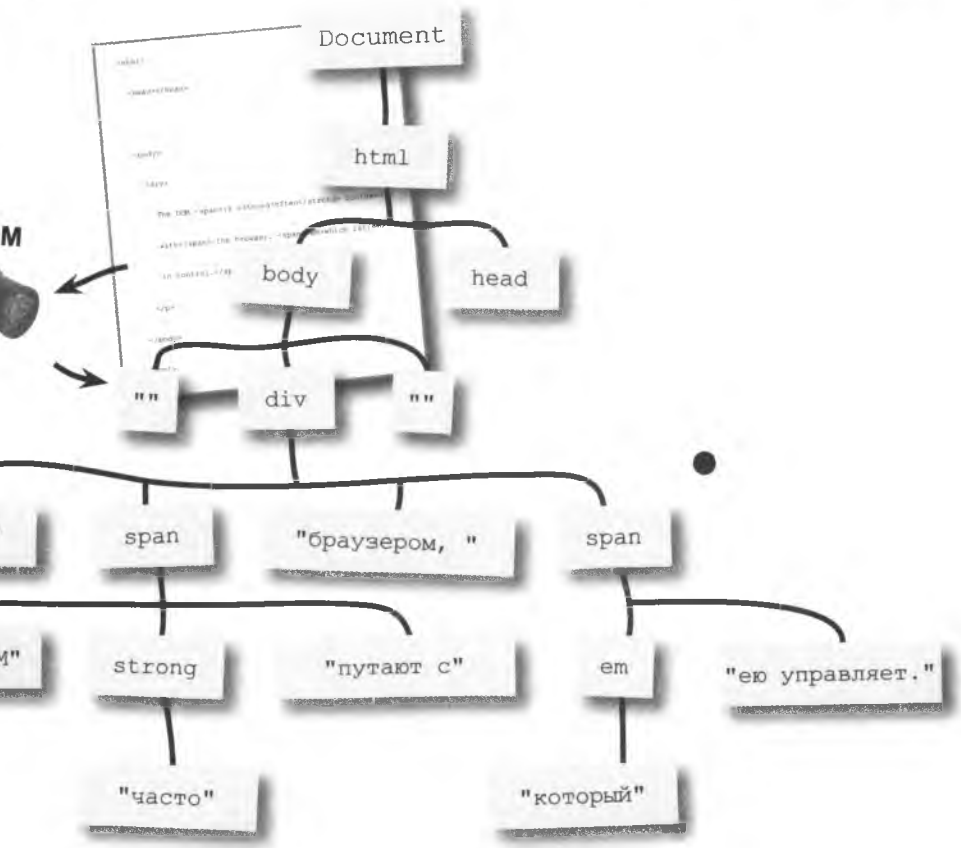
Для этого упражнения не существует готового решения... Так что получайте удовольствие от самостоятельных приключений!

Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Так что же это такое, DOM?

Ум хорошо, а два лучше!



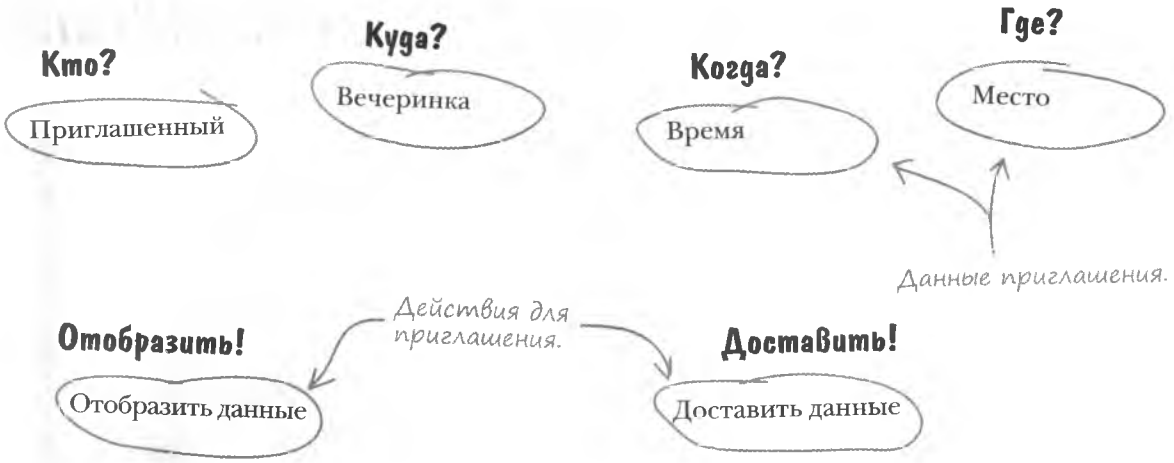
Программистам JavaScript не следует сильно увлекаться DOM. Эта функция дает доступ к HTML-тегам, но старайтесь не слишком ею злоупотреблять.

Объекты как франкенданные

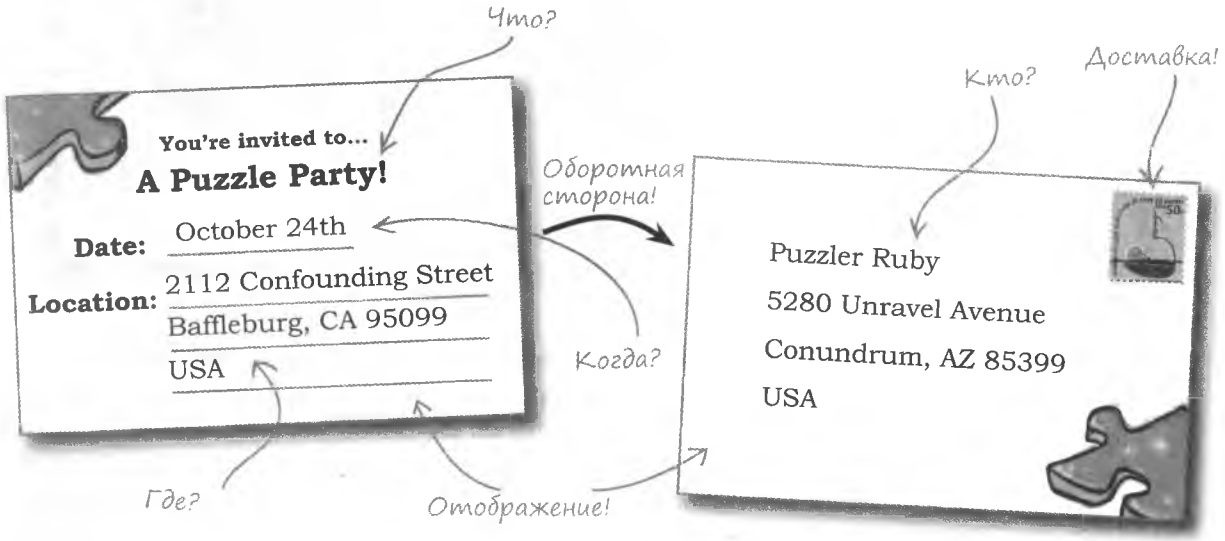
Объекты JavaScript вовсе не так ужасны, как заставил вас думать доктор. Зато они интересны тем, что соединяют друг с другом отдельные части языка JavaScript, делая его более мощным. **Объекты объединяют данные с действиями** в новый **тип**, намного более «живой», чем все, что вы использовали раньше. Вы познакомитесь с **массивами, которые сортируют себя сами**, со строками, которые умеют искать в своем составе указанные последовательности символов, и многими другими замечательными особенностями.

Вечеринка в стиле JavaScript

Давайте устроим вечеринку. Вам нужно пригласить на нее гостей. Какая информация вам для этого требуется?



В JavaScript смоделировать приглашения на вечеринку можно, представив данные в виде переменных, а действия в виде функций. При этом мы сталкиваемся с проблемой, которой не существует в реальном мире, — как разделить данные и действия.

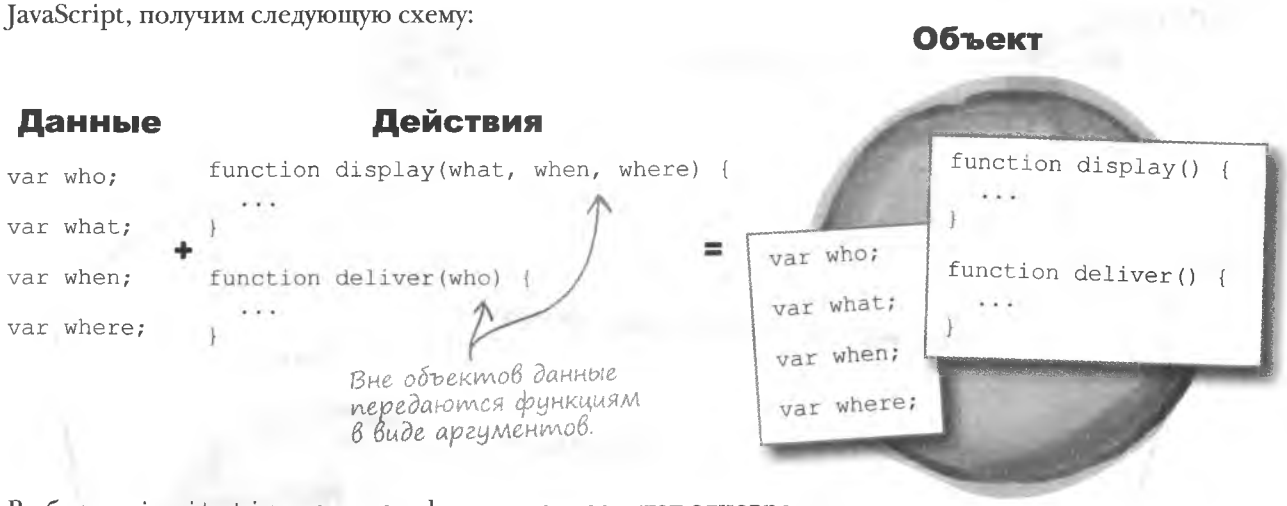


Реальные приглашения совмещают данные и действия в одной сущности, называемой объектом.

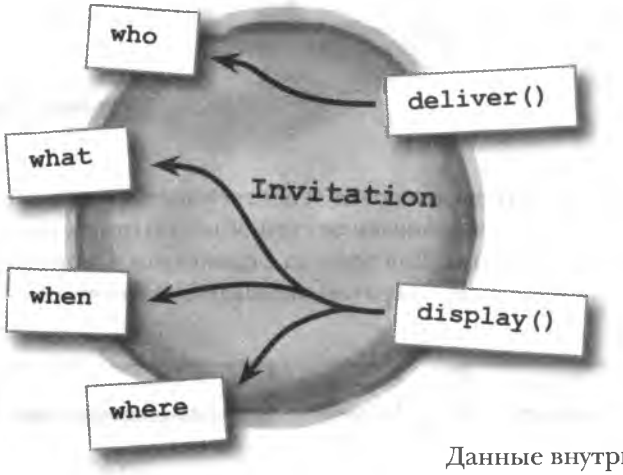
Данные + действия = объект

В JavaScript вовсе не обязательно всегда соблюдать раздельный режим работы с данными и действиями. Более того, существует такая уникальная структурная единица, как **объект**, одновременно **хранящая** данные и выполняющая над ними различные **действия**.

Рассмотрев приглашения на вечеринку с точки зрения объектов JavaScript, получим следующую схему:



В объекте `invitation` данные и функции существуют одновременно и близко связаны друг с другом. Функциям, помещенным внутрь объекта, уже не нужно передавать данные в виде аргументов, так как эти функции имеют к данным прямой доступ.

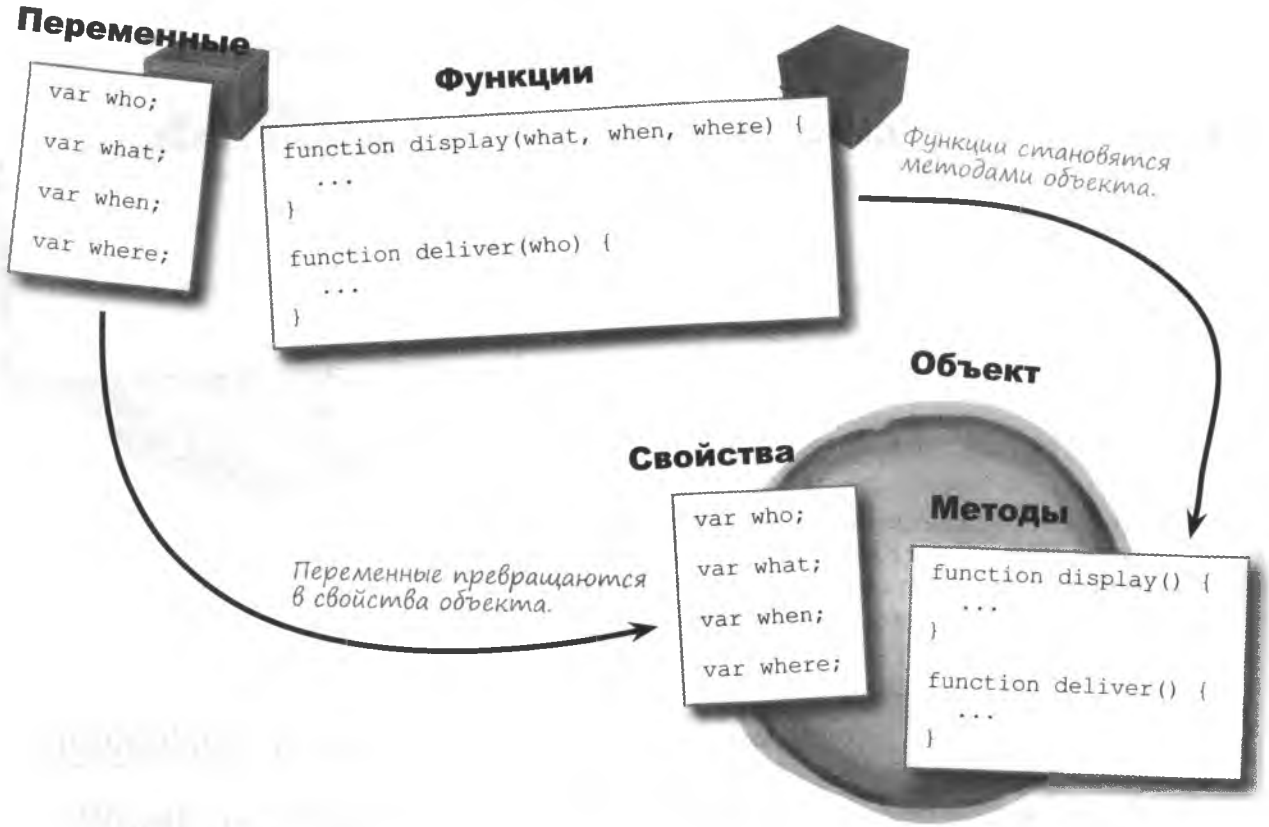


Объекты связывают данные и функции в пределах одного контейнера.

Данные внутри объекта доступны для функции, но скрыты от внешнего мира. Поэтому объект можно рассматривать как контейнер, хранящий данные и связывающий их с обрабатывающим кодом.

Данные — это собственность объекта

Помещенные в объект данные и действия принято называть **членами объекта**. Переменные также называются **свойствами**, а функции **методами**. Эти структурные единицы по-прежнему выполняют свои функции, то есть хранят данные и выполняют над ними различные операции, но теперь они делают это в контексте определенного объекта.



Свойства и методы объекта являются эквивалентами переменных и функций.

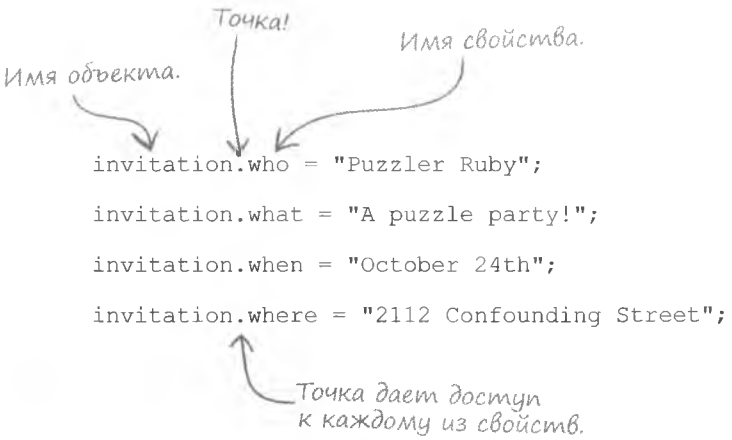
Свойства и методы являются «собственностью» объекта. Это означает, что они хранятся внутри объекта, примерно как данные внутри массива. Для доступа к свойствам и методам используется специальный оператор, называемый **«точкой»**.

Ссылка на члены объекта

Оператор «точка» устанавливает связь между свойством или методом и объектом, к которому они принадлежат. Примерно так же, как имя дает представление о самом человеке, а фамилия, о том, из какой он семьи.

Оператор «точка» дает ссылку на свойства или методы объекта.

Вооруженные этими знаниями, соединим данные для объекта `invitation`. Для этого нам понадобится перечень его свойств и оператор «точка»:



Помните, что, так как данные и действия являются частью одного объекта, методам не нужно передавать какие бы то ни было параметры. В результате любые действия с объектом `invitation` выполняются очень просто:

```
invitation.deliver();
```

Имя объекта.

Имя метода.

Упражнение

Нашим приглашениям не хватает свойства, позволяющего ответить, придет гость на вечеринку или нет. Напишите код, добавляющий объекту свойство `rsvp`, а затем вызовите метод `sendRSVP()`, ответственный за отправление ответов.

.....

.....

Упражнение Решение

Вот как выглядит код свойства `rsvp`, ответственного за отправку ответов на приглашения.

```
invitation.rsvp = "attending";
invitation.sendRSVP();
```

Оператор "точка" дает ссылку на свойство и метод объекта invitation.

Сюда можно поставить также булевское свойство. В этом случае значение true будет означать принятое приглашение, а значение false — отклоненное.

Часто Задаваемые Вопросы

В: Что же такое «объект»? Принадлежит ли он к какому-нибудь типу данных?

О: Объект представляет собой именованный набор свойств и методов. Собственно говоря, объект — **это и есть** тип данных. Уже знакомые вам типы включают в себя числа, тексты и логические значения. Они называются *примитивными*. Объекты же считаются *составными* типами данных, так как состоят из набора фрагментов информации. Соответственно, вы можете добавить к уже известным вам трем типам данных еще и тип «object». Именно этому типу будут принадлежать созданные вами или встроенные объекты JavaScript.

В: Нельзя ли вместо свойств и методов объекта просто использовать глобальные переменные и функции?

О: Можно. Но проблема в том, что доступ к глобальным переменным возможен из любой части кода. А как мы уже упоминали, лучше оставлять доступ только тем фрагментам сценария, которым он действительно нужен. Это позволяет предотвратить случайное редактирование данных.

К сожалению, в настоящий момент JavaScript не позволяет полностью защитить свойства объекта от доступа извне. Кроме того, иногда возникают ситуации, когда к ним требуется непосредственный доступ. Впрочем, основная идея состоит в том, что помещенные в объект данные логически связываются с ним. А связанный с объектом фрагмент информации является более контекстным, чем свободно расположенный где-то в сценарии (глобальная переменная).

В: Я уже несколько раз видел запись с оператором «точка». Неужели я все это время пользовался объектами?

О: Да. Скоро вы поймете, что сложно писать сценарии на JavaScript, не прибегая к объектам. Ведь этот язык — сам по себе лишь один большой набор объектов. Например, функция `alert()` по сути является методом объекта `window`, то есть для ее вызова можно написать `window.alert()`. Объект `window` представляет собой окно браузера и в явном виде ссылаться на него необязательно. Собственно, поэтому вы можете работать только с функцией `alert()`.

В: Я совсем запутался. Вы пытаетесь сказать мне, что функции это на самом деле методы?

О: Да, хотя думать о функциях подобным образом вам пока непривычно. Как вы знаете, функция — это фрагмент кода, который можно вызвать по имени. Метод представляет собой функцию, помещенную внутрь объекта. Другое дело, что с объектами связаны все функции.

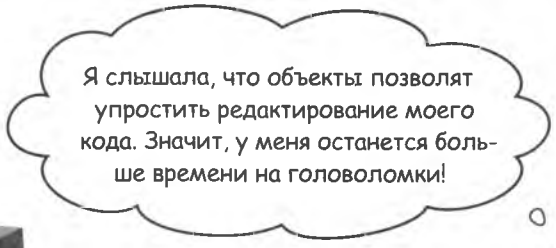
Например, `alert()` — это одновременно функция и метод, что объясняет возможность вызвать ее двумя способами. В данном случае, как и в ряде других, в качестве объекта выступает окно браузера. Так как этот объект по умолчанию используется при вызове методов, если в явном виде не указано иное, можно считать, что метод `alert()` является функцией.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Объектами называют особые **структуры**, объединяющие данные и **обрабатывающий их код**.
- С практической точки зрения объект — это всего лишь переменные и функции, объединенные в **структурную единицу**.
- Помещенные в объект переменные превращаются в **свойства**, а функции — в **методы**.
- Для ссылки на свойство или метод достаточно указать имя **объекта**, затем — **точку**, а затем — **имя** свойства или метода.

Блог для любителей головоломок

Инициатором вечеринки, для которой мы создаем приглашения, является Руби — фанатка головоломки «Кубик Рубика». Она мечтает о встрече с людьми, разделяющими ее страсть. При этом Руби хотела бы не только устроить вечеринку для единомышленников. Она решила завести блог YouCube, посвященный головоломкам.

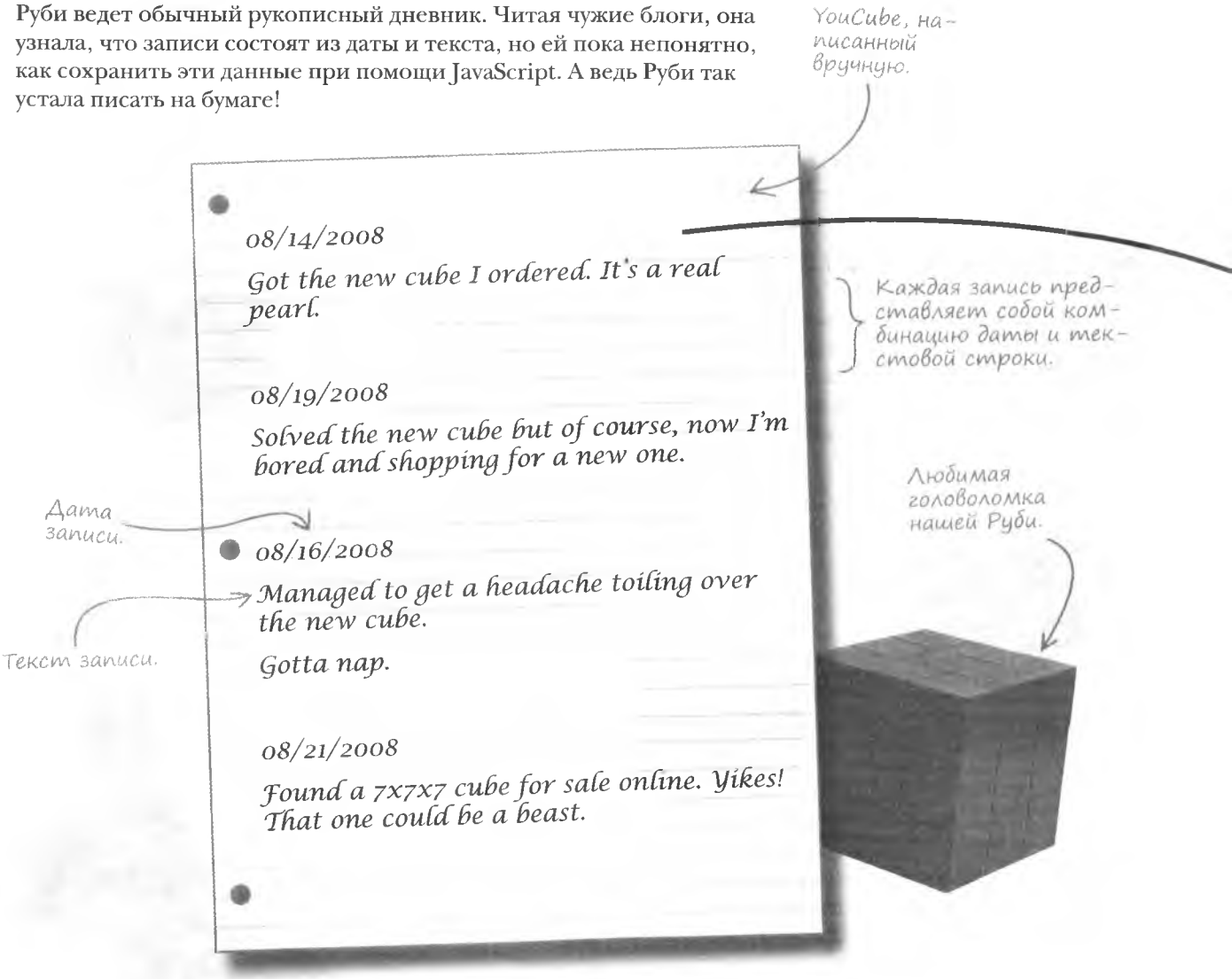


Руби слышала, что в JavaScript поддерживаются специальные объекты, так как это позволяет создавать более устойчивый и управляемый код. Еще она слышала, что многие блоги устаревают, потому что пользователи прекращают ими заниматься. Поэтому Руби решила с самого начала создать **объектно-ориентированный** сценарий блога YouCube, который обеспечит ее прекрасное будущее.

Объектно-ориентированный YouCube = **Больше времени на головоломки!**

Анализ блога YouCube

Руби ведет обычный рукописный дневник. Читая чужие блоги, она узнала, что записи состоят из даты и текста, но ей пока непонятно, как сохранить эти данные при помощи JavaScript. А ведь Руби так устала писать на бумаге!



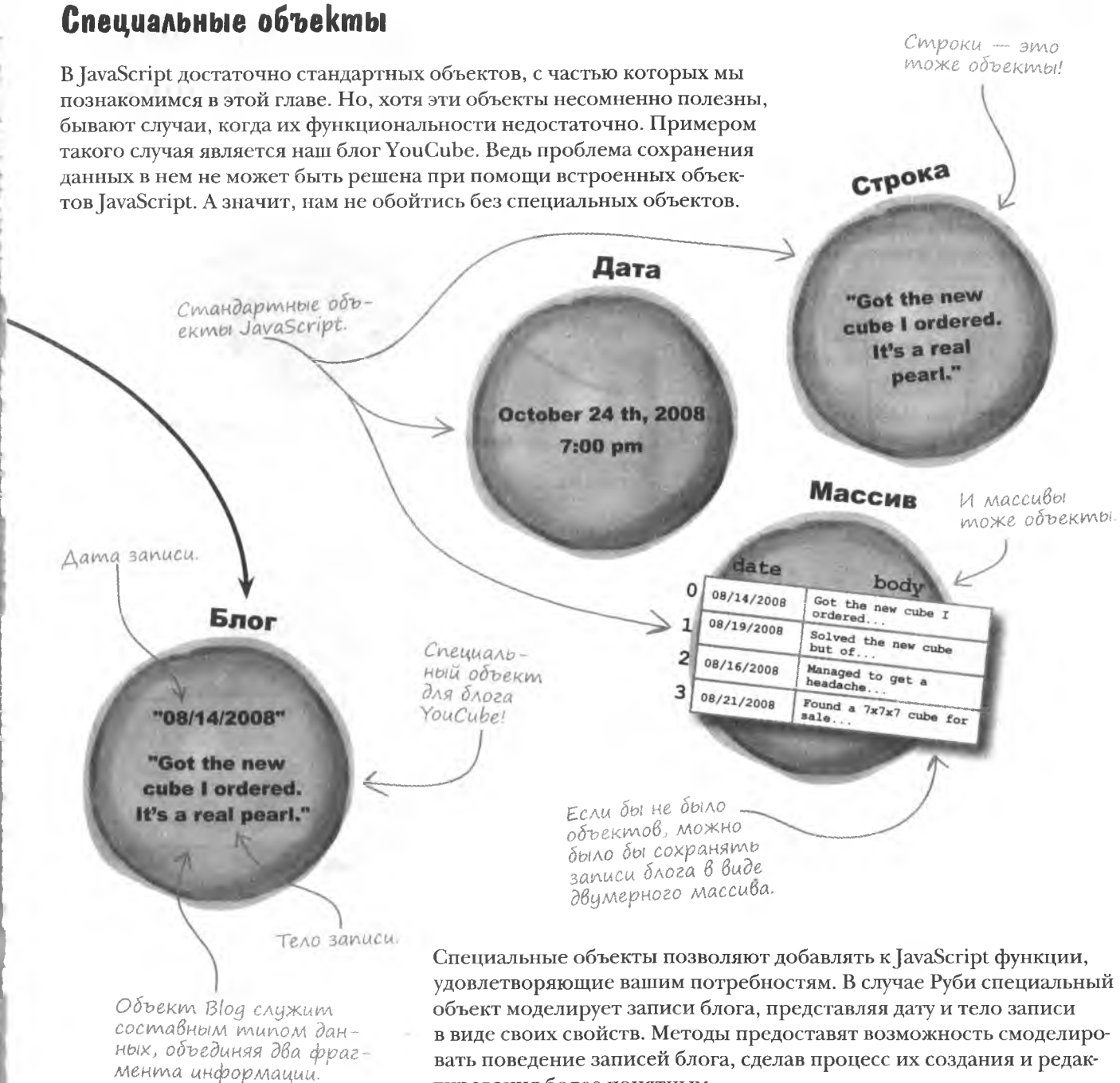
Руби отчаянно нужен непосредственный механизм сохранения набора данных в форме (дата + текст) с последующим доступом к нему. Вы уже слышали подобные слова, когда мы говорили об объектах JavaScript, ведь именно объекты позволяют соединить различные фрагменты информации в целостную структуру.

Дата записи + Тело записи = Объект Blog

Специальный объект объединяет в общую структуру два фрагмента блога.

Специальные объекты

В JavaScript достаточно стандартных объектов, с частью которых мы познакомимся в этой главе. Но, хотя эти объекты несомненно полезны, бывают случаи, когда их функциональности недостаточно. Примером такого случая является наш блог YouCube. Ведь проблема сохранения данных в нем не может быть решена при помощи встроенных объектов JavaScript. А значит, нам не обойтись без специальных объектов.



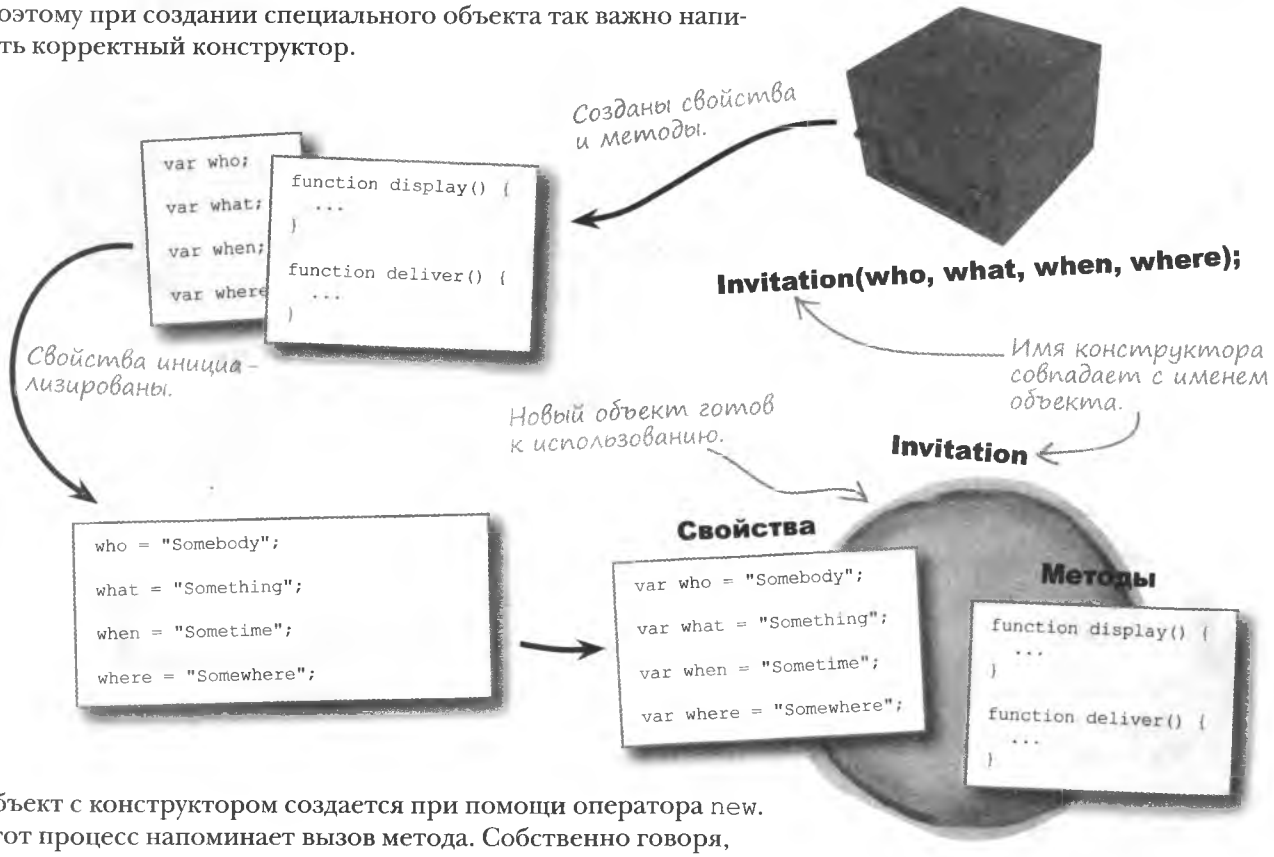
Специальные объекты позволяют добавлять к JavaScript функции, удовлетворяющие вашим потребностям. В случае Руби специальный объект моделирует записи блога, представляя дату и тело записи в виде своих свойств. Методы предоставят возможность смоделировать поведение записей блога, сделав процесс их создания и редактирования более понятным.

Но перед тем, как воспользоваться таким специальным объектом, нужно понять механизм его создания...

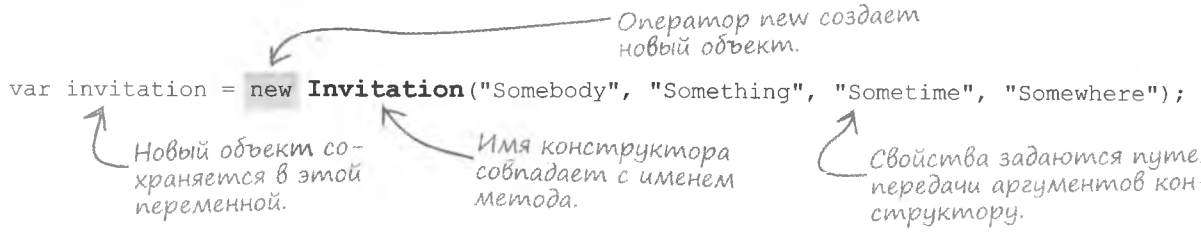
Конструктор

Так как с объектами связаны данные, процесс их создания должен сопровождаться присвоением начальных значений. Для этого используется метод, называемый **конструктором**. Для каждого специального объекта существует свой собственный конструктор, имя которого совпадает с именем объекта. Именно этот метод отвечает за присвоение начальных значений. Поэтому при создании специального объекта так важно написать корректный конструктор.

За создание объекта отвечает конструктор.

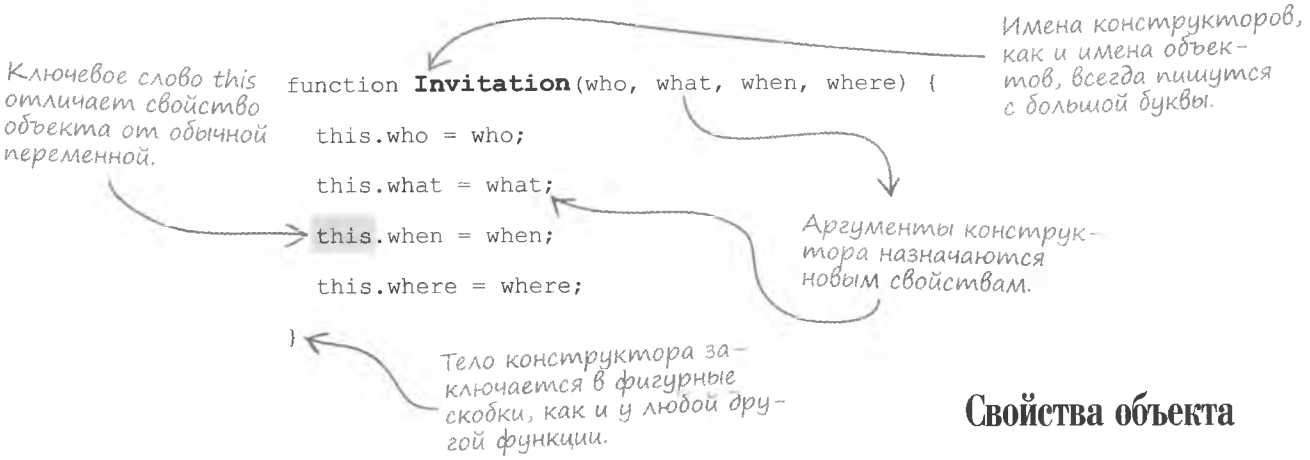


Объект с конструктором создается при помощи оператора `new`. Этот процесс напоминает вызов метода. Собственно говоря, процедура создания объекта осуществляется через вызов его конструктора. Но нельзя сделать это непосредственно, обязательно следует пользоваться оператором `new`.



Структура конструктора

Конструктор задает свойства объекта вместе с их начальными значениями. Для этого используется ключевое слово `this`. Оно связывает свойство с объектом и одновременно задает его начальное значение. С английского слово переводится как «это» и полностью оправдывает свое значение — вы создаете свойство, принадлежащее «этому» объекту, а не просто локальную переменную внутри конструктора.



Свойства объекта создаются и инициализируются конструктором при помощи оператора «точка» и ключевого слова `this`. Это ключевое слово дает конструктору понять, что он создает свойства определенного объекта. В нашем случае результатом работы конструктора стало создание четырех свойств, которым были назначены значения, переданные в качестве аргументов.

Свойства объекта в конструкторе создаются при помощи ключевого слова this.

Возьми в руку карандаш

Напишите конструктор для объекта `Blog`, который будет создавать и инициализировать дату и текст каждой записи.

.....

.....

.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит конструктор для объекта `Blog`, создающий и инициализирующий дату и текст каждой записи.

Имя конструктора совпадает с именем объекта.

```
function Blog(body, date) {
  this.body = body;
  this.date = date;
}
```

Текст записи и дата передаются в качестве аргументов.

Ключевое слово this отсылает к свойствам объекта.

Для инициализации свойств используются переданные в конструктор аргументы.

Создание объектов blog

Объект `Blog` постепенно принимает некую форму, хотя он еще не создан. В теории все прекрасно, но его существование пока только гипотеза, которую нам предстоит доказать. Конструктор формирует модель объекта, но физически у нас ничего нет, пока мы не воспользуемся оператором `new`, который создаст объект, вызвав его конструктор. Рассмотрим эту процедуру на практике.

Рукописная запись в блог.

08/14/2008
Got the new cube I ordered. It's a real pearl.

Продолжим на примерах, которые можно скачать на http://www.headfirstlabs.com/books/hfjs/.

Объект Blog.

```
var blogEntry = new Blog("Got the new cube I ordered...", "08/14/2008");
```

Блог

"Got the new cube I ordered. It's a real pearl."

"08/14/2008"

```
function Blog(body, date) {
  ...
}
```

Для создания объекта вызывается конструктор Blog().

Объект создан.

Часто Задаваемые Вопросы

В: Я не понимаю, объект создается оператором `new` или конструктором?

О: И тем и другим! Оператор `new` инициирует процедуру создания объекта и отвечает за вызов конструктора. При этом простой вызов конструктора как обычной функции без оператора `new` не приведет к созданию объекта. Точно так же бессмысленно использовать оператор `new` без конструктора.

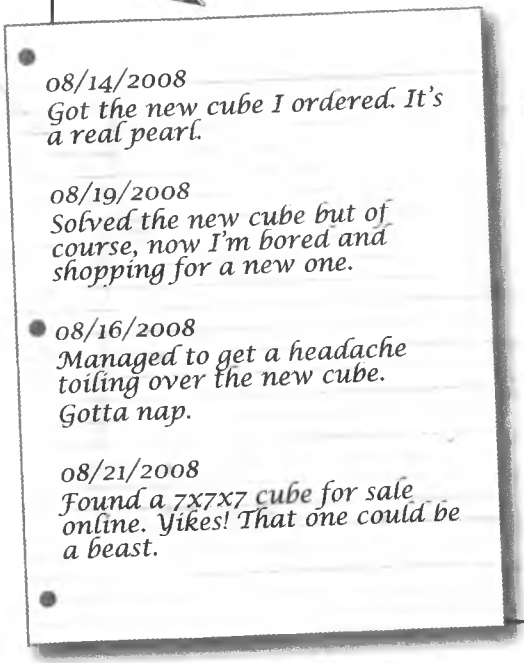
В: Всем ли специальным объектам требуется конструктор?

О: Да. Ведь именно конструктор отвечает за задание свойств объекта, соответственно, без него объект не получит никаких свойств. А значит, его существование будет бессмысленным. Исключением является случай создания чисто организационного объекта, состоящего из набора методов, не влияющих на свойства. Такой объект технически можно получить и без конструктора. Другое дело, что подобные структуры нельзя считать хорошей практикой объектно-ориентированного программирования, ведь они представляют собой не более чем набор связанных друг с другом функций. Но даже при таких условиях JavaScript сам использует организационные объекты для группировки математически связанных задач, как вы увидите в этой главе.

В: Что такое `this`?

О: Это ключевое слово, используемое для ссылки на объект. Точнее, это ссылка на объект **изнутри самого** объекта. Да, звучит несколько странно. Но представьте, что вы потеряли часы в переполненной комнате, а кто-то их нашел. Когда их вам покажут, вы закричите «Это мои часы!» Словом «мои» вы ссылаетесь на себя. Более того, это слово указывает, что именно вы являетесь собственником часов. Ключевое слово `this` аналогичным образом подразумевает **собственность** объекта. В результате запись `this.date` означает, что свойство `date` принадлежит тому объекту, в коде которого оно появилось.

Возьми в руку карандаш



Создайте массив объектов `Blog` в переменной `blog`, которой в качестве начального значения были присвоены записи блога YouCube. Записи полностью переписывать не нужно, достаточно ограничиться первыми словами.

```
var blog =
  [ .....
  .....
  .....
  ..... ];
```

Возьми в руку карандаш

Решение

08/14/2008
Got the new cube I ordered. It's a real pearl.

08/19/2008
Solved the new cube but of course, now I'm bored and shopping for a new one.

08/16/2008
Managed to get a headache toiling over the new cube. Gotta nap.

08/21/2008
Found a 7x7x7 cube for sale online. Yikes! That one could be a beast.

Вот процедура создания массива объектов `Blog` в переменной `blog`, которой в качестве начального значения были присвоены записи блога YouCube.

```
var blog =
  new Blog("Got the new cube I ordered...", "08/14/2008"),
  new Blog("Solved the new cube but of course...", "08/19/2008"),
  new Blog("Managed to get a headache toiling...", "08/16/2008"),
  new Blog("Found a 7x7x7 cube for sale...", "08/21/2008")
];
```

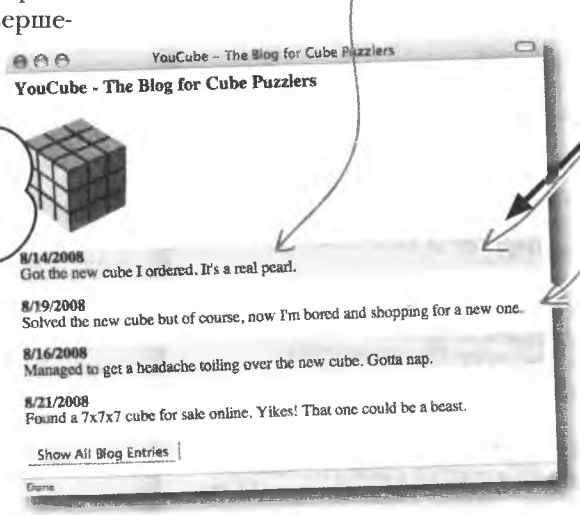
Каждая запись создается как объект Blog, содержащий текст и дату.

Версия YouCube 1.0

Скомбинировав массив объектов `Blog` с кодом JavaScript, отображающим данные, мы получим первую версию страницы YouCube. Руби знает, что работа еще далека от завершения, но первые результаты ее радуют.

Данные, хранящиеся в каждом объекте Blog, аккуратно отображаются на странице YouCube.

Мне нравится, как объект Blog скомбинировал дату и текст на странице YouCube.



Посмотрим, какой же код вдохнул жизнь в объекты `Blog` и сделал страницу YouCube 1.0 реальностью...

Подробно про YouCube

```
<html>
  <head>
    <title>YouCube - The Blog for Cube Puzzlers</title>

    <script type="text/javascript">
      // Конструктор объекта Blog
      function Blog(body, date) {
        // Assign the properties
        this.body = body;
        this.date = date;
      }

      // Глобальный массив записей в блог
      var blog = [ new Blog("Got the new cube I ordered..", "08/14/2008"),
                   new Blog("Solved the new cube but of course...", "08/19/2008"),
                   new Blog("Managed to get a headache toiling...", "08/16/2008"),
                   new Blog("Found a 7x7x7 cube for sale online...", "08/21/2008") ];

      // Показать список записей в блог
      function showBlog(numEntries) {
        // При необходимости редактируем количество записей для показа всего блога
        if (!numEntries)
          numEntries = blog.length;

        // Показать записи блога
        var i = 0, blogText = "";
        while (i < blog.length && i < numEntries) {
          // Используйте серый фон для всех остальных записей блога
          if (i % 2 == 0)
            blogText += "<p style='background-color:#EEEEEE'>";
          else
            blogText += "<p>";

          // Генерируем и форматируем HTML-код блога
          blogText += "<strong>" + blog[i].date + "</strong><br />" + blog[i].body + "</p>";

          i++;
        }

        // Располагаем HTML-код блога на странице
        document.getElementById("blog").innerHTML = blogText;
      }
    </script>
  </head>

  <body onload="showBlog(5);">
    <h3>YouCube - The Blog for Cube Puzzlers</h3>
    <img src="cube.png" alt="YouCube" />
    <div id="blog"></div>
    <input type="button" id="showall" value="Show All Blog Entries" onclick="showBlog();" />
  </body>
</html>
```

Конструктор Blog() создаёт два свойства блога.

Массив объектов Blog.

Функция showBlog() выводит записи блога под атрибутом div.

Если количество выводимых за один раз записей не передано в качестве аргумента, выводим все записи.

Изменим фоновый цвет записей блога, чтобы облегчить их чтение.

Поместим код форматированной записи блога под атрибутом div.

Изначально пустой атрибут div заполняется отформатированными записями.

При щелчке на этой кнопке становятся видимыми все записи блога.

Часто Задаваемые Вопросы

В: Зачем блогу YouCube кнопка Show All Blog Entries?

О: В текущем состоянии блога эта кнопка действительно не нужна. Ведь у нас всего четыре записи. Но по мере заполнения блога потребуется ограничить количество записей, отображаемых на главной странице. В противном случае загрузка будет занимать слишком много времени. Поэтому по умолчанию показываются только первые пять записей. Кнопка Show All Blog Entries позволяет вывести все записи на одной странице.

В: Почему для показа записей блога используется свойство `innerHTML` вместо методов DOM?

О: Хотя методы DOM и являются предпочтительными с точки зрения совместимости с веб-стандартам, реализация динамически генерируемого отформатированного HTML-кода с их помощью будет слишком громоздкой. Ведь каждый тег `<p>` или `<strong>` должен быть родительским по отношению к узлам с содержимым. Свойство же `innerHTML` позволяет значительно упростить код блога YouCube.

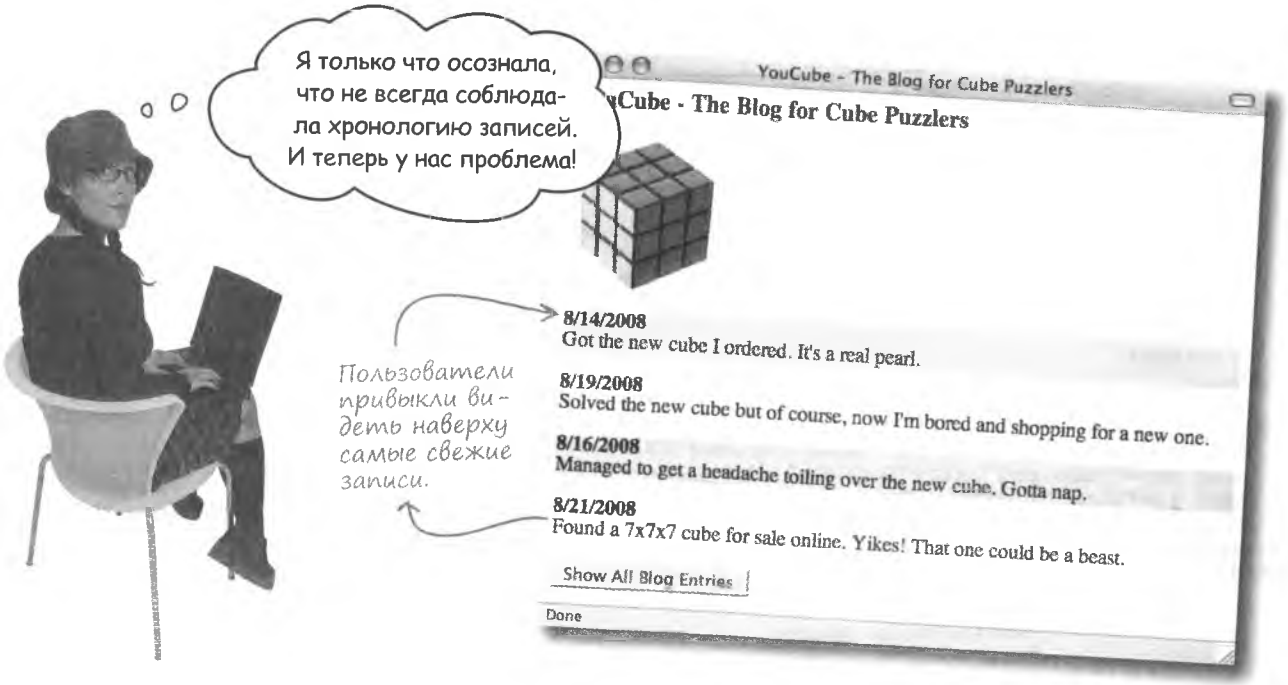
В: Почему объект `Blog` не имеет методов?

О: Их нет на данном этапе. Дело в том, что есть более первоочередные задачи, которые следует реализовать, поэтому до методов объекта `Blog` мы просто пока не добрались. Но не волнуйтесь. Методы являются существенной частью любого хорошо сконструированного объекта, и объект `Blog` в этом отношении не исключение.

Беспорядочный блог

Блог YouCube 1.0 выглядит хорошо, но этого нельзя сказать о порядке его записей. Руби заметила, что в настоящий момент они отображаются в том порядке, в котором были сохранены, в то время как с хронологической точки зрения первыми должны быть показаны самые новые записи.

Самые свежие записи должны показываться первыми.

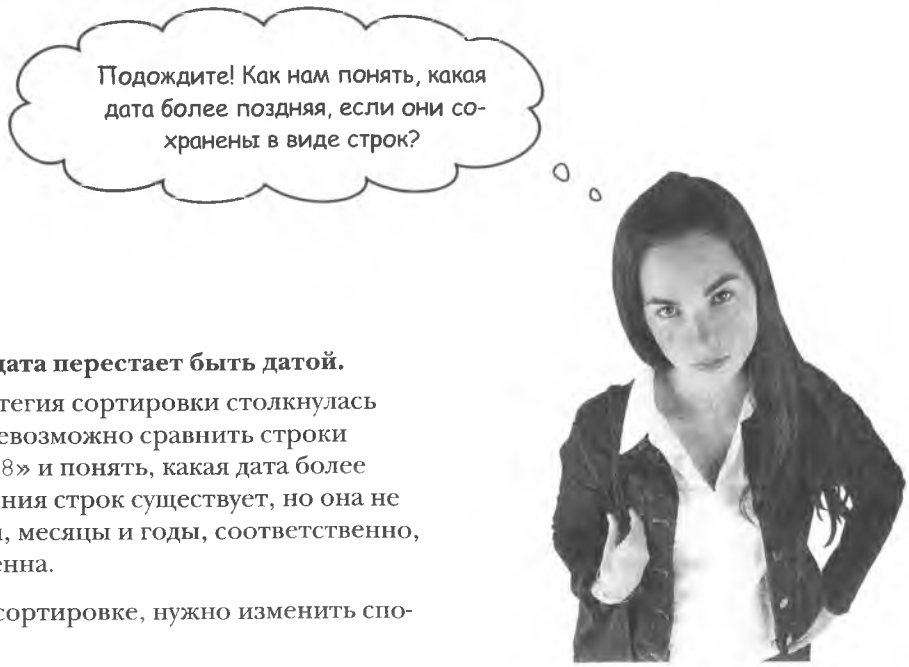


Необходимость сортировки

Проблему с порядком записей можно решить путем сортировки массива `blog` по дате. Так как JavaScript поддерживает циклы и сравнения, мы может просмотреть в цикле все записи, сравнив их даты, и отсортировать в обратном хронологическом порядке.

1. Просмотрим массив в цикле.
2. Сравним дату каждого объекта `Blog` со следующим объектом.
3. Если следующая запись является более свежей, меняем записи местами.

Подобная сортировка имеет свои достоинства и, кажется, должна неплохо работать, если мы как следует продумаем процедуру сравнения дат.



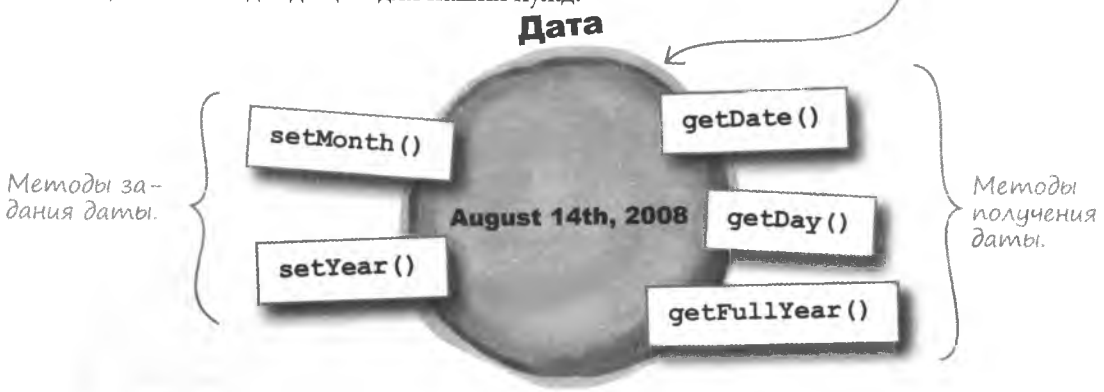
Сохраненная в виде строки дата перестает быть датой.

Итак, придуманная нами стратегия сортировки столкнулась с серьезным препятствием. Невозможно сравнить строки «`08/14/2008`» и «`08/19/2008`» и понять, какая дата более ранняя. Нет, методика сравнения строк существует, но она не поможет нам сопоставить дни, месяцы и годы, соответственно, в нашем случае она бессмысленна.

Значит, прежде чем думать о сортировке, нужно изменить способ хранения дат в блоге.

Объект для дат

Руби требуется такой способ сохранения дат, который даст возможность их дальнейшего сравнения. Другими словами, дата должна вести себя как дата. А ведь аналогичные слова мы говорили в отношении объекта! И действительно, в JavaScript существует встроенный объект `Date`, вполне подходящий для наших нужд.



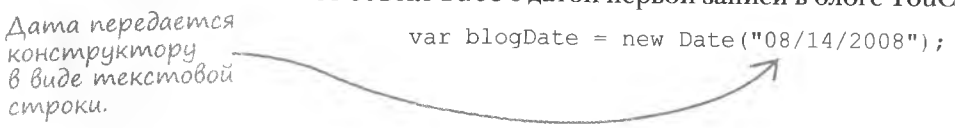
Встроенный объект Date представляет некий момент времени.

Объект `Date` представляет определенный момент времени, выраженный в миллисекундах, и является стандартной частью JavaScript. Хотя свойства этого объекта и используются, вам они не видны. Поэтому работа с объектом `Date` осуществляется исключительно через его методы.

Объект `Date`, как и ранее объект `Blog`, создается при помощи оператора `new`. Вот пример создания объекта, представляющего текущую дату и время.

Время внутри объекта Date выражено в миллисекундах.

Этот объект `Date` создан и инициализирован с текущими датой и временем. Обратите внимание на то, что синтаксис напоминает вызов метода или функции, так как вам приходится вызывать конструктор объекта `Date`. Конструктору в качестве аргумента можно передать строку с датой, отличной от текущей. Скажем, вот объект `Date` с датой первой записи в блоге YouCube:



Вычисление времени

Одним из самых больших достоинств объектов является их способность управлять своим состоянием. Например, представьте, как сложно было бы вручную вычислять количество дней между двумя датами. Потребовалось бы преобразовать дату в дни с известной точкой отсчета, а также учесть, что с началом года начинается новый цикл отсчета. Поэтому мы отдадим эту задачу на откуп объекту `Date`. Посмотрите, как легко он с ней справляется:

Эта функция принимает в качестве аргументов два объекта Date.

Преобразование миллисекунд в секунды, затем в минуты, часы и наконец в дни!

```
function getDaysBetween(date1, date2) {
  var daysBetween = (date2 - date1) / (1000 * 60 * 60 * 24);
  return Math.round(daysBetween);
}
```

Этот простой, но мощный код делает всю работу!

Вернем результат после округления... Метод round() принадлежит объекту Math, с которым мы еще познакомимся в этой главе.

```
getDaysBetween(date1, date2);
```

Данная функция демонстрирует нам, насколько мощным является объект `Date`, на простом фрагменте кода — вычитании. Сложные вычисления разницы между двумя датами скрыты внутри объекта. Нам выдается результат вычитания, представляющий собой количество миллисекунд между указанными датами. Достаточно преобразовать миллисекунды в дни и округлить результат — и у нас под рукой полезнейшая функция, которую можно вызывать всякий раз в подобных случаях.

Упражнение

Создайте два объекта `Date` для первых двух записей блога YouCube. Затем при помощи функции `getDaysBetween()` вычислите, с каким промежутком они написаны, и выведите результат в отдельном окне.

.....

.....

.....

Упражнение Решение

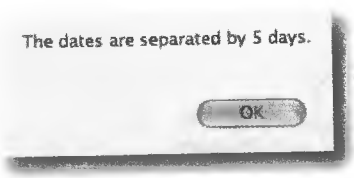
Вот как при помощи функции `getDaysBetween()` вычислить промежуток между двумя первыми записями в блоге, которые представлены при помощи двух объектов `Date`.

Создание объекта Date для двух записей блога.

```
var blogDate1 = new Date("08/14/2008");
var blogDate2 = new Date("08/19/2008");
alert("The dates are separated by " + getDaysBetween(blogDate1, blogDate2) + " days.");
```

Функция возвращает разницу.

Два объекта Date переданы функции в качестве аргументов.



Пересмотр дат в блоге

Хотя мы и познакомились с объектом `Date`, позволяющим легко управлять датами, даты в нашем объекте `Blog` пока еще сохранены в виде строк. А значит, воспользоваться преимуществами, которые нам дает объект `Date`, мы можем только после преобразования дат.

Вопрос в том, можно ли в свойстве `date` объекта `Blog` хранить объект `Date` вместо строки?

Объект внутри другого объекта

На примере нашего блога мы увидим, как одни объекты могут становиться контейнерами для других. У объекта `Blog` есть два свойства, которые, в свою очередь, являются объектами `String`. Они не выглядят как объекты, потому что создаются как литералы, простым цитированием строки текста. Объекты `Date` не столь гибки и потому создаются при помощи оператора `new`.

Чтобы создать свойство блога `date` в виде объекта `Date`, нам тоже потребуется оператор `new`, который сформирует новый объект в процессе создания объекта `Blog`. Это описание может звучать пугающе, но код выглядит достаточно просто.

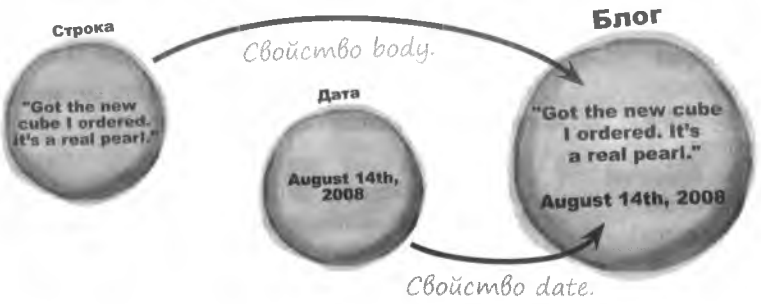
Строковый литерал автоматически создает объект String.

Объект Blog создается оператором new.

```
var blogEntry = new Blog("Nothing going on but the weather.",
                         new Date("10/31/2008"));
```

Объект Date создается и передается конструктору Blog() также при помощи оператора new.

Этот код показывает, каким образом запись в блоге YouCube создается в виде объекта, содержащего два других объекта (`String` и `Date`). Разумеется, для полной имитации блога нам потребуется массив объектов `Blog`.



Оператор new создает объекты с помощью конструктора.

Возьми в руку карандаш

Перепишите код создания массива объектов `Blog` с учетом того, что каждая дата теперь является объектом `Date`. Текст записей давайте в сокращенном виде.

Возьми в руку карандаш

Решение

Вот как выглядит код создания массива объектов `Blog` с учетом того, что каждая дата теперь является объектом `Date`.

Все записи до сих пор создаются как объекты Blog.

```
var blog = [ new Blog("Got the new cube I ordered...", new Date("08/14/2008")),
  new Blog("Solved the new cube but of course...", new Date("08/19/2008")),
  new Blog("Managed to get a headache toiling...", new Date("08/16/2008")),
  new Blog("Found a 7x7x7 cube for sale...", new Date("08/21/2008")) ];
```

Строковые литералы прекрасно подходят для представления текста записей блога.

Дата для каждого объекта Blog теперь создается как объект Date.

Часто Задаваемые Вопросы

В: Почему даты в объекте `Date` сохраняются в миллисекундах?

О: Ну, во-первых, объект `Date` представляет некоторый момент. Если бы для вселенной можно было нажать кнопку Pause, мы получили бы такой момент. Но чтобы объяснить другим людям, когда случился этот момент, требуется некая точка отсчета. В качестве такой точки выбрано 1 января 1970 года. Теперь нужно показать сдвиг относительно этой точки. Пусть он будет равен 38 годам, 8 месяцам, 14 дням, 3 часам, 29 минутам и 11 секундам. Но это слишком громоздкий способ представления данных. Проще выбрать единицу измерения, которая в состоянии представить самую маленькую долю.

Именно поэтому мы пользуемся миллисекундами. И вместо набора различных единиц у нас появляется 1,218,702,551,000 миллисекунд. Да, это много, но JavaScript умеет работать с большими числами.

В: Должен ли я вручную преобразовывать миллисекунды при работе с объектом `Date`?

О: Зависит от ситуации. Объект `Date` снабжен набором методов, умеющих извлекать значимые части, что избавляет вас от необходимости иметь дело с миллисекундами. Но, если нужно определить **разницу** между датами, от миллисекунд вам никуда не деться.

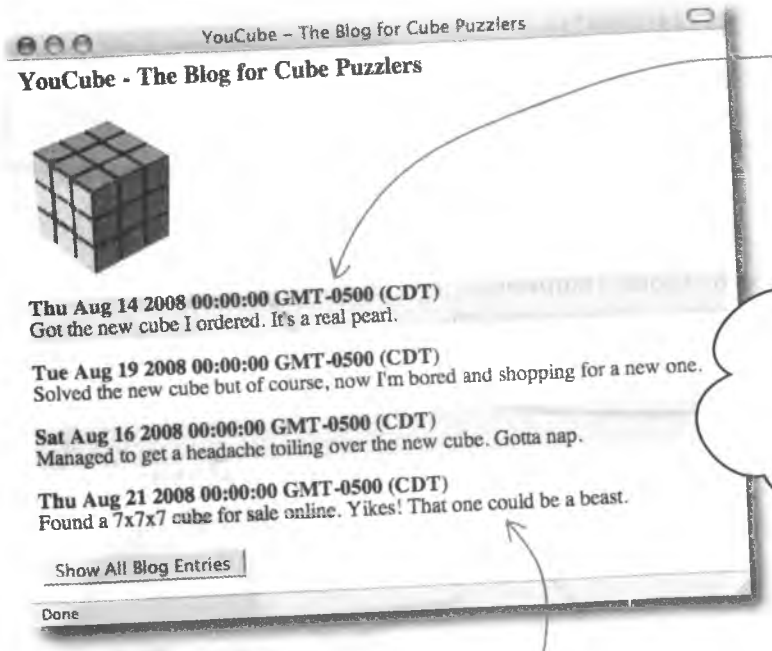
КЛЮЧЕВЫЕ МОМЕНТЫ

- Стандартный объект `Date` представляет **момент времени**, выраженный в миллисекундах.
- Объект `Date` снабжен методами доступа к различным фрагментам даты и времени.
- Объект `Date` умеет совершать математические операции с датами и даже сравнивать их друг с другом.
- В отличие от объекта `String` объект `Date` создается при помощи оператора `new`.

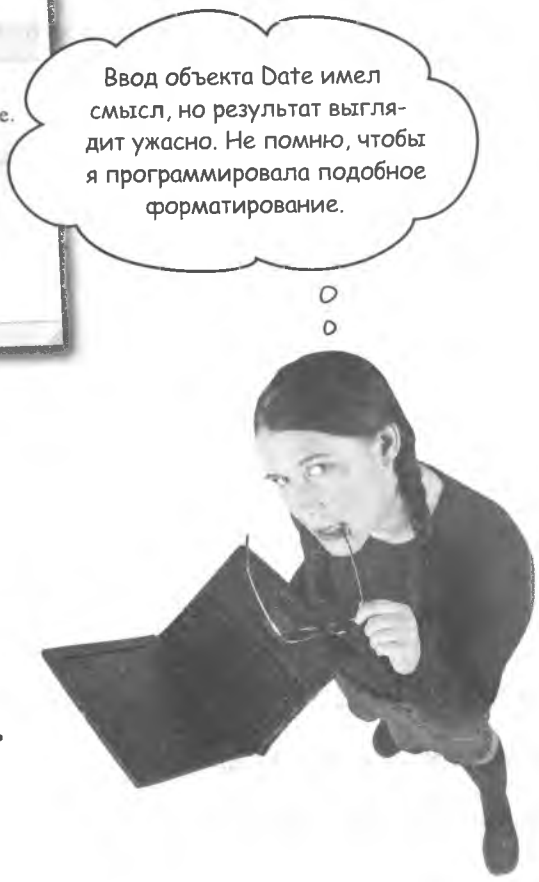
Бесполезные даты

Преобразовав свойство `date` объекта `Blog` в объект `Date`, Руби приготовилась вплотную заняться сортировкой записей. Но она столкнулась с новой проблемой — даты выглядят чересчур сложно. Руби подозревает, что читателям блога не важна информация о временной зоне, в которой она находится, и это будет отвлекать их. Итак, нам нужно более детально рассмотреть процедуру вставки в блог объектов `Date`!

Даты записей слишком громоздки и дают избыточную информацию!



Дело не только в ужасно выглядящих датах. Записи блога все еще не упорядочены!



Руби озадачена странным видом дат в блоге YouCube, ведь она не писала кода для отображения их в подобном виде. Она всего лишь преобразовала строки с датами в объекты `Date`. Неужели это заговор злых сил JavaScript?

Преобразование объектов в текст

К счастью, это не злые силы нужно обвинять в безобразном виде дат. Более того, это обычное поведение объектов JavaScript, отвечающих за форматирование дат, – то есть даты форматируются автоматически! Это работает следующим образом: все объекты JavaScript обладают методом `toString()`, обеспечивающим их текстовое представление. И наши даты являются результатом действия метода `toString()` объекта `Date`.

```
var blogDate = new Date("08/14/2008");
alert(blogDate.toString());
```

Проблема в том, что метод `toString()` автоматически срабатывает, когда объект используется вместо строки. Например, код вызова всплывающего окна с датой записи можно написать вот так, получив тот же самый результат:

```
alert(blogDate);
```

Метод toString() показывает, каким образом объект Date следит за временем.

В качестве аргумента функции alert() требуется строка, поэтому автоматически вызывается метод toString() для преобразования объекта Date.

Метод toString() не только дает строковое представление даты, но появляется и в других объектах.

Метод toString() обеспечивает строковое представление объекта.

Так как функция `alert()` принимает в качестве аргумента строки, объект `Date` знает, что ему необходимо преобразование в другой формат. Для этого он самостоятельно вызывает метод `toString()`.

Действие метода `toString()` не стало бы проблемой, если бы даты отображались в простом формате, например ММ/ДД/ГГГГ. То есть наш блог YouCube не может воспользоваться преимуществом заданного по умолчанию представления строк.

Доступ к фрагментам даты

Руби нужен способ регулировки формата даты. Для этого нам потребуется доступ к отдельным фрагментам, а именно к месяцам, дням и годам. После чего мы сможем собрать дату нужным нам образом. К счастью, объект `Date` обладает всеми необходимыми для этого методами.

Объект `Date` обладает не только тремя вышеуказанными, но и другими методами, обеспечивающими различные способы доступа к дате и времени. Однако для наших целей вполне достаточно этих трех.

Обратите внимание на значения, возвращаемые методами объекта Date.

Метод getMonth() возвращает месяц в виде чисел от 0 (январь) до 11 (декабрь), в то время как метод getDate() возвращает число в диапазоне от 1 до 31.

Возьми в руку карандаш

Устраните проблему с записью дат в блоге YouCube, переписав код, форматирующий запись блога и сохраняющий его в переменной `blogText`. Убедитесь, что даты имеют формат *ММ/ДД/ГГГГ*. Вот исходная версия кода:

```
blogText += "<strong>" + blog[i].date + "</strong><br />" + blog[i].body + "</p>";
```

Возьми в руку карандаш

Решение

Вот как выглядит в результате переписанный код, форматирующий запись блога и сохраняющий его в переменной `blogText`.

```
blogText += "<strong>" + blog[i].date + "</strong><br />" + blog[i].body + "</p>";
```

```
blogText += "<strong>" + (blog[i].date.getMonth() + 1) + "/" +
blog[i].date.getDate() + "/" +
blog[i].date.getFullYear() + "</strong><br />" +
blog[i].body + "</p>";
```

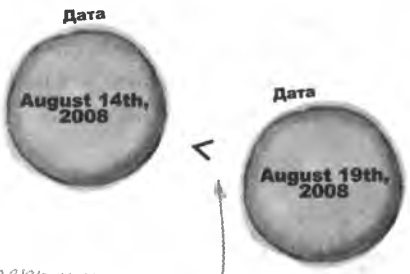
Для получения нужного формата произведем форматирование объекта Date вручную.

Каждый фрагмент даты извлекается из объекта Date при помощи соответствующего метода.

Дата теперь отображается в привычном формате ММ/ДД/ГГГГ.

Даты упрощают сортировку

Теперь, когда даты в блоге представлены в виде объектов `Date`, что лучше отвечает нашим целям, вернемся к вопросу о порядке записей. В настоящее время они отображаются в том порядке, в котором оказались сохранены в массиве `blog`, в то время как нам требуется, чтобы самые свежие записи были наверху. Рассмотрим еще раз процедуру сортировки:



Теперь у нас есть объекты Date, которые можно сравнить друг с другом.

1. Просмотрим массив в цикле.
2. Сравним дату каждого объекта `Blog` со следующим.
3. Если следующая запись является более свежей, меняем записи местами.

Хотя благодаря объекту `Date` та часть стратегии, которая относится к сравнению дат, стала выглядеть менее устрашающей, остальная часть плана все еще требует написания изрядных фрагментов кода. С другой стороны, невозможно поверить, чтобы такая проблема, как сортировка наборов данных, не решалась много раз до этого. А значит, изобретать велосипед лично вам не потребуется...


А здорово было,бы если бы
в JavaScript была встроенная
функция для выполнения рутинной
работы по сортировке данных...

Массивы как объекты

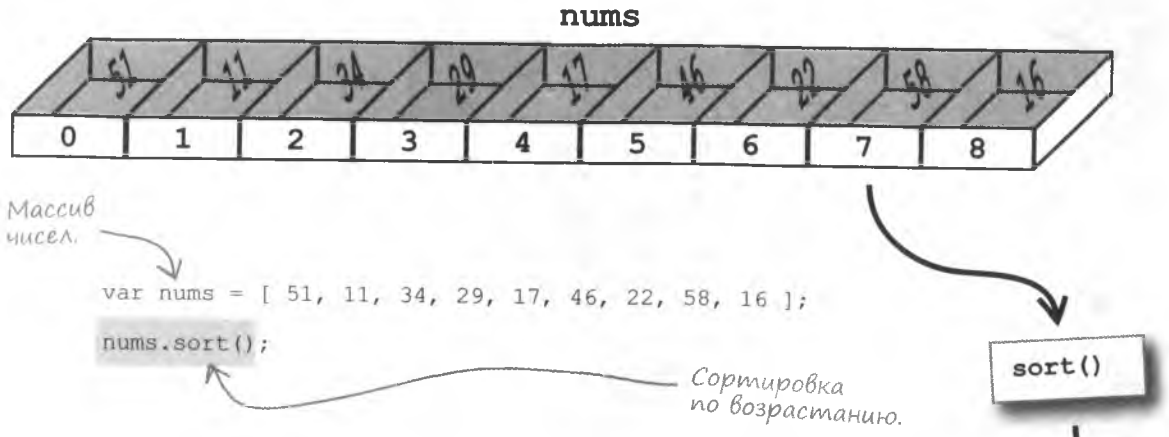
Как вы думаете, может ли массив сам себя сортировать? Это предположение не лишено смысла, в конце концов, данные же умеют самостоятельно осуществлять преобразование в строковый формат. Но чтобы такое стало возможным, массив должен быть объектом и обладать соответствующими методами. Самое интересное, что так оно и есть. Помните код из сценария Mandango?


```
for (var i = 0; i < seats.length; i++) {
  ...
}
```

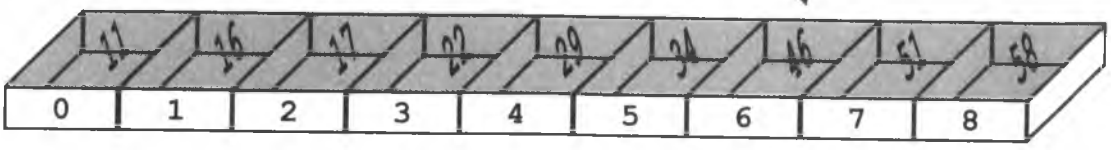
Переменная seats является массивом.

length — это свойство массива, предоставляющее информацию о количестве элементов в нем.

Итак, массивы являются объектами. Означает ли это, что они могут сортировать себя сами? Массивы обладают не только свойством `length`, но еще и методами обработки своих данных. В том числе и методом `sort()`, который нам нужен. Посмотрим, как именно он работает.



Метод `sort()` меняет порядок элементов внутри массива. По умолчанию сортировка осуществляется по возрастанию, поэтому массив `nums` приобретет вот такой вид:



Пользовательская сортировка

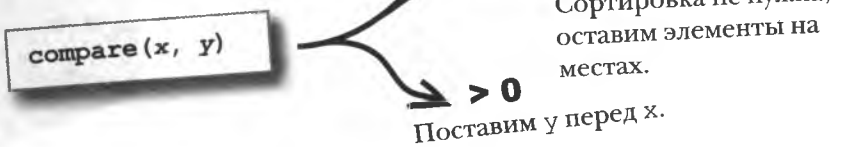
Заданного по умолчанию поведения метода `sort()` объекта `Array` часто недостаточно. Но на помощь приходит тот факт, что процедура сортировки определяется функцией сравнения, которую вызывает метод `sort()`. Значит, вы можете повлиять на сортировку, написав свой вариант этой функции. Вот как она обычно выглядит:

```
function compare(x, y) {

  return x - y;

}
```

В качестве аргументов выступают два элемента массива, которые сравниваются с целью сортировки.

Возвращаемое значение определяет, останутся ли x и y на своих местах или же поменяются друг с другом.

Функция `compare()` возвращает число, определяющее результирующее положение `x` и `y`.



Написанная функция `compare()` вставляется в уравнение сортировки массива при вызове метода `sort()` — достаточно передать ссылку на нее этому методу.

```
nums.sort(compare);
```

Сортировка массива теперь управляется написанной вами функцией compare().

Возьми в руку карандаш

Напишите код для функции `compare()`, располагающей массив записей блога YouCube в обратном хронологическом порядке. Подсказка: вычитание объектов `Blog` друг из друга осуществляется при помощи обычного знака минус.

.....

.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит код функции `compare()`, располагающей массив записей блога YouCube в обратном хронологическом порядке.

```
function compare(blog1, blog2) {
  return blog2.date - blog1.date;
}
```

Аргументами являются объекты Blog, ведь именно из них состоит массив.

Вычитание первой даты из второй приводит к сортировке в обратном хронологическом порядке.

Две даты вычитаются друг из друга, как числа (миллисекунды).

Упрощение сортировки

Написанная нами функция сравнения элементов массива используется исключительно в методе `sort()`. А значит, именованная функция нам в данном случае не требуется.

Помните литералы функции для термостата, с которым мы работали в главе 6? Функция `compare()` также является превосходным кандидатом на эту роль. Более того, сортировка записей блога упростится, если эту функцию превратить в литерал, непосредственно передаваемый методу `sort()`.

Литерал функции передается непосредственно методу sort() массива.

```
blog.sort(function(blog1, blog2) {
  return blog2.date - blog1.date;
});
```

Как любитель головоломок, Руби на первое место ставит эффективность. А удаление ненужной именованной функции увеличивает эффективность метода `sort()`. Кроме того, Руби не понимает, зачем функции сравнения целых три строчки кода. Наш литерал достаточно прост, чтобы обойтись всего одной строчкой.

Литерал функции сократился до одной строчки кода.

```
blog.sort(function(blog1, blog2) { return blog2.date - blog1.date; });
```

Часто Задаваемые Вопросы

В: Любой ли объект обладает методом `toString()`?

О: Да. Даже если вы создадите специальный объект, не снабдив его методом `toString()`, JavaScript по крайней мере укажет на существование такого объекта при попытке использовать его там, где требуется строка.
В некоторых случаях строки не имеют особого значения, но вы можете по своему желанию добавить объекту метод `toString()`, чтобы получить возможность читать его значение.

В: Каким образом функция сравнения работает с объектами `Date` objects?

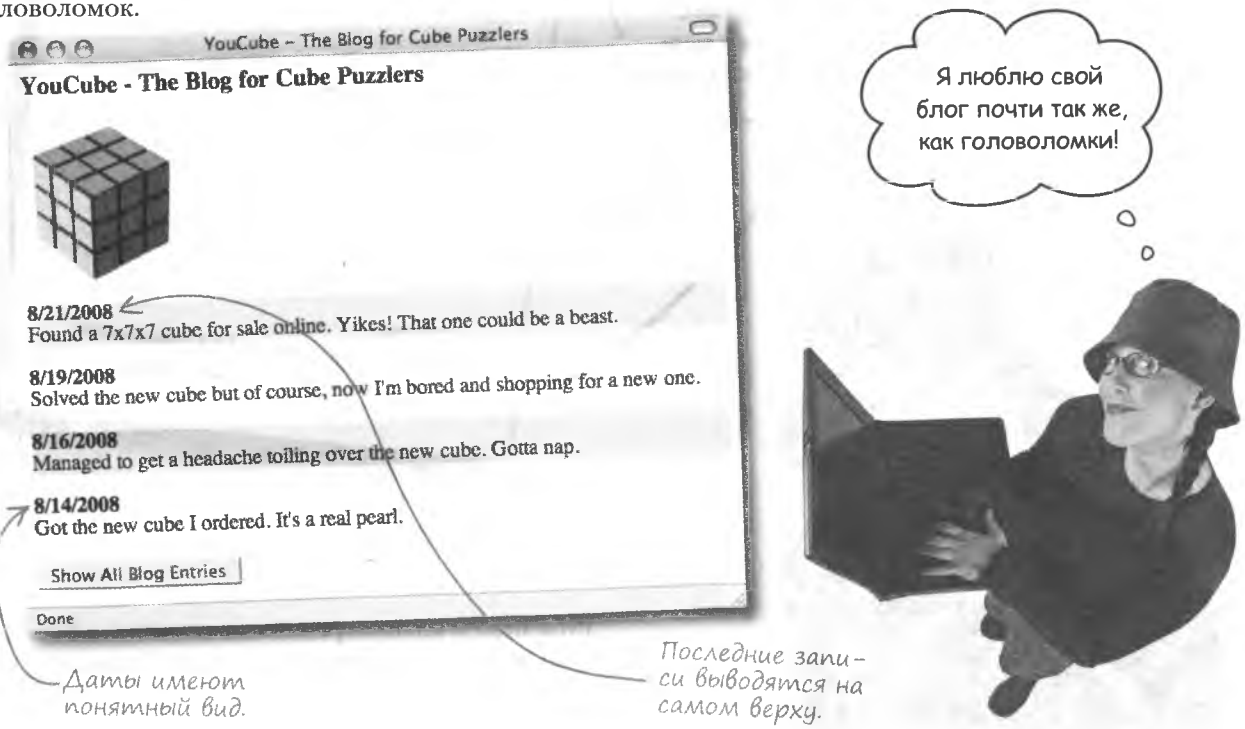
О: Функция сравнения возвращает число, значение которого отвечает за сортировку двух аргументов. При сравнении дат нужно, чтобы более поздняя дата шла впереди. Более поздняя дата больше, поэтому определить порядок следования можно при помощи обычного вычитания. Вторая дата оказывается перед первой только при условии, что она больше (результат вычитания больше 0).

В: Как метод `Array.sort()` определяет, использовать ему пользовательскую функцию сравнения или функцию по умолчанию?

О: Это решение зависит от того, был ли функции `sort()` передан аргумент. Отсутствие аргумента означает сортировку по умолчанию. Его наличие интерпретируется как ссылка на функцию и используется как основа для сравнения сортируемых элементов. То есть ссылка на функцию сравнения является **необязательным** аргументом.

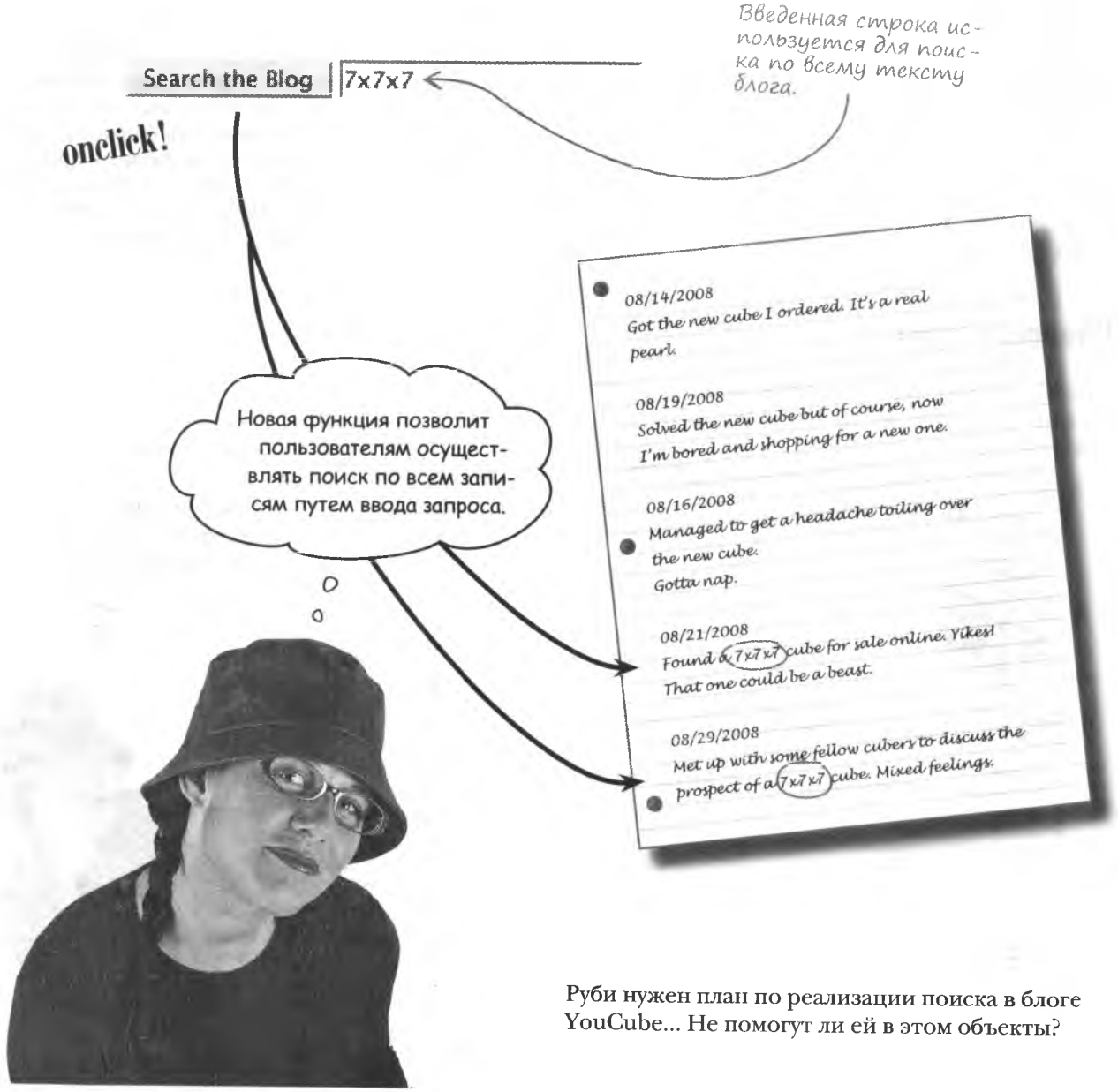
Счастье Руби

Блог YouCube постепенно приближается к представлениям Руби об идеальном месте общения с другими поклонниками головоломок.



Поиск

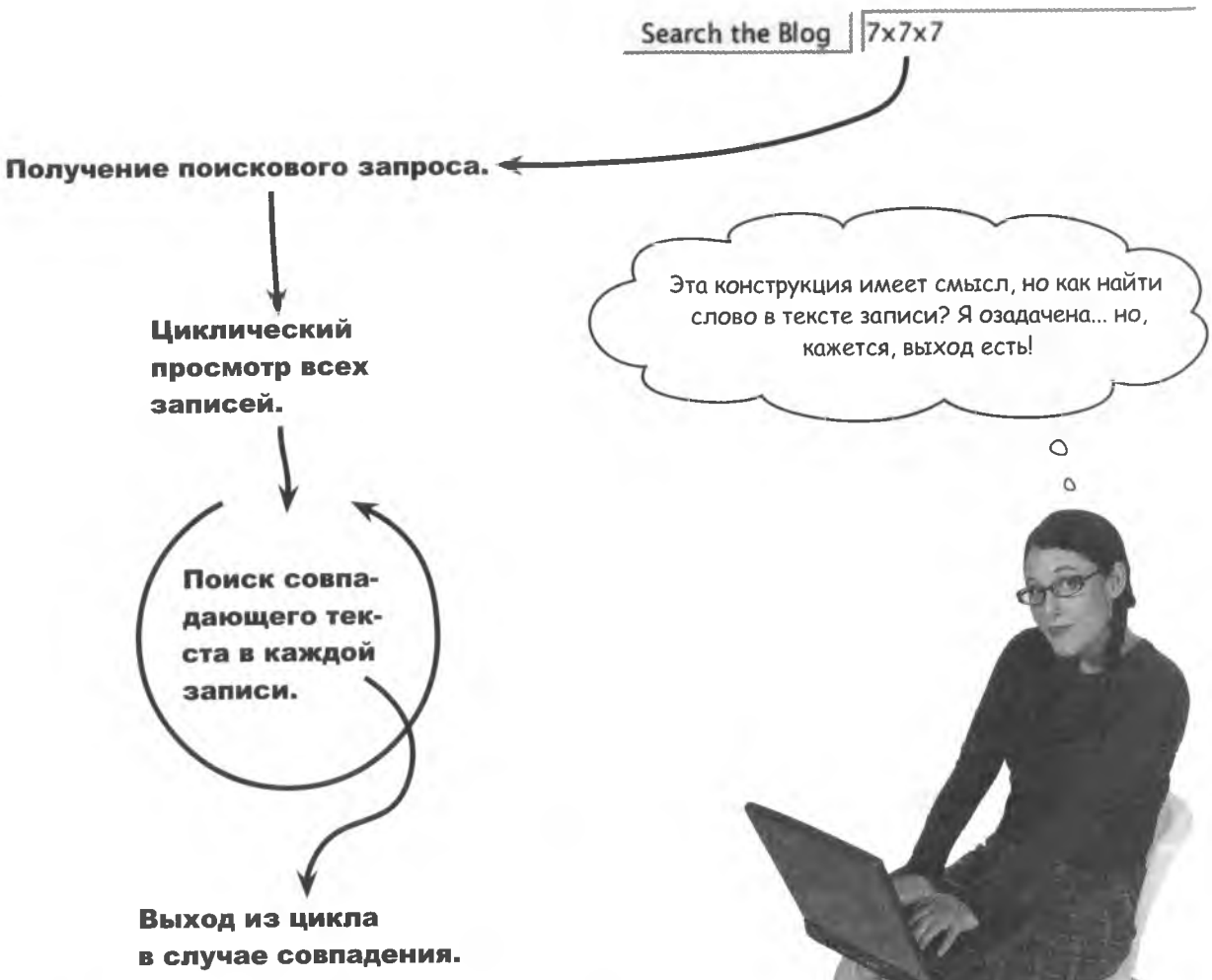
БлогYouCube прекрасно работает, но некоторые пользователи заговорили о такой функции, как поиск по всем записям. Так как Руби планирует писать туда часто и много, она согласна, что это крайне полезная функция, особенно в долгосрочной перспективе.



Руби нужен план по реализации поиска в блоге YouCube... Не помогут ли ей в этом объекты?

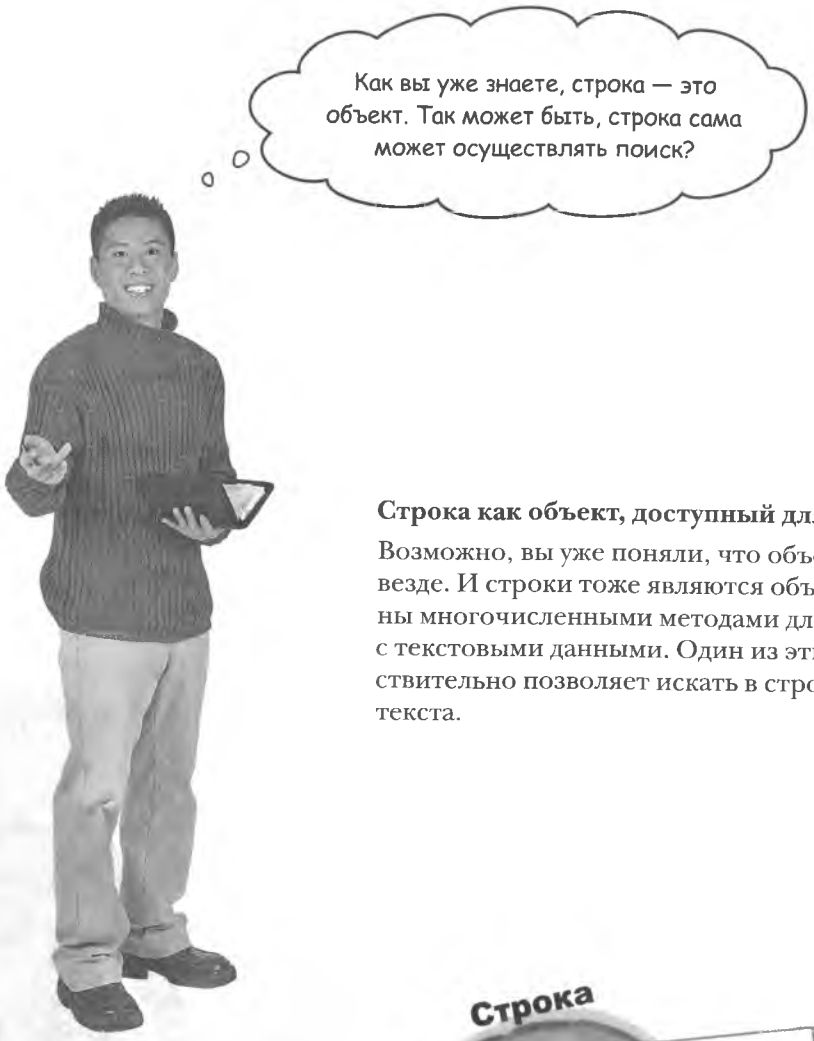
Поиск по массиву

Функция поиска в блоге YouCube включает в себя циклический просмотр массива всех записей и поиск совпадающих символов в каждой из них.



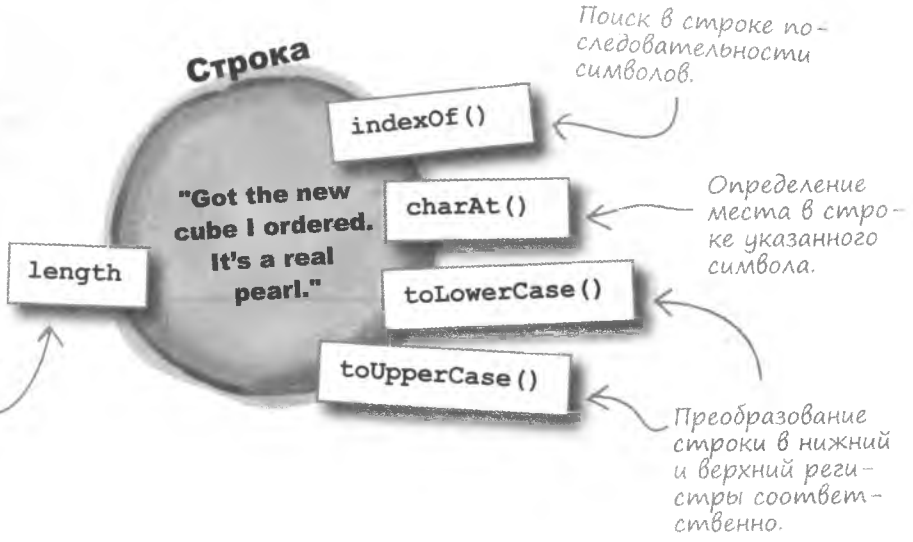
МОЗГОВОЙ ШТУРМ

Как бы вы реализовали поиск среди записей блога YouCube совпадающей строки текста?



Строка как объект, доступный для поиска.

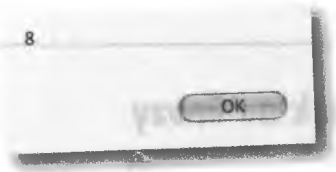
Возможно, вы уже поняли, что объекты в JavaScript везде. И строки тоже являются объектами и снабжены многочисленными методами для взаимодействия с текстовыми данными. Один из этих методов действительно позволяет искать в строке фрагменты текста.



Метод indexOf()

Метод `indexOf()` позволяет искать в объекте `String` фрагменты текста. Поисковый запрос передается методу как аргумент — так как метод принадлежит объекту `String`, других аргументов передавать не нужно. Метод `indexOf()` возвращает индекс, указывающий на местоположение обнаруженных символов, или –1, если поиск оказался безуспешным.

```
var str = "Got the new cube I ordered. It's a real pearl.";
alert(str.indexOf("new"));
```



Чтобы понять, откуда появилась цифра 8, рассмотрим строку как массив отдельных символов.

Поисковый запрос new появляется в строке под индексом 8.

Каждому символу в строке соответствует уникальный индекс. Отсчет начинается с 0, что соответствует началу строки.

Если нужной последовательности символов в строке не существует, метод `indexOf()` возвращает –1.

```
var searchIndex = str.indexOf("used");
```

В результате получаем –1, так как строка поиска не найдена в объекте String.

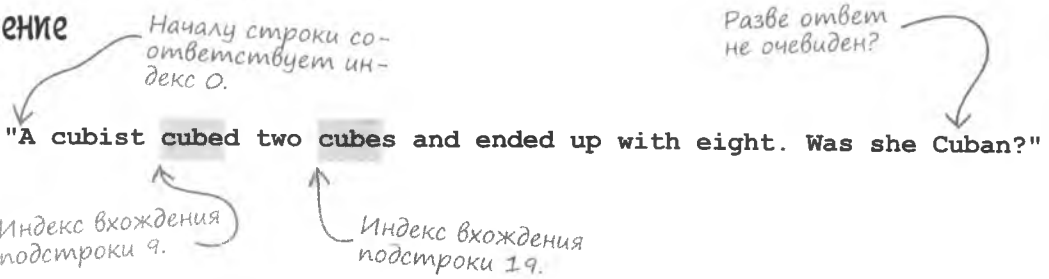
Упражнение

Ниже показана любимая загадка Руби. Определите индекс каждого вхождения подстроки «`cube`» в строку загадки.

«Кубистка возводила в куб два куба и получила восемь. Была ли она кубинкой?»

Упражнение Решение

Вот индексы вхождения подстроки «cube» в любимую загадку Руби. «Кубистка возводила в куб два куба и получила восемь. Была ли она кубинкой?».

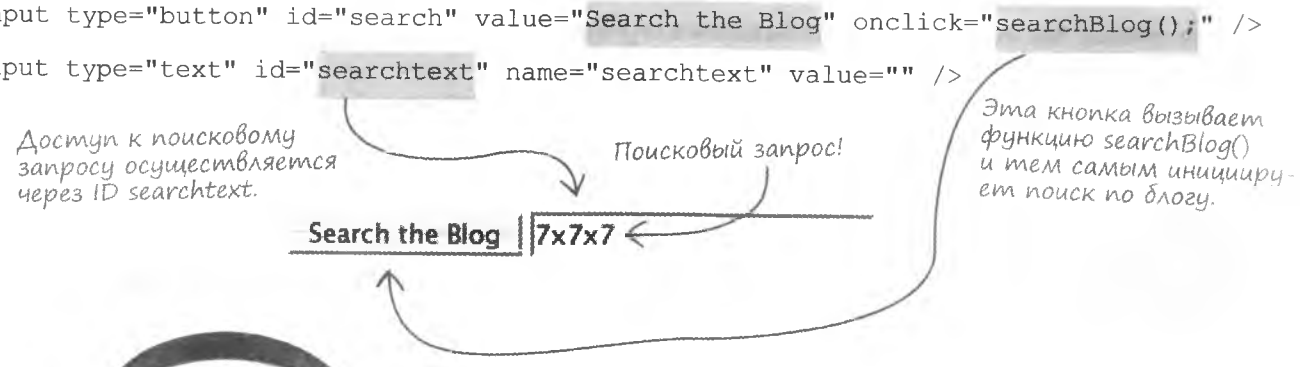


Поиск по блогу

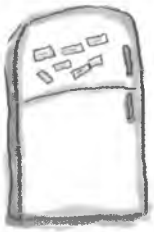
Благодаря методу `indexOf()` поиск по строкам выполняется достаточно просто, но у Руби для поиска имеется целый блог. Она собирается в цикле просматривать все записи и на каждой итерации использовать метод `indexOf()`. Обнаруженные совпадения должны выводиться в отдельном окне.

Перед тем как приступить к написанию функции поиска по блогу, нам потребуется создать текстовое поле, в которое будут вводиться поисковые запросы, и кнопку, запускающую поиск.

```
<input type="button" id="search" value="Search the Blog" onclick="searchBlog();" />
<input type="text" id="searchtext" name="searchtext" value="" />
```



Теперь, когда поисковые HTML-элементы на месте, осталось написать код функции `searchBlog()`. Так как для отображения результатов поиска эта функция использует всплывающее окно, она не возвращает никакой информации. А так как поисковый запрос считывается из текстового поля, в которое его ввел пользователь, нам не потребуются и никакие аргументы.



Магниты JavaScript

Функция `searchBlog()` отвечает за циклический просмотр записей блога и поиск текста, совпадающего с поисковым запросом. Помогите Руби закончить код, поместив магниты на пустые места. Подсказка: совпадающие результаты поиска должны отображаться вместе с датой записи в виде *ММ/ДД/ГГГГ*, выводимой в квадратных скобках после текста блога.

```
function searchBlog() {

  var ..... = document.getElementById("").value;

  for (var i = 0; i < .....; i++) {

    // Проверяем, содержит ли запись поисковый текст

    if (blog[i]......toLowerCase().indexOf(searchText.toLowerCase()) != -1) {

      alert("[" + (blog[i].....+ 1) + "/" +

        blog[i].date.getDate() + "/" + blog[i].....getFullYear() + "] " +

        blog[i].....);

      break;
    }
  }

  // Если поисковый текст не найден, сообщим об этом

  if (i == .....)

    alert("Sorry, there are no blog entries containing the search text.");
}
```

searchText

blog.length

date

1

searchtext

body

getMonth()

Решение задачи с магнитами

Вот как выглядит функция `searchBlog()`, отвечающая за циклический просмотр записей блога и поиск в них указанного пользователем текста.

Возьмем текст поискового запроса из текстового поля.

```
function searchBlog() {
  var searchText = document.getElementById("searchtext").value;
  for (var i = 0; i < blog.length ; i++) {
    // Проверяем, содержит ли запись поисковый текст
    if (blog[i].body.toLowerCase().indexOf(searchText.toLowerCase()) != -1) {
      alert("[" + (blog[i].date.getMonth() + 1) + "/" +
        blog[i].date.getDate() + "/" + blog[i].date.getFullYear() + "] " +
        blog[i].body);
      break;
    }
  }
  // Если поисковый текст не найден, сообщим об этом
  if (i == blog.length )
    alert("Sorry, there are no blog entries containing the search text.");
}
```

Если i равно свойству length объекта blog, значит, все записи уже были просмотрены и совпадений не обнаружено.

После совпадающего текста выводится дата записи в квадратных скобках в формате ММ/ДД/ГГГГ.

Поиск заработал!

Итак, готова версия YouCube 2.0 с функцией поиска, работающей благодаря свойствам объекта `String`. Вы видите, каким образом объекты делают данные активными. В данном случае строка текста (обычные данные) превратилась в структуру с поведением (она умеет осуществлять поиск). Самое главное, что Руби не пришлось изобретать собственный поисковый механизм, а освободившееся таким образом время она потратила на ведение блога.

Руби в восторге от новой функциональности блога, но она не из тех, кто почивает на лаврах. И уже подумывает о версии YouCube 3.0...

Часто Задаваемые Вопросы

В: Неужели каждая строка это действительно объект?

О: Да. Все строки в JavaScript являются объектами. Если в коде JavaScript вы поместите свое имя в кавычки, например «Ruby», вы создадите объект. Такой подход может казаться излишеством, но его преимущество в том, что каждая строка умеет делать много полезного, например: она знает свою длину, умеет искать заданные последовательности символов и многое другое.

В: Но строка так похожа на массив, даже все ее символы имеют свои индексы. Неужели она — массив?

О: Нет. Строка определенно не является массивом. Однако и в самом деле многие методы объекта `String` производят операции над данными таким образом, как будто перед нами массив, составленный из набора символов. Например, первый символ строки имеет индекс 0, и этот индекс инкрементно увеличивается. Но доступ к любому элементу массива можно получить, указав его индекс в квадратных скобках (`[]`). Со строками такое невозможно. Считайте для простоты, что вы работаете с массивом символов, учитывайте разницу между работой с объектами `String` и `Array`.

В: Можно ли в функции поиска `searchBlog()` использовать метод `charAt()` вместо `indexOf()`?

О: Нет. Метод `charAt()` ищет единичные символы, в то время как в блоге обычно хотят найти слово или даже фразу. Именно поэтому для данной цели больше подходит метод `indexOf()`, умеющий искать произвольные наборы символов.

В: Можно ли найти все вхождения поискового запроса в текст блога?

О: Да. Метод `indexOf()` по умолчанию ищет только первое вхождение. Но ему можно передать еще один необязательный аргумент, указывающий, где следует начать поиск. Предположим, вы ищете слово «cube» и обнаружили его под индексом 11. Снова вызвав метод `indexOf()` со вторым аргументом 11, вы инициируете новый поиск, начинающийся с индекса 12. Таким образом, для продолжения поиска вам нужно только передать индекс найденной строки методу `indexOf()`.

В: Зачем в функции `searchBlog()` два раза вызывается метод `toLowerCase()`?

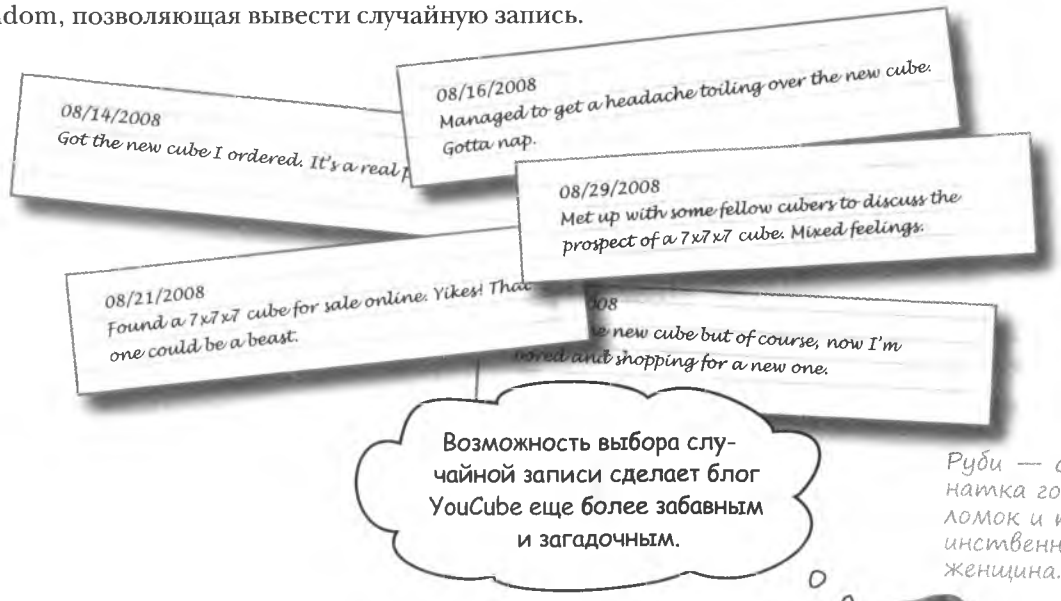
О: Ответ на этот вопрос связан с проблемой регистра при поиске по блогу. Ведь если пользователь ищет, например, слово «cube», его, скорее всего, интересуют все варианты этого слова: «cube», «Cube», «CUBE» и т. д. Проще всего эту проблему можно решить преобразованием как текста блога, так и поискового запроса к одному регистру. Хотя в нашей функции `searchBlog()` используется метод `toLowerCase()`, можно было взять и метод `toUpperCase()`. Главное, выполнить поиск без учета регистра букв.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Метод `toString()` обеспечивает текстовое представление объектов.
- Как массивы, так и строки являются объектами и получают способ хранения данных и методы от стандартных объектов `Array` и `String`.
- Метод `sort()` объекта `Array` сортирует массив в нужном вам порядке.
- Метод `indexOf()` объекта `String` ищет строку текста в другой строке, возвращая индекс обнаруженного совпадающего текста.

Случайный Выбор

В бесконечном стремлении поддержать интерес пользователей к своему блогу Руби решила добавить к нему функцию, которая, как ей кажется, должна понравиться посетителям. Это кнопка Random, позволяющая вывести случайную запись.



Возможность выбора случайной записи сделает блог YouCube еще более забавным и загадочным.

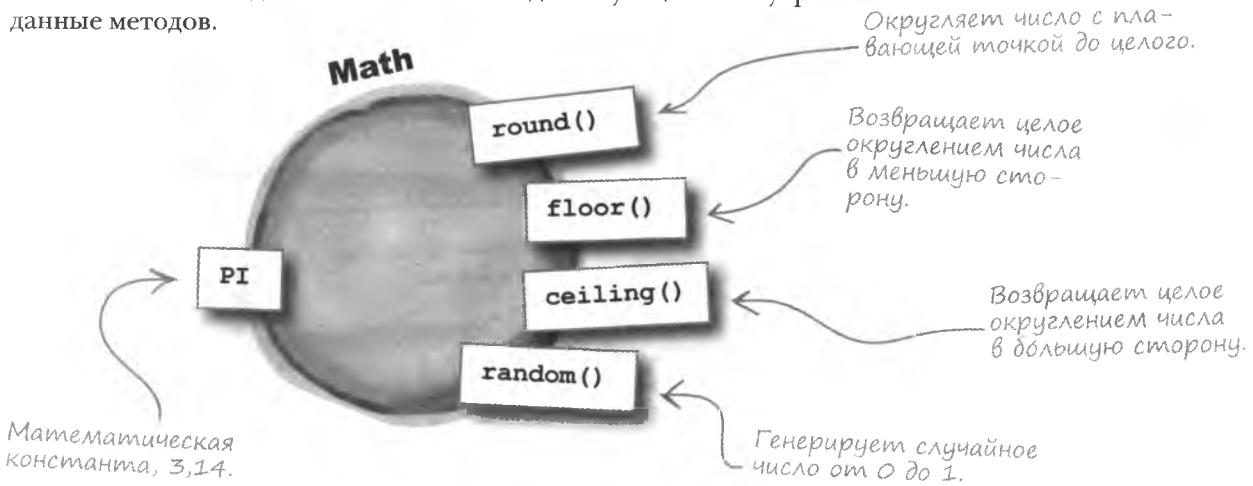
Руби — фанатка головоломок и таинственная женщина.

МОЗГОВОЙ ШТУРМ

Как вы реализовали бы случайный выбор записи из блога?

Объект Math

Чтобы добавить к блогу YouCube возможность перехода к случайной записи, нам потребуется способ генерации случайных чисел. В этом нам поможет встроенный объект JavaScript, который не является настолько «живым», как остальные знакомые вам объекты. Это стандартный объект `Math`, внутри которого генерятся случайные числа. При этом он не меняет никаких данных и не имеет воздействующих на внутренние данные методов.



Объект `Math` является набором математических методов и констант. У него нет переменных, соответственно, он не имеет состояния — вы не можете использовать его для хранения информации. Единственными данными этого объекта являются несколько констант, в том числе `PI` (3,14). Но он обладает весьма полезными методами. Один из них, метод `random()`, создает числа с десятичной точкой в промежутке от 0 до 1.

Объект Math состоит из набора математических методов и констант.

Упражнение

Напишите результаты вызова методов объекта `Math`.

```
Math.round(Math.PI)      .....
Math.ceiling(Math.PI)    .....
Math.random()            .....
```

Ответ на стр. 454.

ОБЪЕКТ MATH О СЕБЕ

Интервью недели: Когда математические функции сталкиваются

Head First: Я совсем запутался. Вы объект, но содержите в себе всего лишь набор математических методов и несколько констант. Я же думал, что объекты должны делать данные активными. То есть брать данные и при помощи методов делать с ними всякие удивительные вещи.

Math: Да, так традиционно принято думать, но не все объекты предназначены для этой цели. Некоторые из них могут играть роль организаторов, как я, например.

Head First: Но разве все эти математические методы нельзя создать в виде стандартных функций?

Math: Разумеется, можно, но вы забываете, что язык JavaScript построен из объектов. Так что по сути такой вещи, как «стандартная» функция, не существует.

Head First: Но я создавал функции вне объектов, и они прекрасно работали.

Math: На самом деле все функции являются методами, так как они принадлежат какому-либо объекту, иногда скрытому.

Head First: Значит, вот почему вы содержите эти математические методы...

Math: И не думайте, что отсутствие данных, которыми я могу управлять, исключает меня из сословия объектов.

Head First: Что вы имеете в виду?

Math: Ну, представьте людей с общими интересами. Часто они объединяются в группы по интересам. Математические методы, конечно, не так социально активны, как люди, но и они имеют выгоду от организации, которую я им предоставляю.

Head First: Вы хотите сказать, что у них много общего?

Math: Да! Все они решают математические задачи, например округляя числа, выполняя тригонометрические операции и генерируя случайные числа.

Head First: Вы упомянули генератор случайных чисел. Я слышал, что на самом деле они не случайны. Это правда?

Math: Должен признаться, что да. Они действительно не совсем случайны, как и большинство случайных чисел, генерируемых компьютером. Они «псевдослучайны», но для большинства задач этого достаточно.

Head First: Псевдослучайность — это как псевдонаука... или псевдокод?

Math: И да, и нет. Нет, ничего подобного псевдонауке. И да, похожи на псевдокод, ведь он предназначен для показа основной идеи кода, не являясь им по сути. Так и псевдослучайные числа имитируют хаос, не будучи по своей природе случайными.

Head First: То есть их можно считать достаточно случайными для большинства приложений JavaScript?

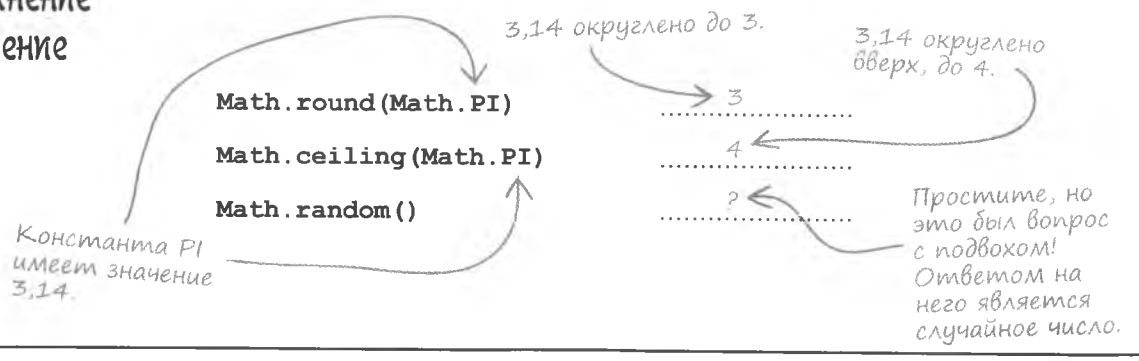
Math: Да. И это хорошее выражение: «достаточно случайные». Вряд ли стоит использовать их для задач, связанных с национальной безопасностью, но для обычных сценариев они вполне подходят.

Head First: Понятно. Спасибо за вашу откровенность по поводу случайных чисел.

Math: Рад был побеседовать... как вы знаете, я не умею врать.

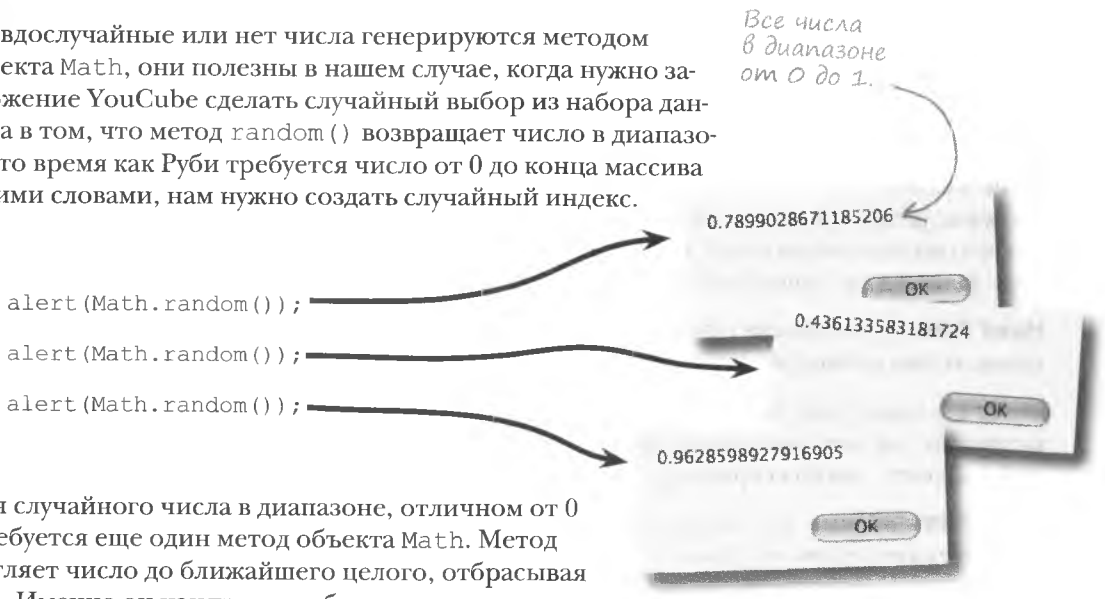
Упражнение Решение

Вот результат вызова методов объекта `Math`.



Генерация случайных чисел

Не важно, псевдослучайные или нет числа генерируются методом `random()` объекта `Math`, они полезны в нашем случае, когда нужно заставить приложение YouCube сделать случайный выбор из набора данных. Проблема в том, что метод `random()` возвращает число в диапазоне от 0 до 1, в то время как Руби требуется число от 0 до конца массива записей. Другими словами, нам нужно создать случайный индекс.



Для получения случайного числа в диапазоне, отличном от 0 до 1, вам потребуется еще один метод объекта `Math`. Метод `floor()` округляет число до ближайшего целого, отбрасывая дробную часть. Именно он наилучшим образом подходит для наших целей.

```
var oneToSix = Math.floor(Math.random() * 6) + 1;
```

0 - 5

1 - 6

Часто Задаваемые Вопросы

В: Почему не требуется создавать объект `Math`?

О: Этот вопрос касается крайне важного понятия, связанного с объектами. Так как объект `Math` не содержит данных, которые может изменять, его не требуется создавать. Как вы помните, этот объект представляет собой всего лишь набор статических методов и констант, так что все его составные части уже существуют — и вам просто нечего создавать. Впрочем, все это станет понятнее в главе 10, когда мы приступим к изучению классов и реализаций объектов.

В: В чем разница между методами `round()` и `floor()` объекта `Math`?

О: Метод `round()` округляет в зависимости от значения десятичной части. Например, `Math.round(11,375)` даст в результате 11, в то время как `Math.round(11,625)` — 12. Метод же `floor()` всегда округляет путем отбрасывания десятичной части, не важно, насколько она велика.

В: Что еще умеет объект `Math`?

О: Многое. Например, два полезных метода, которые нам пока не требовались — `min()` и `max()`, анализируют два числа и возвращают меньшее или большее из них. Метод `abs()` возвращает положительное число вне зависимости от переданного ему аргумента.

Возьми в руку карандаш

Напишите код функции `randomBlog()`, случайным образом выбирающей запись из блога и отображающей ее в отдельном окне. Подсказка: выведенная в отдельном окне запись может быть отформатированна так же, как и результаты поиска в методе `searchBlog()`.

Возьми в руку карандаш

Решение

Вот код функции `randomBlog()`, случайным образом выбирающий запись из блога и отображающий ее в отдельном окне.

Используйте случайное число для выбора записи.

Генерирует случайное число в диапазоне от 0 до числа на единицу меньшего, чем количество записей в блоге.

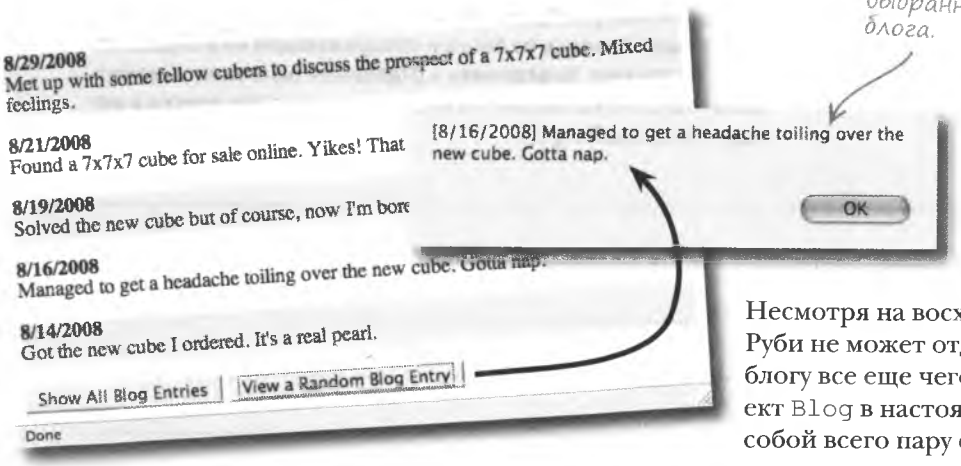
```
function randomBlog() {
  // Выберите случайное число от 0 до числа записей – 1
  var i = Math.floor(Math.random() * blog.length);
  alert("[" + (blog[i].date.getMonth() + 1) + "/" + blog[i].date.getDate() + "/" +
    blog[i].date.getFullYear() + "] " + blog[i].body);
}
```

Отформатируйте результат таким образом, чтобы после текста записи выводилась ее дата в виде ММ/ДД/ГГГГ.

Хотелось бы чего-нибудь еще...

Блог Руби теперь поддерживает функцию поиска случайных сообщений, и она счастлива. Теперь пользователи при просмотре заранее заинтригованы, ведь они не знают, какая из записей будет им показана.

Случайным образом выбранная запись блога.



Несмотря на восхищение новой функцией, Руби не может отделаться от чувства, что блогу все еще чего-то не хватает. Ее объект `Blog` в настоящее время представляет собой всего пару свойств, в основу работы которых положены отдельные функции. Не самая лучшая конструкция, на взгляд Руби...

Объект в поисках действий

Руби обеспокоена. Поведенческая часть объекта совершенно недостаточна и нуждается в серьезной перестройке, в результате которой отдельные задачи начнут решаться при помощи методов. Руби нужны методы, которые определят **поведение** объекта `Blog`!

Возьми в руку карандаш

Изучите код YouCube и обведите код, который, с вашей точки зрения, можно поместить в методы объекта `Blog`; присвойте методам имена.

```
function showBlog(numEntries) {
  // Сортируем записи в обратном хронологическом порядке (последние впереди)
  blog.sort(function(blog1, blog2) { return blog2.date - blog1.date; });

  // Выбираем число записей, чтобы при необходимости показать блог целиком
  if (!numEntries)
    numEntries = blog.length;

  // Показ записей блога
  var i = 0, blogText = "";
  while (i < blog.length && i < numEntries) {
    // Используем для каждой записи серый фон
    if (i % 2 == 0)
      blogText += "<p style='background-color:#EEEEEE'>";
    else
      blogText += "<p>";

    // Генерируем отформатированный HTML-код блога
    blogText += "<strong>" + (blog[i].date.getMonth() + 1) + "/" +
      blog[i].date.getDate() + "/" +
      blog[i].date.getFullYear() + "</strong><br />" +
      blog[i].body + "</p>";

    i++;
  }

  // Располагаем HTML-код блога на странице
  document.getElementById("blog").innerHTML = blogText;
}

function searchBlog() {
  var searchText = document.getElementById("searchtext").value;
  for (var i = 0; i < blog.length; i++) {
    // Проверяем, включает ли запись строку поиска
    if (blog[i].body.toLowerCase().indexOf(searchText.toLowerCase()) != -1) {
      alert("[" + (blog[i].date.getMonth() + 1) + "/" + blog[i].date.getDate() + "/" +
        blog[i].date.getFullYear() + "] " + blog[i].body);
      break;
    }
  }

  // Если строка поиска не обнаружена, сообщаем об этом
  if (i == blog.length)
    alert("Sorry, there are no blog entries containing the search text.");
}

function randomBlog() {
  // Выбираем случайное число от 0 до blog.length - 1
  var i = Math.floor(Math.random() * blog.length);
  alert("[" + (blog[i].date.getMonth() + 1) + "/" + blog[i].date.getDate() + "/" +
    blog[i].date.getFullYear() + "] " + blog[i].body);
}
```

Возьми в руку карандаш

Решение

Вот код, который можно поместить в методы объекта `Blog`.

```
function showBlog(numEntries) {
  // Сортируем записи в обратном хронологическом порядке (последние впереди)
  blog.sort(function(blog1, blog2) { return blog2.date - blog1.date; });

  // Выбираем число записей, чтобы при необходимости показать блог целиком
  if (!numEntries)
    numEntries = blog.length;

  // Показ записей блога
  var i = 0, blogText = "";
  while (i < blog.length && i < numEntries) {
    // Используем для каждой записи серый фон
    if (i % 2 == 0)
      blogText += "<p style='background-color:#EEEEEE'>";
    else
      blogText += "<p>";

    // Генерируем отформатированный HTML-код блога
    blogText += "<strong>" + (blog[i].date.getMonth() + 1) + "/" +
      blog[i].date.getDate() + "/" +
      blog[i].date.getFullYear() + "</strong><br />" +
      blog[i].body + "</p>";

    i++;
  }

  // Располагаем HTML-код блога на странице
  document.getElementById("blog").innerHTML = blogText;
}

function searchBlog() {
  var searchText = document.getElementById("searchtext").value;
  for (var i = 0; i < blog.length; i++) {
    // Проверяем, включает ли запись строку поиска
    if (blog[i].body.toLowerCase().indexOf(searchText.toLowerCase()) != -1) {
      alert("[" + (blog[i].date.getMonth() + 1) + "/" + blog[i].date.getDate() + "/" +
        blog[i].date.getFullYear() + "] " + blog[i].body);
      break;
    }
  }

  // Если строка поиска не обнаружена, сообщаем об этом
  if (i == blog.length)
    alert("Sorry, there are no blog entries containing the search text.");
}

function randomBlog() {
  // Выбираем случайное число от 0 до blog.length - 1
  var i = Math.floor(Math.random() * blog.length);
  alert("[" + (blog[i].date.getMonth() + 1) + "/" + blog[i].date.getDate() + "/" +
    blog[i].date.getFullYear() + "] " + blog[i].body);
}
```

Метод Blog.toHTML() преобразует запись блога в отформатированный HTML-фрагмент.

Метод Blog.toHTML() снимет большую нагрузку с остального кода, который должен отображать хорошо отформатированный блог.

Небольшой по размеру, но полезный метод Blog.containsText() будет отвечать за поиск текста в блоге.

Метод Blog.toString() будет преобразовывать записи блога в строку. Его имеет смысл использовать в ситуациях, когда дата в квадратных скобках выводится после текста записи.

Часто Задаваемые Вопросы

В: Как понять, какой код можно превратить в метод?

О: Для начала следует понять, для чего будет предназначен этот метод и какие действия он будет совершать над данными объекта. В некоторой степени определение методов объекта зависит от того, что именно делает или должен будет делать объект. После чего мы даем объекту возможность выполнять эти действия.

Например, объекту `Blog` целесообразно иметь способность превращать себя в строку или в отформатированный HTML-код, ведь эти действия требуют доступа к внутренним данным блога. Аналогично, поиск текста в записях — это внутреннее действие объекта `Blog`, а значит, имеет смысл превратить его в метод.

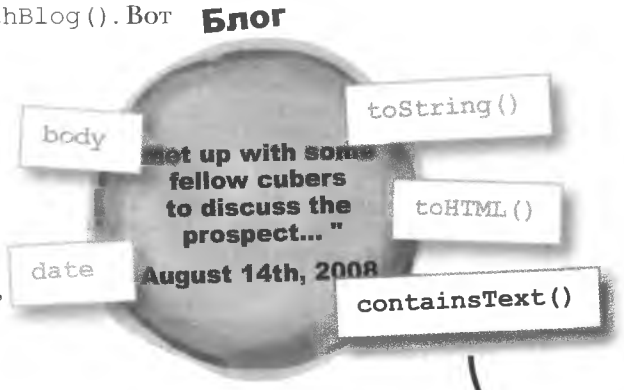
В: А как насчет примера действия, которое объект `Blog` не сможет выполнить?

О: За пределы возможностей объекта `Blog` выходят такие действия, как поиск или отображение списка записей. Ведь этот объект представляет единственную запись. Именно поэтому массив `blog` состоит из набора **отдельных** объектов `Blog`. И каждый такой объект никак не должен касаться действий, производимых со всем набором.

Превращение функции в метод

Теперь, когда мы решили, какие именно фрагменты кода блога YouCube достойны превращения в методы объекта `Blog`, посмотрим на саму процедуру преобразования. В качестве примера возьмем метод `containsText()`, отвечающий за поиск в теле записи указанного пользователем набора символов. Перемещение кода в метод проводит операцию непосредственно над свойством `body` объекта `Blog`, в отличие от локальной переменной в функции `searchBlog()`. Вот пошаговое описание процесса:

1. Объявим метод, составим список аргументов, например, для метода `containsText()` это строка поиска.
2. Поместим код в новый метод.
3. Заставим код использовать свойства объекта, в нашем случае свойство `this.body`.



Возьми в руку карандаш

Напишите код для метода `containsText()`, создаваемого в конструкторе `Blog()` путем назначения литерала функции ссылке `this.containsText`.

.....

.....

.....

Возьми в руку карандаш

Решение

Вот код для метода `containsText()`, создаваемого в конструкторе `Blog()` путем назначения литерала функции ссылке `this.containsText`.

Метод создается назначением литерала функции ссылки на него.

```
this.containsText = function(text) {
  return (this.body.toLowerCase().indexOf(text.toLowerCase()) != -1);
};
```

Ключевое this используется для создания метода тем же способом, которым с его помощью создавались свойства.

Код метода имеет непосредственный доступ к свойствам объекта при помощи ключевого слова this.

Восхитительный новый объект blog

Эй, теперь я в хорошей форме!

Еще два метода соединились с методом `containsText()` в новой версии объекта `Blog`, который теперь обладает и свойствами и поведением.

```
function Blog(body, date) {
  // назначаем свойства
  this.body = body;
  this.date = date;

  // Return a string representation of the blog entry
  this.toString = function() {
    return "[" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
      this.date.getFullYear() + "] " + this.body;
  };

  // Возвращаем форматированное HTML-представление записи
  this.toHTML = function(highlight) {
    // Используем для выделения серый фон
    var blogHTML = "";
    blogHTML += highlight ? "<p style='background-color:#EEEEEE'>" : "<p>";

    // Генерируем отформатированный HTML-код блога
    blogHTML += "<strong>" + (this.date.getMonth() + 1) + "/" +
      this.date.getDate() + "/" + this.date.getFullYear() + "</strong><br />" +
      this.body + "</p>";
    return blogHTML;
  };

  // Проверяем, содержит ли блог строку текста
  this.containsText = function(text) {
    return ((this.body.toLowerCase().indexOf(text.toLowerCase()) != -1);
  };
}
```

Создание и инициализация свойств.

Метод toString() возвращает запись блога, отформатированную в виде текстовой строки.

Метод toHTML() возвращает запись блога как отформатированный HTML-код.

Метод containsText() возвращает значение true, если текст содержит поисковую строку.

Что дают объекты блогу YouCube?

Новый объект Blog упрощает сценарий блога YouCube.

Думаю, вы поняли преимущества объектно-ориентированного программирования еще до появления новой версии объекта `Blog` (она доступна по адресу *http://www.headfirstlabs.com/books/hfjs/*). Теперь, когда решение ряда важных задач отдано на откуп методам объекта `Blog`, код сценария стал намного проще.

```
// Show the list of blog entries
function showBlog(numEntries) {
  // Сортируем записи в обратном хронологическом порядке (последние впереди)
  blog.sort(function(blog1, blog2) { return blog2.date - blog1.date; });

  // Выбираем число записей, чтобы при необходимости показать блог целиком
  if (!numEntries)
    numEntries = blog.length;

  // Цикл пока записей блога
  var i = 0, blogListHTML = "";
  while (i < blog.length && i < numEntries) {
    blogListHTML += blog[i].toHTML(i % 2 == 0);
    i++;
  }

  // Располагаем HTML-код блога на странице
  document.getElementById("blog").innerHTML = blogListHTML;
}

// Ищем фрагмент текста в записях блога
function searchBlog() {
  var searchText = document.getElementById("searchtext").value;
  for (var i = 0; i < blog.length; i++) {
    // Проверяем, включает ли запись строку поиска
    if (blog[i].containsText(searchText)) {
      alert(blog[i]);
      break;
    }
  }

  // Если строка поиска не обнаружена, сообщаем об этом
  if (i == blog.length)
    alert("Sorry, there are no blog entries containing the search text.");
}

// Отображаем случайно выбранную запись
function randomBlog() {
  // Выбираем случайное число от 0 до blog.length - 1
  var i = Math.floor(Math.random() * blog.length);
  alert(blog[i]);
}
```

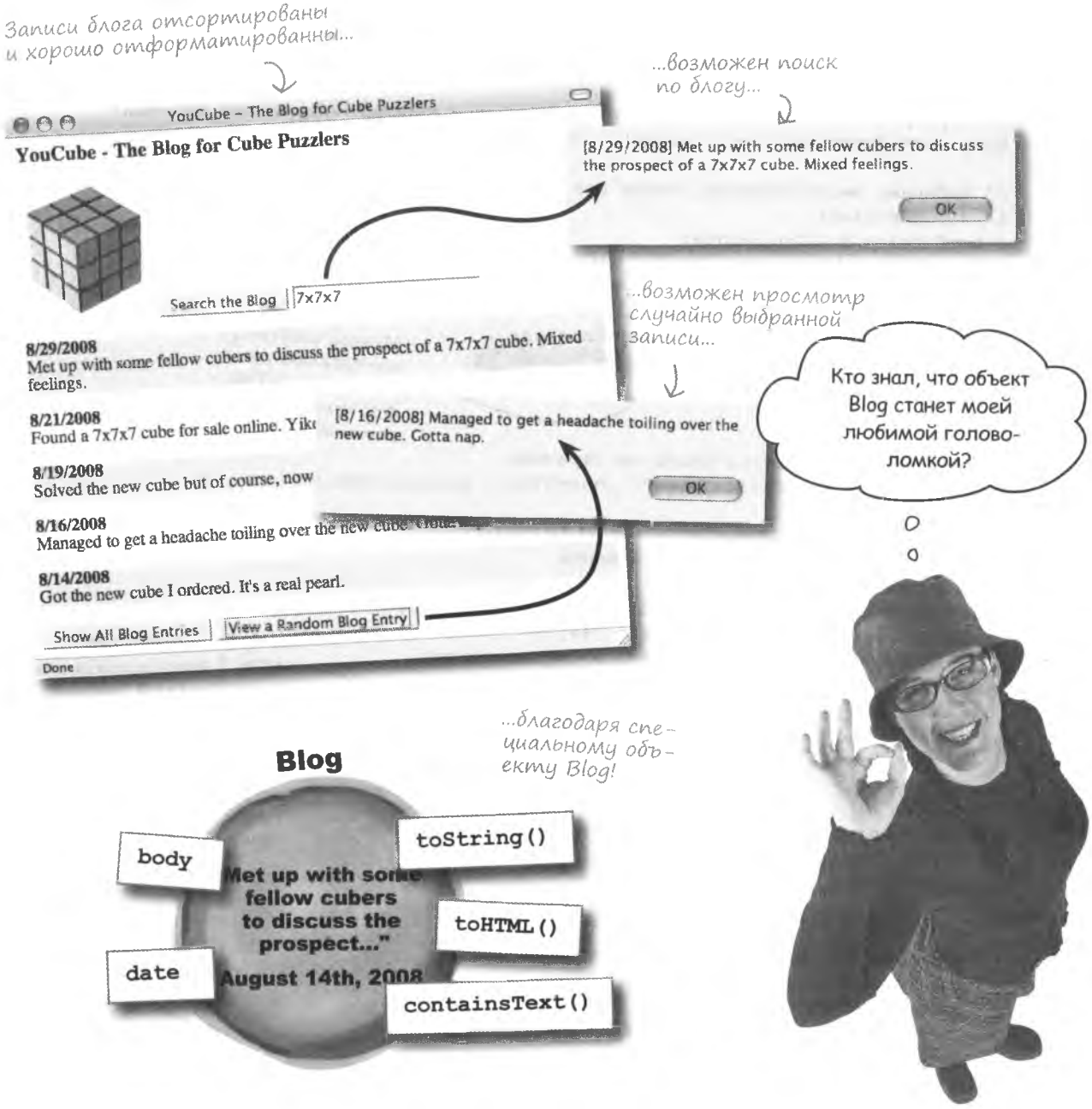
Метод toHTML() полностью отвечает за HTML-форматирование записей блога.

Метод containsText() ищет в записи указанную пользователем строку.

Метод toString() автоматически вызывается в случаях, когда запись используется там, где нужна строка.

YouCube 3.0!

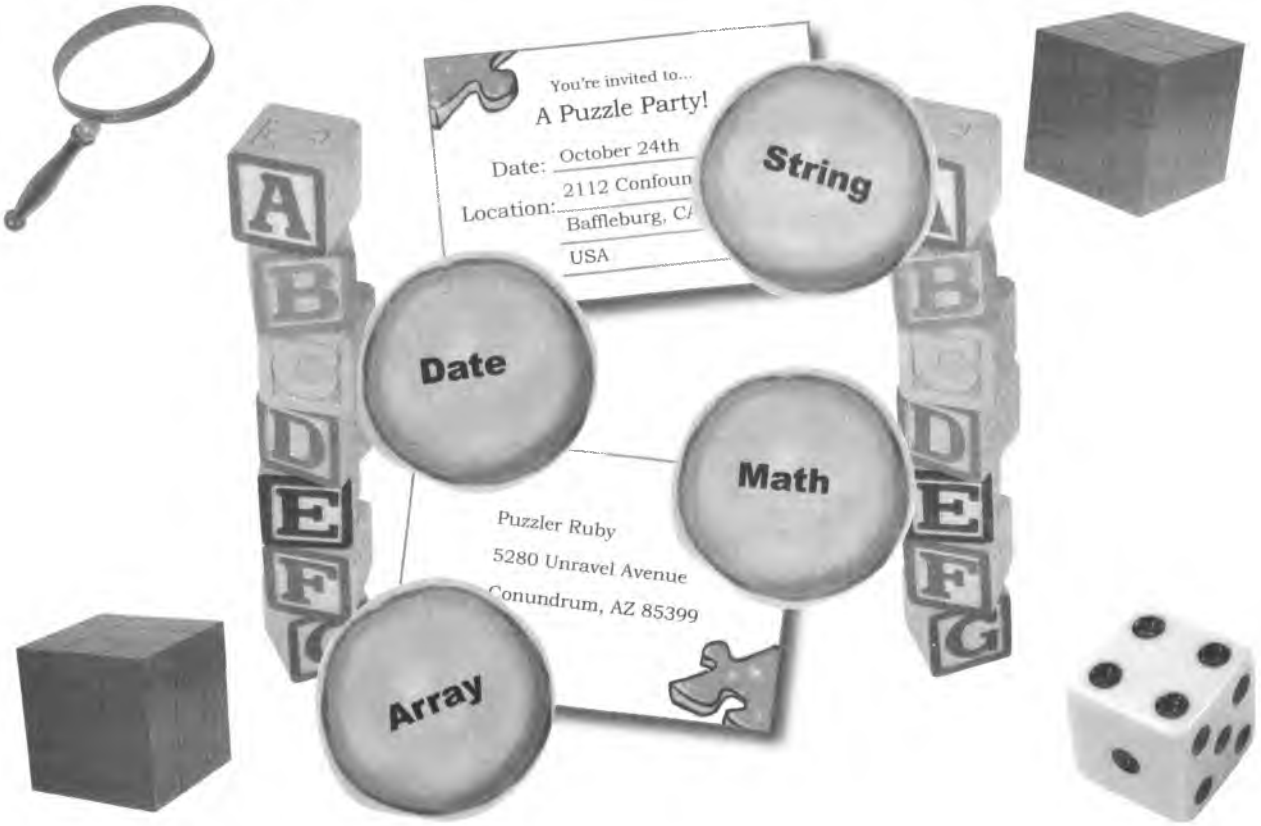
Руби официально объявила, что она довольна версией блога YouCube 3.0 и предпочитает вернуться к головоломкам и к подготовке вечеринки для таких же, как она, фанатов...



Вкладка

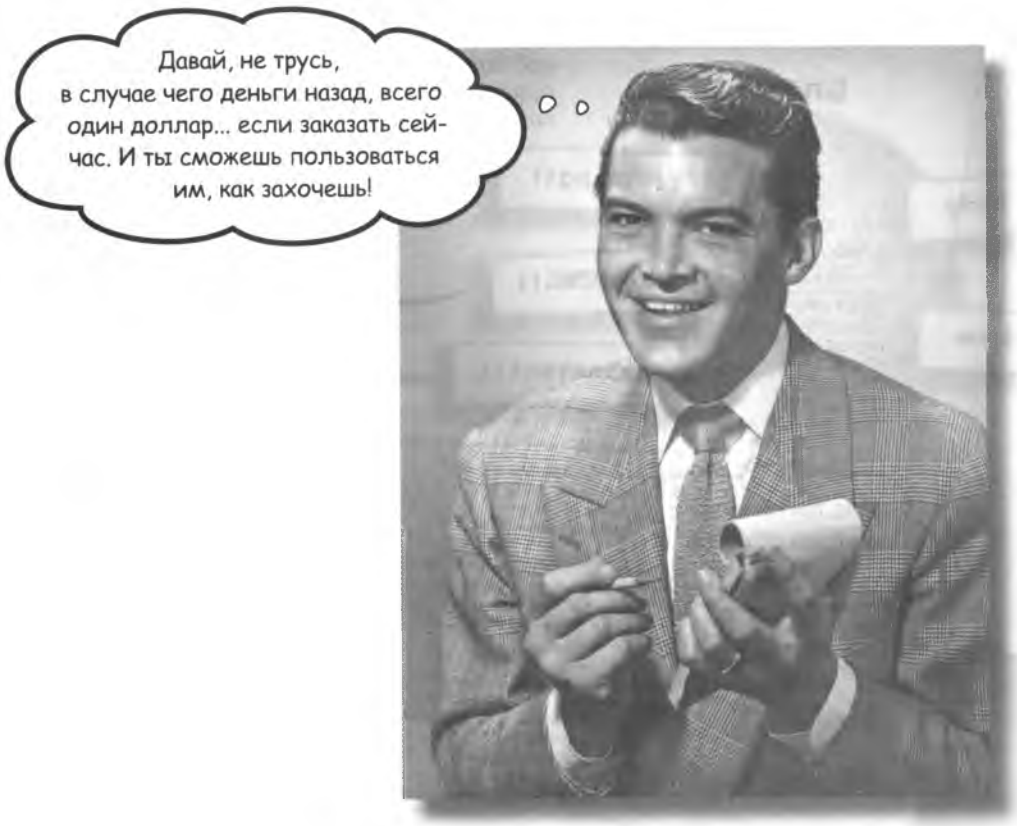
Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Что могут сделать с данными объекты JavaScript?



Ищите, что хотите, но вряд ли вы найдете что-то лучше, чем объекты JavaScript для сортировки и анализа данных. Они умеют даже генерить случайные числа.

Работа со специальными объектами



Если бы все было так легко, мы бы, конечно, так и сделали. JavaScript не гарантирует возврат денег, но вы действительно можете делать с ним все, что захотите. Специальные объекты — это эквивалент тройного эспрессо с сахаром и корицей. Вот такая специальная чашка кофе! Точно так же в специальных объектах вы можете смешивать код, добиваясь именно того результата, который вам нужен, и пользуясь преимуществами свойств и методов. И в конце получается объектно-ориентированный код, расширяющий язык JavaScript... только для вас!

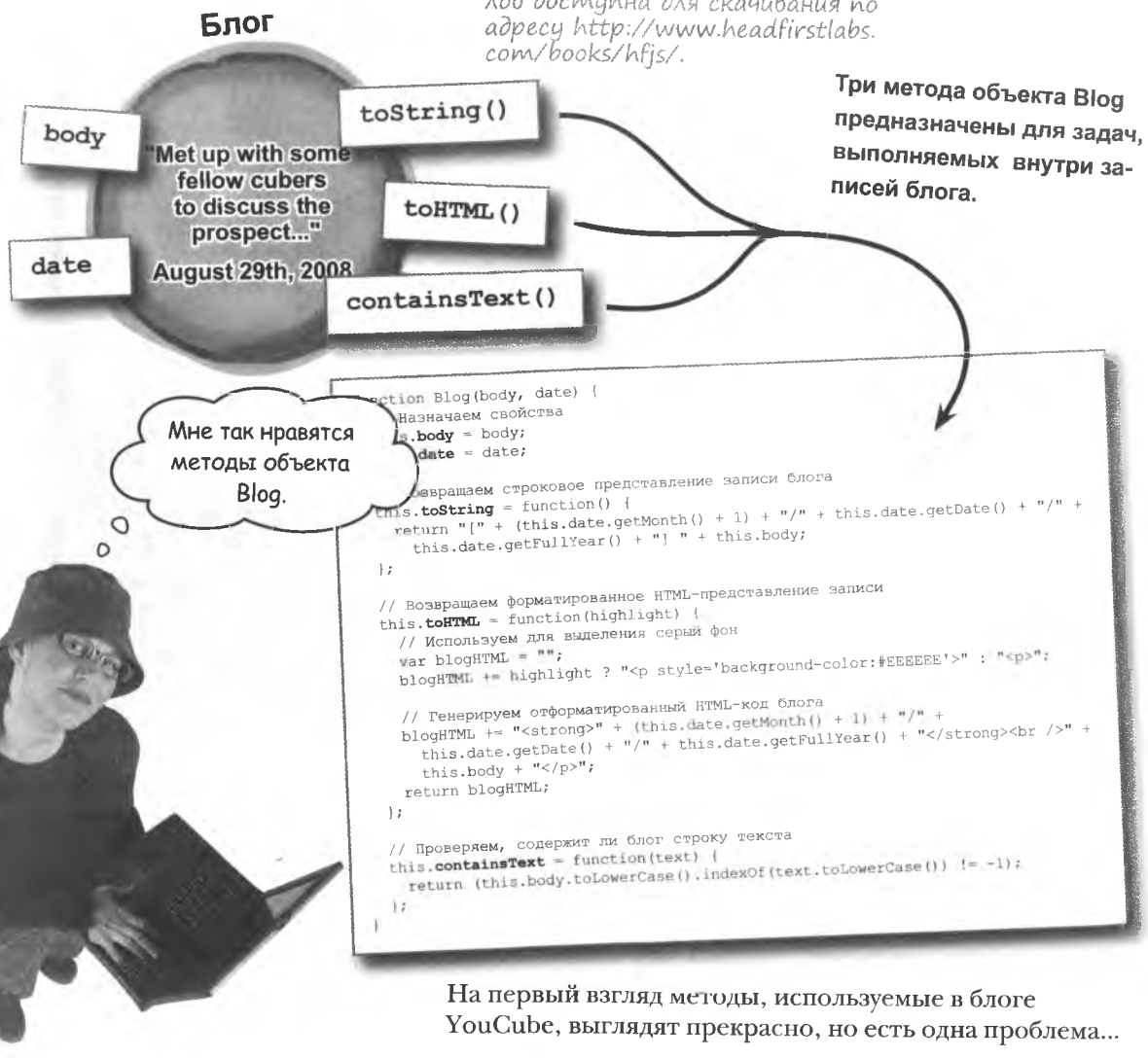
Снова о методах блога YouCube

Несмотря на то что в прошлой главе Руби проделала большую работу по созданию объекта `Blog` для управления своим блогом YouCube, несколько ключевых для объектно-ориентированного программирования возможностей остались за кадром. Она не уделила должного внимания эффективности объекта `Blog`, его структуре и, следовательно, возможности редактировать его в будущем.

Блог YouCube работает, но его нельзя причислить к самым удачным образцам объектно-ориентированного программирования.

Остановились мы, как вы помните, на создании трех методов для выполнения связанных с блогом задач.

Последняя версия данных файлов доступна для скачивания по адресу http://www.headfirstlabs.com/books/hfjs/.



Три метода объекта Blog предназначены для задач, выполняемых внутри записей блога.

На первый взгляд методы, используемые в блоге YouCube, выглядят прекрасно, но есть одна проблема...

Перегрузка методов

Точно так же, как и свойства, методы объекта `Blog` созданы внутри конструктора при помощи ключевого слова `this`. При таком подходе для каждого вновь созданного объекта `Blog` создаются и три копии методов. Так что, например, для блога из шести записей мы получим шесть их копий.

Объект Blog ненамеренно создает больше методов, чем это необходимо, что крайне неэффективно.

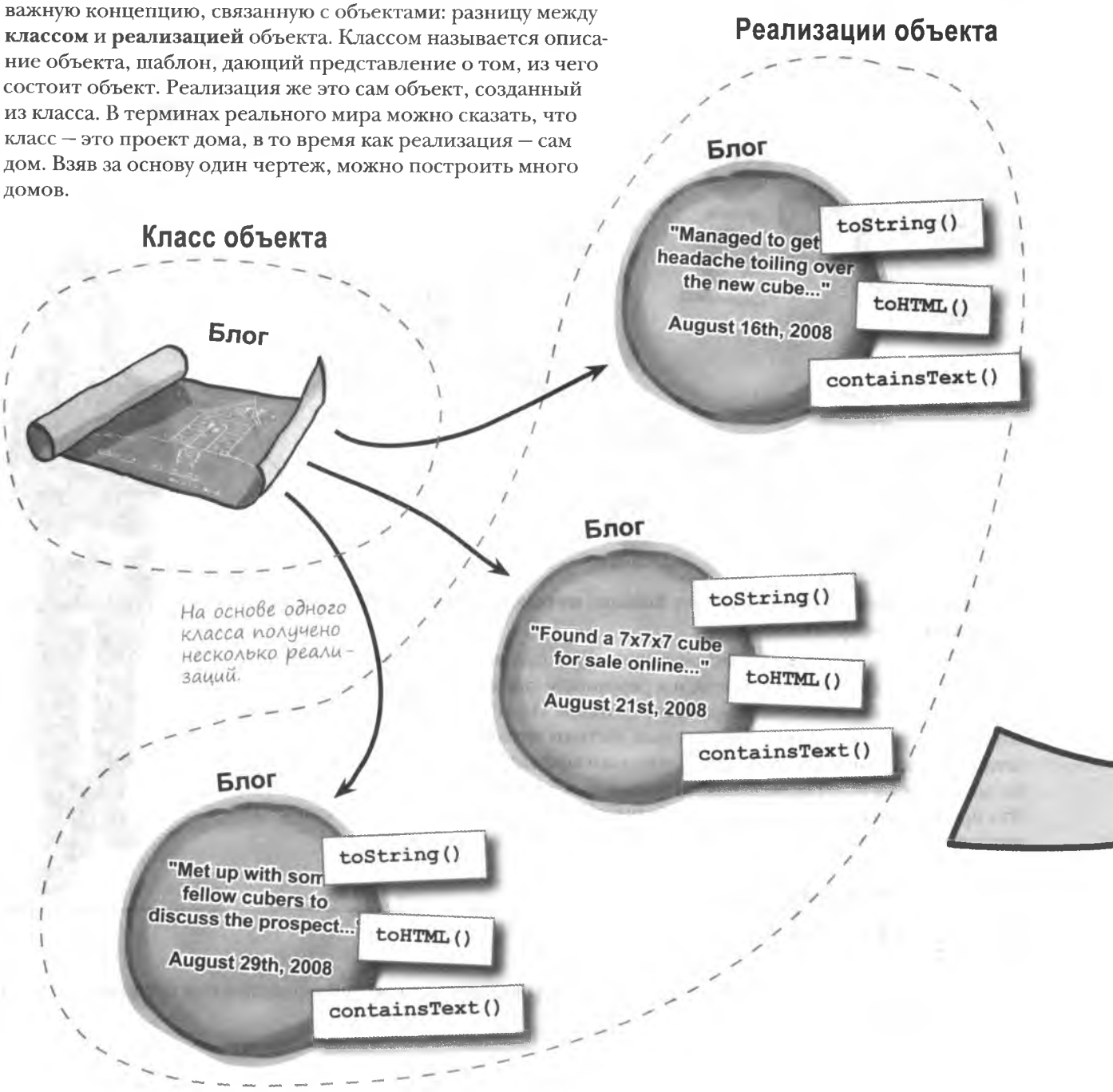
Никуда от этого не деться, конструктор `Blog()` создает по три метода при каждом создании нового объекта, и в результате каждый объект `Blog` получает собственную копию каждого из методов. Но если свойства каждого объекта сохраняют уникальные данные, методы вполне можно отдать объектам в совместное пользование. Насколько эффективней было бы, если бы все объекты `Blog` использовали одну и ту же копию каждого из методов. Это предотвратило бы излишнее увеличение объема сценария из-за появления избыточных методов по мере неизбежного роста числа записей.

МОЗГОВОЙ ШТУРМ

Что нужно сделать с объектом `Blog`, чтобы код методов не дублировался при создании каждой следующей копии?

Классы и реализации

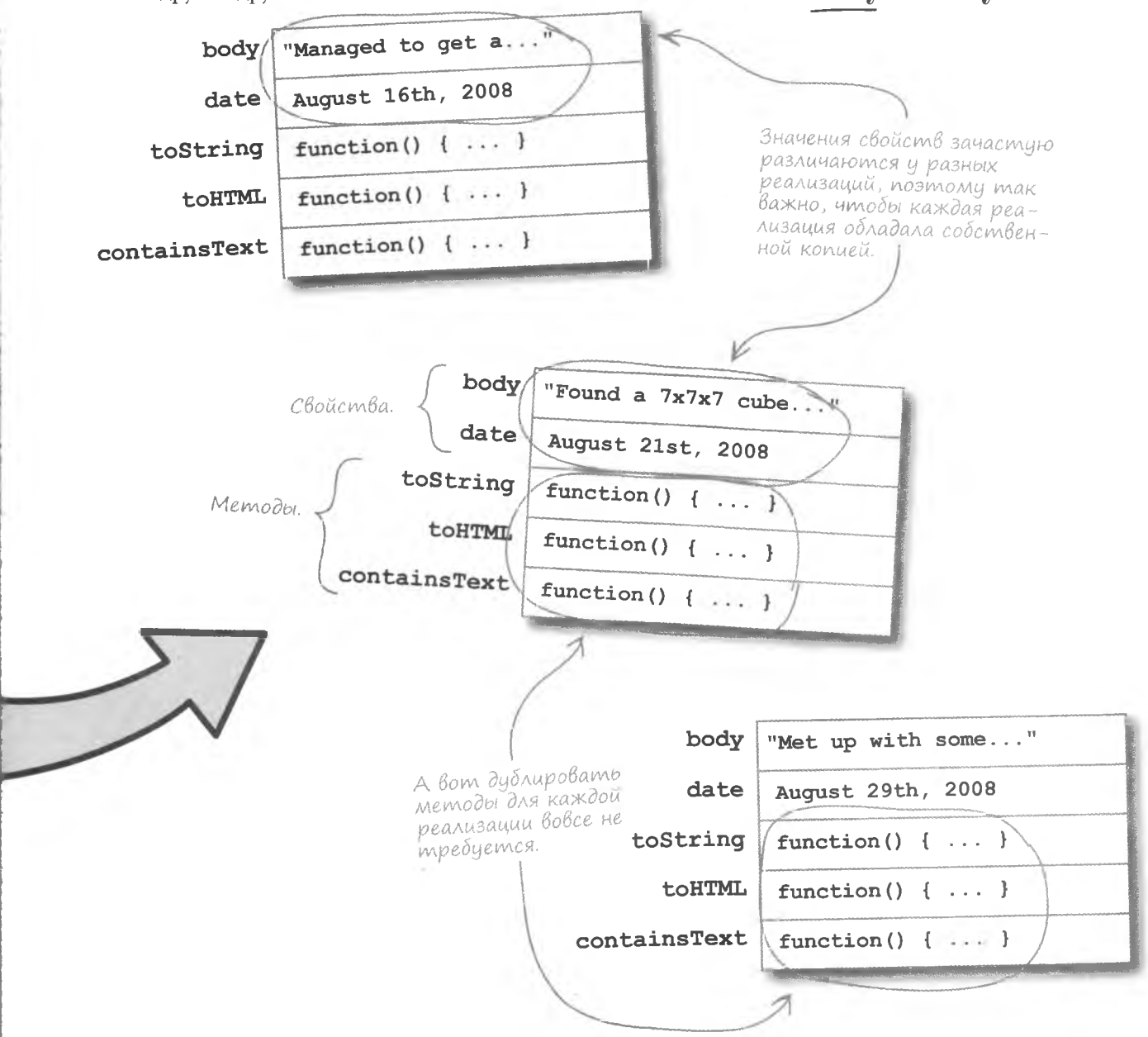
Проблема с дублированием методов затрагивает крайне важную концепцию, связанную с объектами: разницу между **классом** и **реализацией** объекта. Классом называется описание объекта, шаблон, дающий представление о том, из чего состоит объект. Реализация же это сам объект, созданный из класса. В терминах реального мира можно сказать, что класс — это проект дома, в то время как реализация — сам дом. Взяв за основу один чертеж, можно построить много домов.



Реализации

Класс описывает свойства и методы объекта, в то время как реализация присваивает свойствам реальные данные и заставляет их действовать. Все реализации имеют собственные копии свойств, что позволяет им уникальным образом отличаться друг от друга.

Класс объекта — это шаблон, в то время как реализация — созданный по этому шаблону объект.



Ключевое слово this

Ключевое слово this задает свойства и методы реализаций.

До сих пор мы имели дело в основном со **свойствами реализаций**. При этом каждая реализация обладала их собственной копией. Такие свойства легко идентифицировать, так как именно они задаются в конструкторе при помощи ключевого слова `this`.

```
function Blog(body, date) {
  this.body = body;
  this.date = date ;
  ...
}
```

Это свойства реализации, так как на них ссылаются при помощи ключевого слова this.

Существуют и методы реализаций, но с ними все гораздо сложнее, так как они могут находиться в собственности как реализации, так и класса. До этого момента мы имели дело с методами, заданными при помощи ключевого слова `this`, то есть с методами реализаций. Именно поэтому методы дублировались от экземпляра к экземпляру.

```
function Blog(body, date) {
  ...
  this.toString = function() {
    ...
  }
  this.toHTML = function() {
    ...
  }
  this.containsText = function() {
    ...
  }
}
```

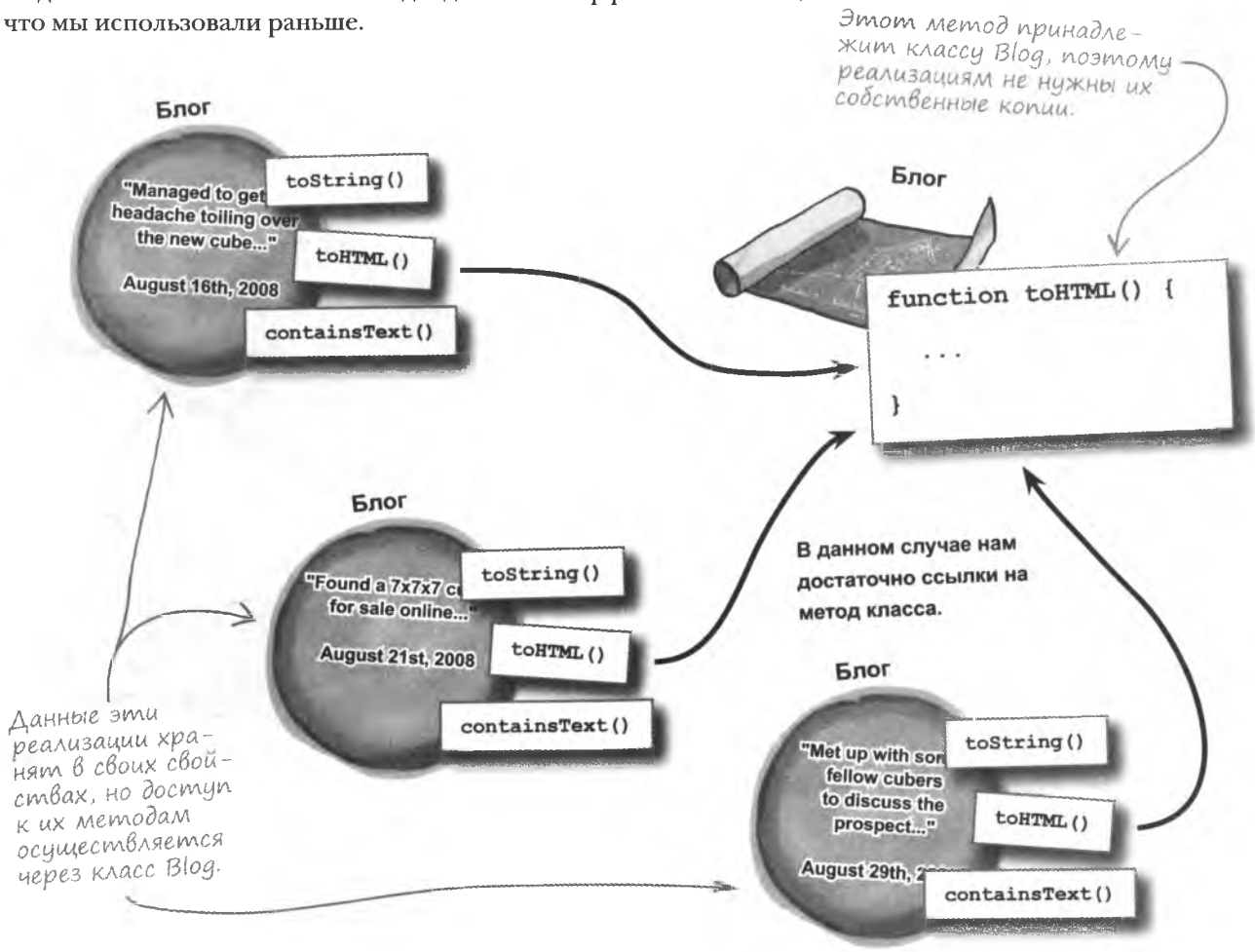
Все реализации объекта Blog получают собственные копии этих методов.

Все это методы реализаций, так как они задаются в конструкторе ключевым словом this.

Хорошей новостью является то, что специальные объекты вовсе не обязаны впустую создавать дублирующийся код методов с появлением каждой новой реализации. Достаточно создать метод таким образом, чтобы все реализации смогли использовать одну и ту же копию его кода.

Методы классов

Существует и другой вид методов, принадлежащий собственно классам, что означает возможность доступа **всех** реализаций к единственной копии. Такой подход намного эффективней того, что мы использовали раньше.



Если метод принадлежит классу, к нему имеют доступ все реализации, и поэтому их собственные копии им уже не требуются. Такой подход намного более эффективен, особенно если учесть, сколько лишних методов в итоге появляется в приложениях, постоянно создающих новые реализации объектов. В блоге YouCube три метода (`toString()`, `toHTML()` и `containsText()`) дублировались бы вместе с каждой новой записью.

Теперь осталось понять, как сделать так, чтобы метод стал принадлежать классу...

Хранение метода в классе позволяет всем реализациям пользоваться одной копией.

Прототипы

Появление того, что мы назвали классами, стало возможным в JavaScript благодаря скрытому объекту `prototype`, существующему как свойство любого объекта. Именно он позволяет задавать свойства и методы, **принадлежащие классам**. Давайте посмотрим, как это делается.

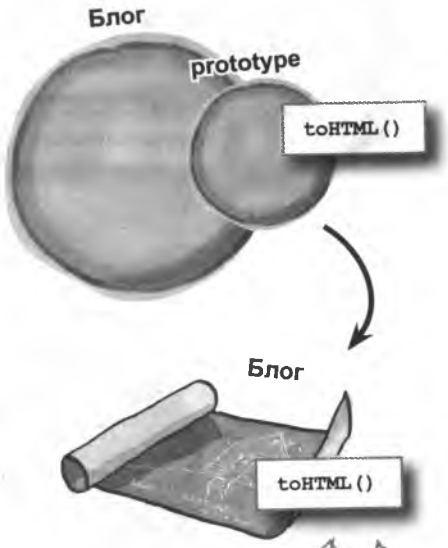
```
Blog.prototype.toHTML = function() {
    ...
}
```

Телом функции является литерал.

Имя функции.

Объект prototype доступен как свойство класса.

Имя класса.



В данном примере метод `toHTML()` добавляется к классу `Blog`, а не к какой-либо реализации этого класса. Неважно, сколько объектов `Blog` мы создадим, копия метода `toHTML()` все равно будет только одна.

Так как метод `toHTML()` принадлежит классу `Blog`, в этом же классе расположен и код, запускаемый при вызове метода. Но технически это все равно метод реализации, так как он может быть вызван на уровне объекта и имеет доступ к свойствам этого объекта.


```
var blogEntry1 = new Blog("Not much going on.", ...);

blogEntry1.toHTML();
```

Код, запускаемый при вызове метода toHTML(), работает внутри класса.

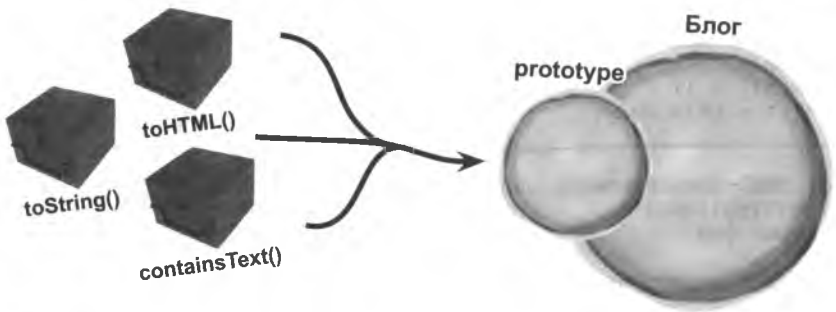
При создании следующего объекта `Blog` и вызове для него метода `toHTML()` запускается тот же самый код внутри класса. Видите, как выгодно, – сохранил один раз, а используешь, сколько нужно!

```
var blogEntry2 = new Blog("Still just hanging around.", ...);

blogEntry2.toHTML();
```

Остальные объекты используют тот же самый метод класса.

Классы, прототипы и YouCube

Руби немного ошеломлена сведениями о классах и прототипах, но она чувствует, что блог YouCube можно выгодно модернизировать, связав методы объекта `Blog` с объектом `prototype`.



Возьми в руку карандаш

Для хранения методов, которые теперь принадлежат классу, в новой версии кода используется объект `prototype`. Вставьте ваши примечания и объясните, что именно происходит.

```
function Blog(body, date) {
  // Назначаем свойства
  this.body = body;
  this.date = date;
}

// Возвращаем строковое представление записи блога
Blog.prototype.toString = function() {
  return "[" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
    this.date.getFullYear() + "] " + this.body;
};

// Возвращаем отформатированное HTML-представление записи блога
Blog.prototype.toHTML = function(highlight) {
  // Используем для выделения серый фон
  var blogHTML = "";
  blogHTML += highlight ? "<p style='background-color:#EEEEEE'>" : "<p>";

  // Генерируем отформатированный HTML-код блога
  blogHTML += "<strong>" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
    this.date.getFullYear() + "</strong><br />" + this.body + "</p>";
  return blogHTML;
};

// Проверяем, содержит ли блог строку текста
Blog.prototype.containsText = function(text) {
  return (this.body.toLowerCase().indexOf(text.toLowerCase()) != -1);
};
```

Возьми в руку карандаш

Решение

Итак, для хранения методов, которые теперь принадлежат классу, в новой версии кода используется объект `prototype`.

```
function Blog(body, date) {
  // Назначаем свойства
  this.body = body;
  this.date = date;
}

// Возвращаем строковое представление записи блога
Blog.prototype.toString = function() {
  return "[" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
    this.date.getFullYear() + "] " + this.body;
};

// Возвращаем отформатированное HTML-представление записи блога
Blog.prototype.toHTML = function(highlight) {
  // Используем для выделения серый фон
  var blogHTML = "";
  blogHTML += highlight ? "<p style='background-color:#EEEEEE'>" : "<p>";

  // Генерируем отформатированный HTML-код блога
  blogHTML += "<strong>" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
    this.date.getFullYear() + "</strong><br />" + this.body + "</p>";
  return blogHTML;
};

// Проверяем, содержит ли блог строку текста
Blog.prototype.containsText = function(text) {
  return (this.body.toLowerCase().indexOf(text.toLowerCase()) != -1);
};
```

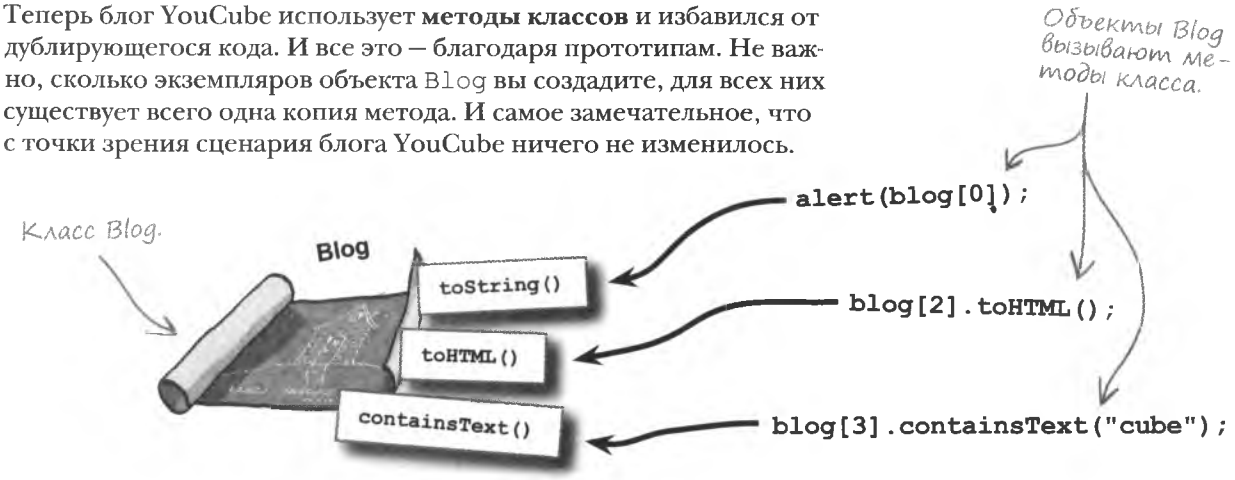
Теперь в задачу конструктора входит только создание и инициализация свойств.

Так как методы назначаются не объекту Blog, назначение происходит вне конструктора.

Каждый метод назначен объекту prototype вместо того, чтобы воспользоваться ключевым словом this в конструкторе Blog().

Более эффективный YouCube

Теперь блог YouCube использует **методы классов** и избавился от дублирующегося кода. И все это — благодаря прототипам. Не важно, сколько экземпляров объекта `Blog` вы создадите, для всех них существует всего одна копия метода. И самое замечательное, что с точки зрения сценария блога YouCube ничего не изменилось.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Класс — это **описание** объекта, в то время как реализация — это **сам** объект, созданный по этому описанию.
- Класс состоит из свойств и методов объекта, в то время как сами объекты помещают в свойства данные и дают работу методам.
- Ключевое слово `this` используется для доступа к объекту из **его собственного кода**.
- Объект `prototype` позволяет сохранять методы в **класс**, предотвращая ненужное дублирование кода.

Часто Задаваемые Вопросы

В: Я до сих пор не совсем понимаю, зачем нужны классы и реализации.

О: Классы введены, чтобы облегчить создание и повторное использование объектов. Единичные объекты можно создавать в виде литералов, но при этом много ресурсов тратится зря. Ведь этот процесс сопровождается созданием дублирующегося кода. Вы, как архитектор, который требует перед постройкой очередного типового дома заново чертить ему план.

А ведь можно создать шаблон и уже по нему получить нужное количество объектов, сэкономив усилия. Здесь нам на помощь приходят классы — мы используем один класс для создания нужного количества объектов.

В: Хорошо, классы помогают создать копии объектов. А зачем при этом нужны ключевое слово `this` и объект `prototype`?

О: Ключевое слово `this` дает доступ к объекту из его собственных методов. Чаще всего оно используется для доступа к свойствам. Скажем, для доступа к свойству `x` из метода нужно написать `this.x`. Если написать просто `x`, сценарий не будет знать, где именно искать данное свойство объекта; более того, он может принять `x` за переменную. Именно поэтому в конструкторе при создании и инициализации свойств требуется ключевое слово `this`.
Совершенно другая природа объекта `prototype`. Он обеспечивает нас механизмом для создания классов. В отличие от таких языков, как C++ и Java, в JavaScript классы не поддерживаются. Они эмулируются при помощи прототипов. Вы получаете похожий конечный результат, но с применением объекта `prototype`, который относится к «скрытым» свойствам всех объектов JavaScript. Сохранив свойство или метод в объекте `prototype`, вы, тем самым, делаете его принадлежащим к классу, а не к реализации объекта.

В: А каким образом в картину классов вписываются конструкторы?

О: Как вы уже знаете, за создание объектов в JavaScript отвечают именно конструкторы. Вместе с прототипами они представляют два основных фрагмента головоломки с классами в JavaScript. Конструкторы задают все параметры объектов, в то время как прототипы делают то же самое на уровне классов. Оба механизма работают в паре, давая вам возможность делать потрясающие вещи, ведь некоторые члены сценария требуется поместить на уровень объектов, в то время как для других нужен уровень классов. Мы подробнее поговорим об этом позднее в этой главе.

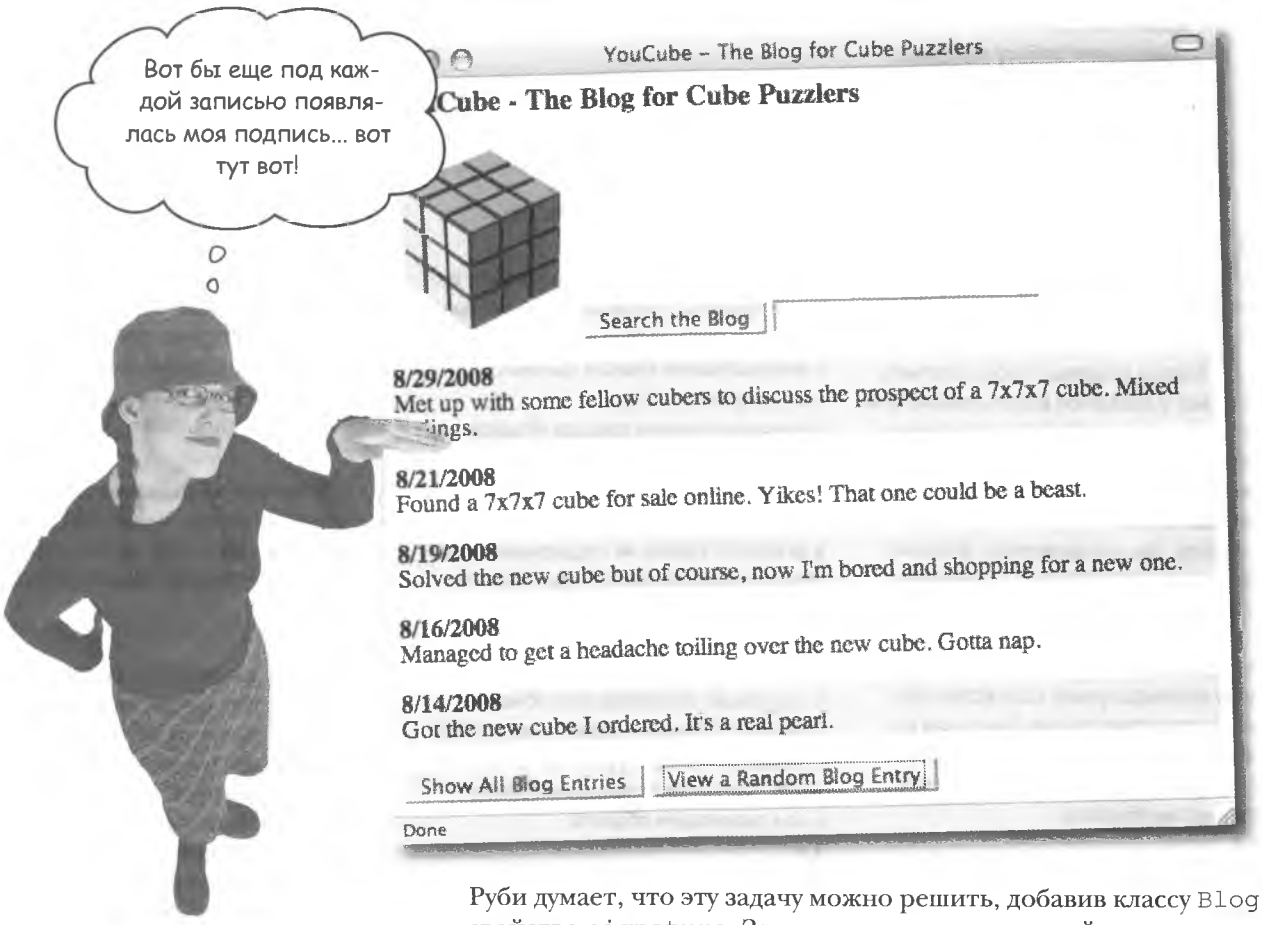
В: Я немного не понимаю стандарты именования. Иногда имена объектов начинаются с большой буквы, иногда используется нижний стильВерблюда. Почему так?

О: Имена классов пишутся с большой буквы, в то время как для записи имен объектов используется стильВерблюда. Ведь реализация объекта является не более чем переменной, а имена переменных у нас записываются именно так. Некоторая непоследовательность связана с весьма вольным поначалу использованием термина «объект». Если быть точным, то имена классов, например `Blog`, надо писать с большой буквы, а имена объектов, например `blogEntry` или `blog[0]`,— стилемВерблюда.

Вспомните стандартные объекты, с которыми мы работали. Текущую дату/время можно было сохранить в переменную (реализацию) с именем `now`, которая создавалась из объекта `Date` (класса).

Подпись в блоге

Руби ищет информацию об увеличении эффективности и улучшении структуры, которых можно достичь средствами объектно-ориентированного программирования. Впрочем, ее интересует не только усовершенствование внутреннего кода, еще она решила добавить к блогу новую функцию.



Руби думает, что эту задачу можно решить, добавив классу `Blog` свойство `signature`. Затем параметры этого свойства задаются в конструкторе и начинают отображаться в каждой записи... и проблема решена!

МОЗГОВОЙ ШТУРМ

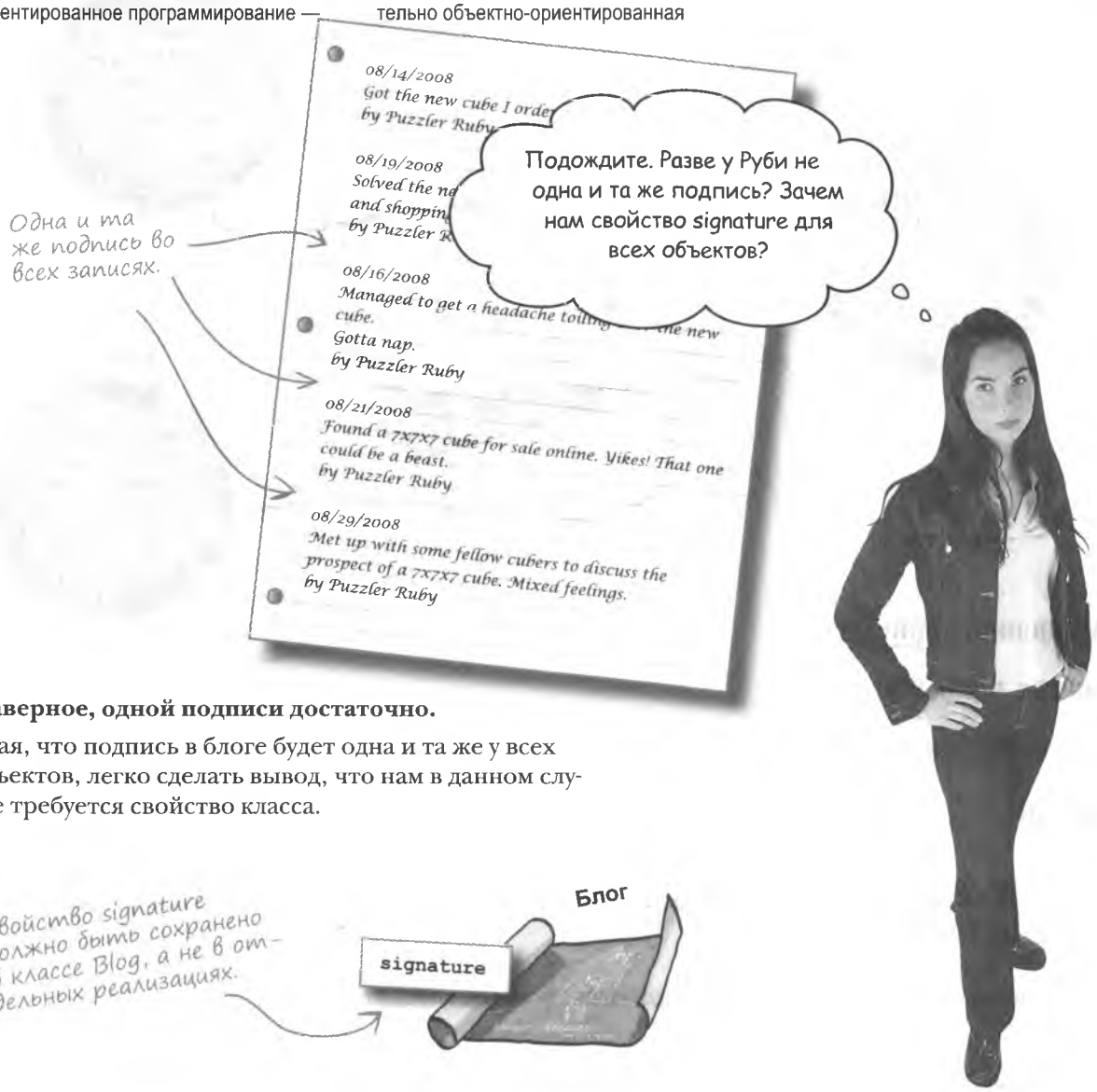
А может быть, подпись следует сделать свойством реализации? Как вы думаете, почему это не самая хорошая идея?

Часто Задаваемые Вопросы

В: Что означает постоянно появляющееся словосочетание «объектно-ориентированный»?

О: Термин **объектно-ориентированный** часто используется в программистских кругах, и порой им обозначают разные вещи. В общем случае объектно-ориентированное программирование — это построение программ из объектов. Вот, скажем, свойство `date`, используемое в записях блога, получено из объекта `Date`.

Большинство программистов связывают ООП с интенсивным использованием объектов в программах. В теории действительно объектно-ориентированная программа может быть разбита на набор взаимодействующих друг с другом объектов. Некоторые пуристы даже считают, что JavaScript не относится к языкам ООП. Как сторонники, так и противники такой точки зрения имеют свои весомые аргументы, но пока ни один из них не взял верх.

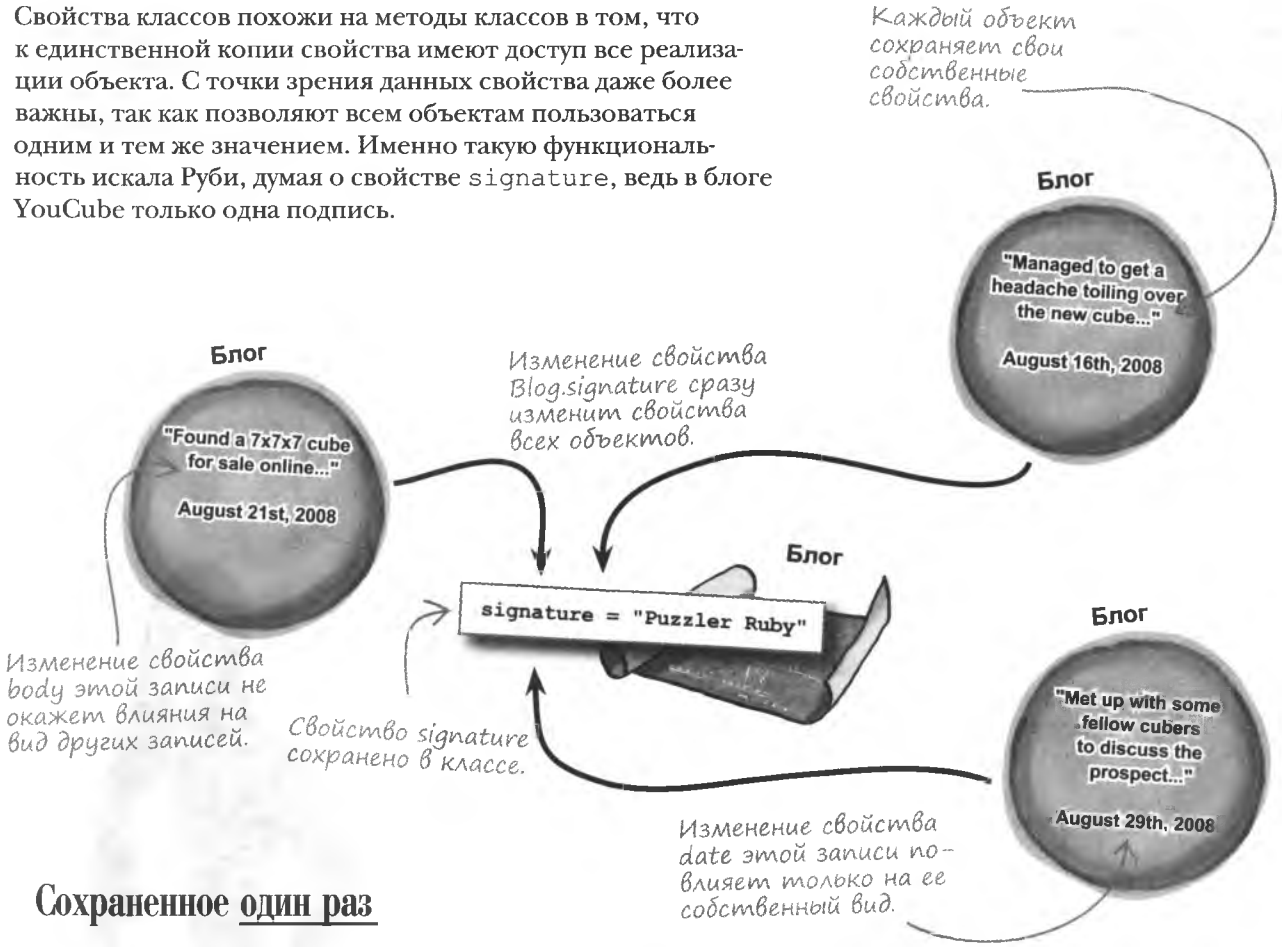


Наверное, одной подписи достаточно.

Зная, что подпись в блоге будет одна и та же у всех объектов, легко сделать вывод, что нам в данном случае требуется свойство класса.

Свойства общего доступа

Свойства классов похожи на методы классов в том, что к единственной копии свойства имеют доступ все реализации объекта. С точки зрения данных свойства даже более важны, так как позволяют всем объектам пользоваться одним и тем же значением. Именно такую функциональность искала Руби, думая о свойстве `signature`, ведь в блоге YouCube только одна подпись.



Сохраненное один раз свойство класса доступно затем всем реализациям объекта.

Несмотря на то что свойство `signature` сохранено в классе `Blog`, доступ к нему имеют все реализации объекта, которым требуется подпись автора блога.

МОЗГОВОЙ ШТУРМ

Как вы думаете, каким способом создаются свойства класса?

Создание свойств класса

За всеми этими разговорами о том, где сохраняется свойство класса и какое значение оно имеет, процесс его создания оказывается на удивление обыденным. Это всего лишь одна строчка кода:

```
Blog.prototype.signature = "Puzzler Ruby";
```

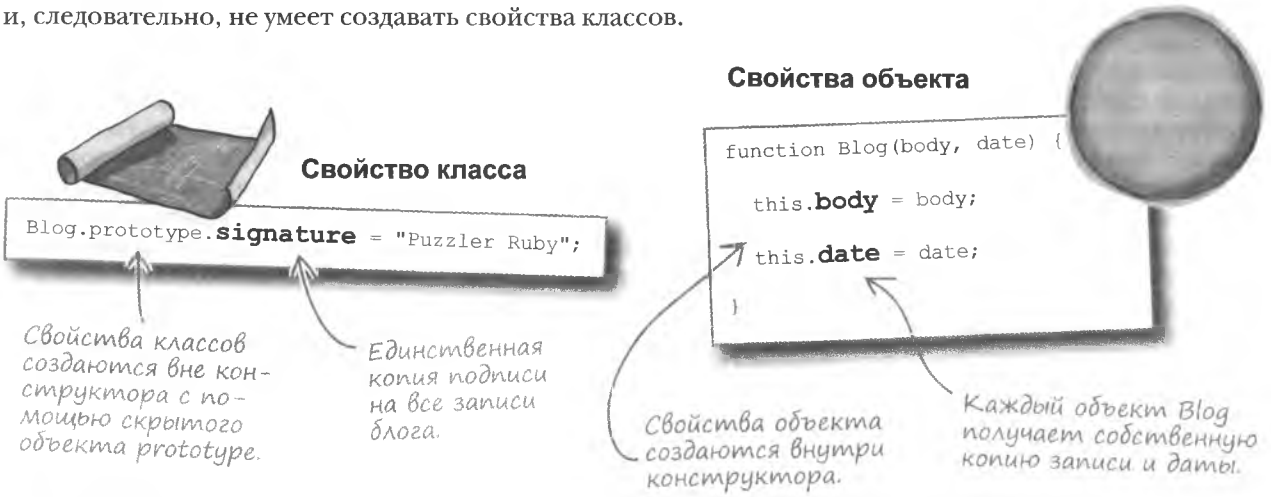
Для доступа к такому свойству нужно указать его название после имени объекта и точки.

Сначала указывается класс Blog, а потом объект prototype.

Вы, вероятно, уже догадались, что свойства класса сохраняются в объекте prototype.

Свойства класса не нуждаются в инициализации, но в данном случае это имеет смысл, так как мы уже знаем автора блога.

По виду этого кода не понятна его самая интересная особенность — в отличие от кода, создающего свойства объектов, его не нужно помещать внутрь конструктора. Ведь конструктор используется для создания объектов и, следовательно, не умеет создавать свойства классов.



Возьми в руку карандаш

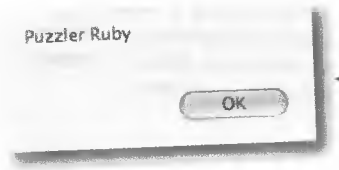
Напишите код, отображающий значение свойства `signature` в отдельном окне. Подсказка: мы предполагаем, что код располагается внутри метода `Blog`.

.....

Возьми в руку карандаш

Решение

Вот как выглядит код, отображающий значение свойства `signature` в отдельном окне.




```
alert(this.signature);
```

Доступ к свойствам классов, как и к свойствам объектов, осуществляется при помощи ключевого слова this!

Часто Задаваемые Вопросы

В: А зачем вообще в блоге YouCube сохранять подпись в виде свойства? Почему нельзя сделать ее частью текста записи?

О: Разумеется, подпись можно включить в текст каждой записи, но это потребует лишнего времени и усилий, при том что в блог пишет всего один человек. Да и зачем Руби подписывать вручную каждый пост, если есть более простой способ это сделать. Кроме того, это исключает вероятность опечаток. Есть и другая возможность — использовать для подписи строковый литерал в процессе HTML-форматирования текста записи. Такой подход хорошо работает, но в результате важный фрагмент данных — подпись — оказывается зарыт где-то внутри кода форматирования. И при необходимости его не так-то просто найти и отредактировать. Поместив подпись в свойство класса, вы делаете ее легко доступной и, следовательно, легко обнаруживаемой и редактируемой.

В: Как бы изменилось создание записи, если бы подпись была свойством объекта?

О: Каждая реализация объекта имеет свой собственный набор свойств, инициализированный в конструкторе. Если бы подпись была свойством объекта, конструктору `Blog()` пришлось бы задавать ее каждый раз заново. Это несложная задача, так как конструктор просто присваивает свойству строку с подписью. Но при этом создается множество копий, и вы получаете возможность менять содержимое каждой подписи, независимо от остальных.

В: То есть если я захочу отредактировать блог YouCube таким образом, чтобы туда писали разные люди, стоит ли сделать подпись свойством объекта?

О: Именно это и следует сделать, потому что при таком сценарии свойство `signature` **должно будет** иметь разные значения для разных объектов. Лучше всего решить такую задачу, добавив к конструктору `Blog()` аргумент, позволяющий передавать в конструктор строку с подписью. Эту строку вы будете использовать для инициализации свойства `signature` каждого из объектов. Другими словами, вы будете работать со свойством `signature` точно так же, как и с другими свойствами объекта `Blog`.

В: Свойства классов напоминают глобальные переменные. Чем они различаются?

О: Свойства классов действительно напоминают глобальные переменные, ведь доступ к ним возможен из любой части сценария. Они даже создаются похожим образом — на уровне главного сценария, вне другого кода. Различие же заключается в том, что эти свойства связаны с классами, а следовательно, и с реализациями объектов. Это означает, что доступ к такому свойству всегда будет осуществляться относительно объекта.

В: Подождите. Доступ к свойствам классов *должен быть* выполнен через объекты?

О: Да, несмотря на то что свойства класса созданы при помощи объекта `prototype`, доступ к ним осуществляется через конкретные реализации. Для этого используется ключевое слово `this`, после которого стоит точка и имя объекта. Вся разница только в том, где именно хранится то или иное свойство — в классе или в отдельной реализации.

Подписан и доставлен

Создав свойство `signature` уровня класса и присвоив ему значение, Руби хочет посмотреть, как все работает. Взглянув на код, форматирующий запись блога для отображения в браузере, мы обнаружим код подписи в методе `toHTML()`.

Метод toHTML() форматирует подпись как часть записи блога.

```
Blog.prototype.toHTML = function(highlight) {

  // Используем для выделения серый фон

  var blogHTML = "";

  blogHTML += highlight ? "<p style='background-color:#EEEEEE'>" : "<p>";


  // Генерируем отформатированный HTML-код блога

  blogHTML += "<strong>" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
    this.date.getFullYear() + "</strong><br />" + this.body + "<br /><em>" + this.signature +
    "</em></p>";

  return blogHTML;

};
```

Ссылка на свойство signature уровня класса ничем не отличается от ссылки на обычное свойство объекта.

Подпись Руби появляется в каждой записи.

Теперь сразу видно, кто автор всех записей в блоге.

9/1/2008
Went ahead and ordered the scary 7x7x7 cube. Starting a mental exercise regimen to prepare.
by Puzzler Ruby

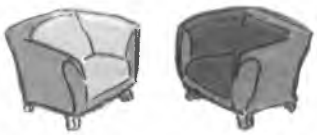
8/29/2008
Met up with some fellow cubers to discuss the prospect of a 7x7x7 cube. Mi
feelings.
by Puzzler Ruby

8/21/2008
Found a 7x7x7 cube for sale online. Yikes! That one could be a beast.
by Puzzler Ruby

8/19/2008
Solved the new cube but of course, now I'm bored and shopping for a new one.
by Puzzler Ruby

Руби воспользовалась приемом из объектно-ориентированного программирования, чтобы расширить язык JavaScript, добавив свойство `signature` к классу `Blog`. Этим она сделала блог YouCube еще более личной территорией.

Беседа у камина



Свойства объектов и классов говорят об обладании данными и секретных клубах.

Свойство объекта:

Так это о тебе я все время слышу. Должен сказать, что я не понимаю, зачем ты нужен. Я отлично выполняю свою работу, обеспечивая уникальность объектов и отслеживая значения их свойств.

Сложно поверить в такое. Продолжай...

То есть ты утверждаешь, что я не подхожу для хранения информации о секретных опознавательных знаках?

А могу ли я хранить секретный пароль?

Превосходно! Тогда я организую секретный клуб, и у каждого из пас будет свой пароль.

Прекрасно! А что это вообще такое? Нет, действительно. Я серьезно спрашиваю...

Свойство класса:

Конечно, ты это делаешь, и это достойно восхищения. Но ты задумывался, что иногда объектам не требуются их личные данные?

Ну, бывают ситуации, когда фрагменты данных являются общими для всех объектов. Как секретный опознавательный знак в секретном клубе. Его знают все члены клуба. И если кто-то из них изобретет свой опознавательный знак, вся система рухнет. Потому что кто-то захочет ему последовать, и вскоре никто не сможет опознать друг друга, так как знаков станет слишком много.

Именно так. Не обижайся, но все члены клуба должны знать только один знак.

Возможно. Если пароль у каждого свой, персональный, то да, ты прекрасно подходишь для его хранения.

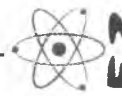
Но ведь ты не знаешь секретного опознавательного знака!

Нет дублирующемуся коду!

Руби не знает отдыха, она решила продолжить работу над эффективностью кода YouCube. Она заметила повторения в форматирующем коде и считает, что от них тоже неплохо было бы избавиться, воспользовавшись преимуществами объектно-ориентированного программирования.


```
<html>
 <head>
   <title>YouCube - The Blog for Cube Puzzlers</title>

   <script type="text/javascript">
     // Конструктор объекта Blog
     function Blog(body, date) {
       // Назначаем свойства
       this.body = body || "Nothing going on today.";
       this.date = date || new Date();
     }

     // Возвращаем строковое представление записи блога
     Blog.prototype.toString = function() {
       return "[" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
         (this.date.getFullYear() + "] " + this.body;
     };

     // Возвращаем отформатированное HTML-представление записи блога
     Blog.prototype.toHTML = function(highlight) {
       // Используем для выделения серый фон
       var blogHTML = "";
       blogHTML += highlight ? "<p style='background-color:#EEEEEE'>" : "<p>";

       // Генерируем отформатированный HTML-код блога
       blogHTML += "<strong>" + (this.date.getMonth() + 1) + "/" + this.date.getDate() + "/" +
         this.date.getFullYear() + "</strong><br />" + this.body + "<br /><em>" + this.signature +
         "</em></p>";
       return blogHTML;
     };

     // Проверяем, содержится ли в блоге строка поиска
     Blog.prototype.containsText = function(text) {
       return (this.body.indexOf(text) != -1);
     };

     // Задаем подпись
     Blog.prototype.signature = "by Puzzler Ruby";
```

Этот код, форматирующий наши данные, повторяется.

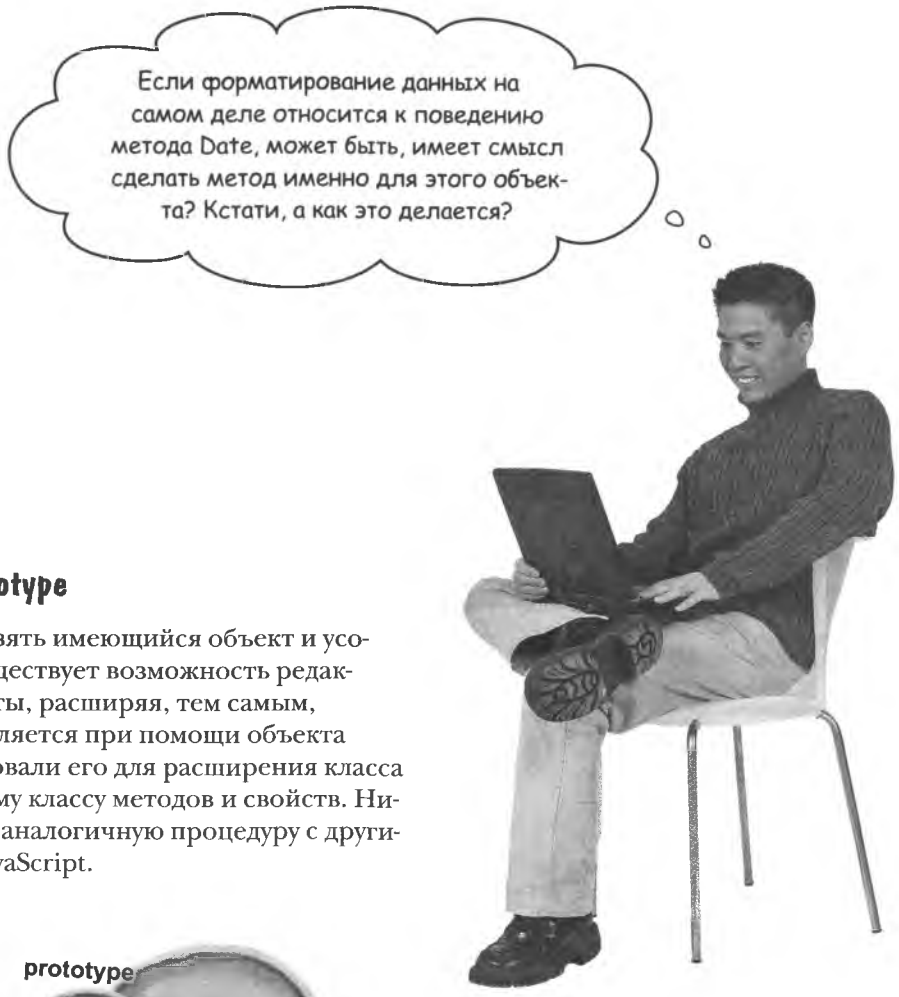


МОЗГОВОЙ ШТУРМ

Каким образом избавиться от дубликата форматирующего кода в блоге YouCube?

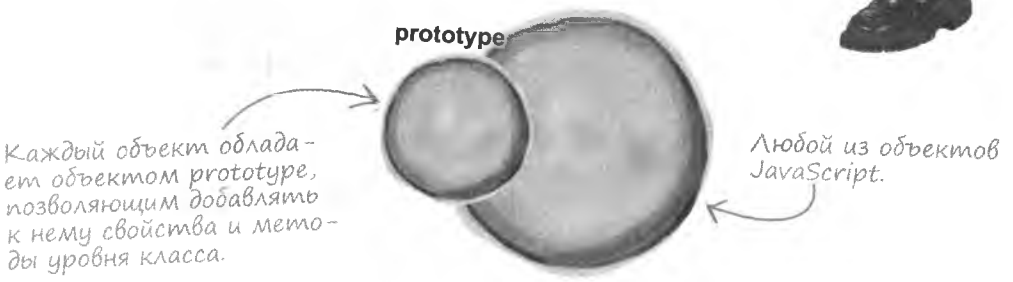
Метод форматирования данных

Руби считает, что для избавления от дубликатов форматирующего кода нужно добавить к объекту `Blog` еще один метод. Ведь чтобы использовать один и тот же фрагмент кода многократно, его нужно превратить в метод или функцию. А именно объект `Blog` отвечает за форматирование данных как часть форматирования записи блога. Или нужно действовать по-другому?



Вернемся к объекту prototype

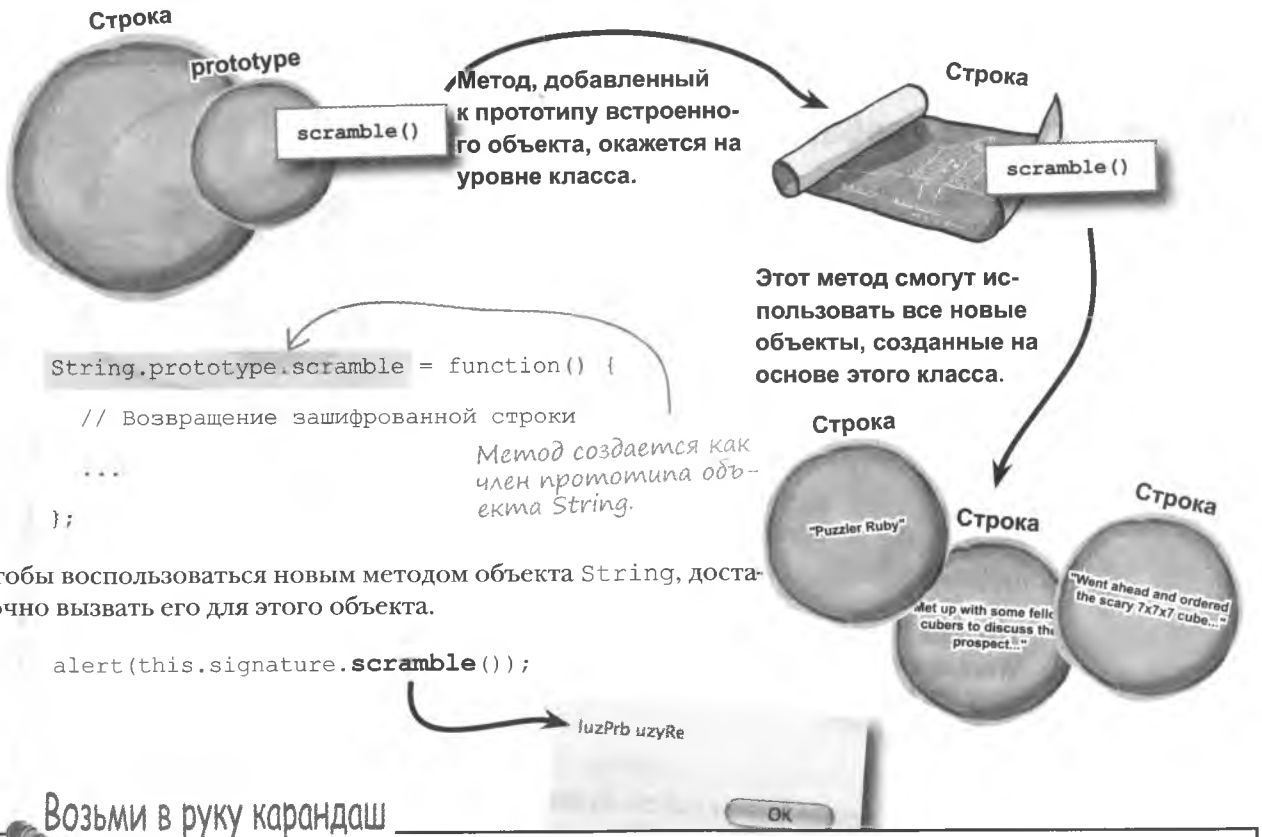
Что может быть лучше, чем взять имеющийся объект и усовершенствовать его? Ведь существует возможность редактировать стандартные объекты, расширяя, тем самым, язык JavaScript. Это осуществляется при помощи объекта `prototype`. Мы уже использовали его для расширения класса `Blog` путем добавления к этому классу методов и свойств. Ничто не мешает нам проделать аналогичную процедуру с другими встроенными классами JavaScript.



Расширение стандартных объектов

Расширение стандартных объектов осуществляется при помощи объекта `prototype`. Для этого достаточно добавить к прототипу нужные свойства и методы. В случае встроенных объектов JavaScript результатом будет доступ к добавленным свойствам и методам всех новых экземпляров объекта.

Объект prototype позволяет расширять встроенные объекты JavaScript.



Чтобы воспользоваться новым методом объекта `String`, достаточно вызвать его для этого объекта.

```
alert(this.signature.scramble());
```

Возьми в руку карандаш

Напишите код метода `shortFormat()`, который является расширением стандартного объекта `Date` и форматирует дату в виде *ММ/ДД/ГГГГ*.

.....

.....

.....

Возьми в руку карандаш

Решение

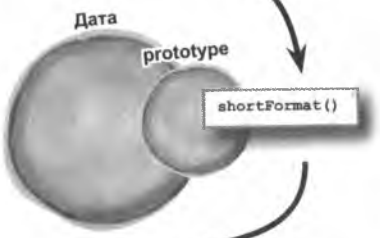
Вот как выглядит код метода `shortFormat()`, который является расширением стандартного объекта `Date` и форматирует дату в виде *ММ/ДД/ГГГГ*.

Метод добавлен к прототипу объекта Date.

```
Date.prototype.shortFormat = function() {
  return (this.getMonth() + 1) + "/" + this.getDate() + "/" + this.getFullYear();
};
```

Улучшенный блог YouCube

Редактирование объекта `Date` увеличивает эффективность сценария YouCube. Упрощается его редактирование, ведь форматирование данных теперь выполняется в одном месте, а его действие распространяется на все записи блога. Результаты такого редактирования далеко не всегда заметны визуально, но они улучшают код с точки зрения его долгосрочного использования.



9/3/2008
Attended a rally outside of a local toy store that stopped carrying cube puzzles. Power to the puzzlers!
by Puzzler Ruby

9/1/2008
Went ahead and ordered the scary 7x7x7 cube. Starting a mental exercise regimen to prepare.
by Puzzler Ruby

8/29/2008
Met up with some fellow cubers to discuss the prospect of a 7x7x7 cube. Mixed feelings.
by Puzzler Ruby

8/21/2008
Found a 7x7x7 cube for sale online. Yikes! That one could be a beast.
by Puzzler Ruby

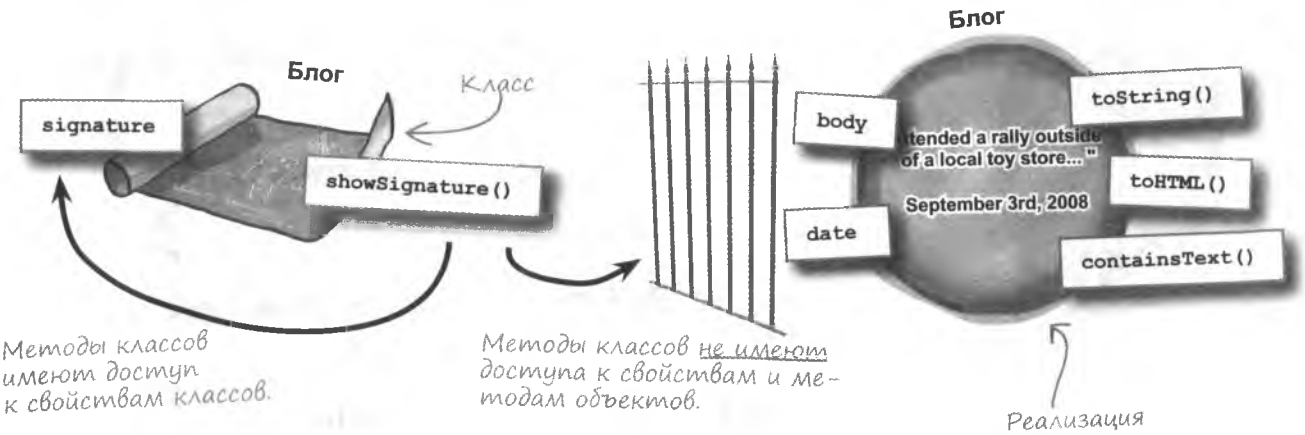
8/19/2008
Solved the new cube but of course, now I'm bored and shopping for a new one.
by Puzzler Ruby

Форматирование дат теперь выполняется пользовательским методом объекта Date.

Методы классов

Метод `shortFormat()` объекта `Date` является **принадлежащим классу методом объекта**. Именно это дает ему возможность форматировать сохраненные в отдельных объектах даты. Но можно создать и **метод класса**, который **не будет** иметь доступа к свойствам объектов. Доступными для них окажутся только свойства классов, такие как, например, свойство `signature` класса `Blog`.

Методы класса имеют доступ только к свойствам этого же класса.



Для создания метода класса объект `prototype` не нужен — достаточно назначить метод классу, указав имена класса и объекта, как показано ниже.

```
Blog.showSignature = function() {
  alert("This blog created by " + Blog.prototype.signature + ".");
};
```

Для доступа к свойству класса такому методу понадобится свойство объекта prototype.

Так как signature — это свойство класса, оно доступно для нашего метода.

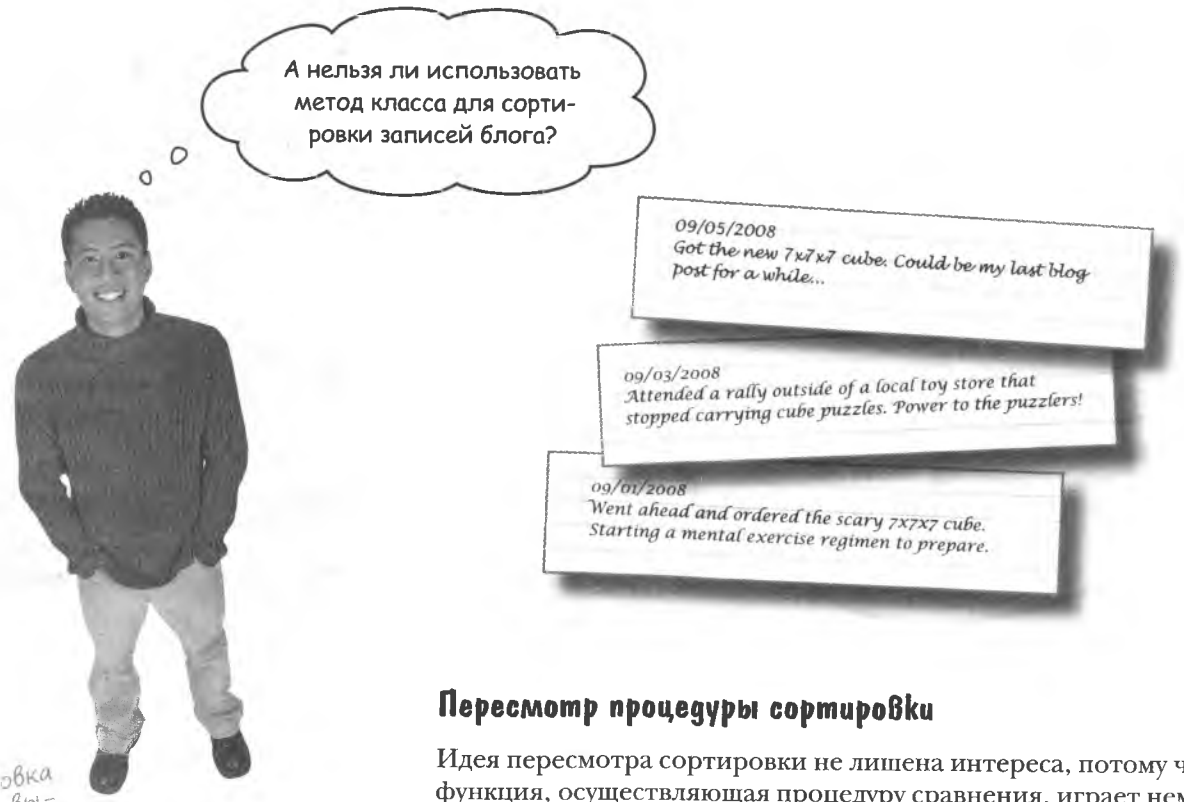
Так как методы классов не имеют связи с отдельными объектами, для их вызова достаточно сослаться на имя класса. Такой метод может быть вызван и объектом, но для этого потребуется указать имя класса.

```
Blog.showSignature();
```

Для вызова метода класса нужно указать имя этого класса.

МОЗГОВОЙ ШТУРМ

Существует ли код блога YouCube, который имеет смысл сделать методом класса `Blog`?



Пересмотр процедуры сортировки

Идея пересмотра сортировки не лишена интереса, потому что функция, осуществляющая процедуру сравнения, играет немалую роль, связанную с объектом `Blog`. В настоящее время она представляет собой литерал внутри функции `showBlog()`.

Сортировка записей выполняется внутри функции showBlog(), которая не является частью объекта Blog.

```
function showBlog(numEntries) {
  // Сортируем записи блога в обратном хронологическом порядке
  blog.sort(function(blog1, blog2) { return blog2.date - blog1.date; });
  ...
}
```

Вполне возможно переместить код сравнения в метод класса.

Одним из фундаментальных принципов объектно-ориентированного программирования является связь функциональности объекта с самим объектом. Другими словами, не следует оставлять на откуп внешнему коду действия, которые объект может выполнить самостоятельно. В данном случае сортировка записей вполне может вместо функции `showBlog()` выполняться средствами самого объекта. Но нельзя ли поместить осуществляющий эту процедуру метод в класс `Blog`? Для ответа на данный вопрос нужно понять, требуется ли этому методу доступ к данным или методам отдельных объектов.

Функция сравнения

Единственным способом ответить на заданный в конце предыдущего раздела вопрос является анализ функции сравнения. Вот соответствующий литерал:

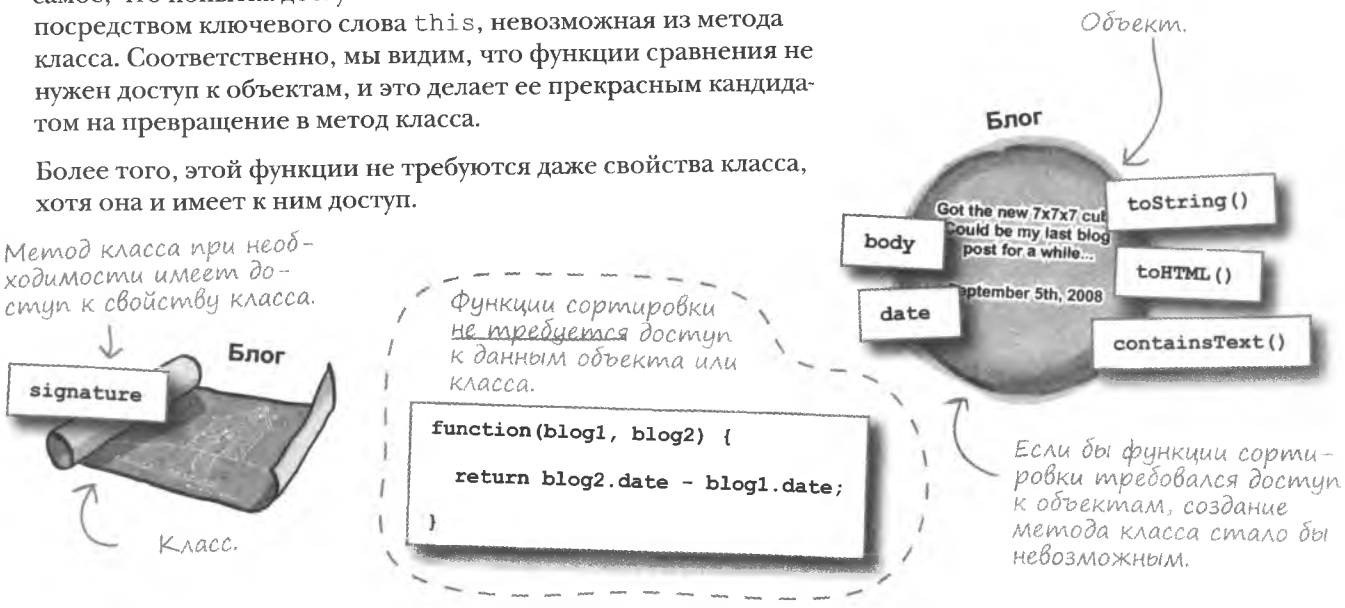
Два объекта blog передаются функции в качестве аргументов.

```
function(blog1, blog2) {

  return blog2.date - blog1.date;

}
```

Процедура сравнения сводится к вычитанию аргументов.

Хотя функция и имеет дело непосредственно с объектами, они передаются ей в качестве аргументов. Это совсем не то же самое, что попытка доступа к свойствам или методам объекта посредством ключевого слова `this`, невозможная из метода класса. Соответственно, мы видим, что функции сравнения не нужен доступ к объектам, и это делает ее прекрасным кандидатом на превращение в метод класса.

Более того, этой функции не требуются даже свойства класса, хотя она и имеет к ним доступ.



Возьми в руку карандаш

Перепишите код функции сравнения, превратив ее в метод класса `Blog` и присвоив ей имя `blogSorter()`.

.....

.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит функция сравнения записей блога YouCube после превращения ее в метод класса `Blog`.

```
Blog.blogSorter = function(blog1, blog2) {
  return blog2.date - blog1.date;
};
```

Сортировка теперь выполняется методом класса Blog, который называется blogSorter().

Вызов метода класса

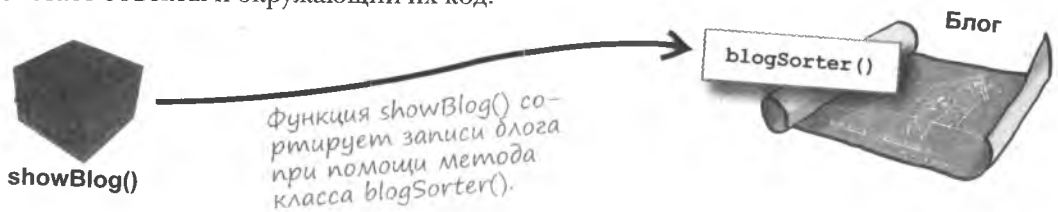
Преимущества преобразования функции сортировки записей блога в метод `Blog` становятся более ясными, если посмотреть на код вызова этого метода.

```
function showBlog(numEntries) {

  // Сортировка записей блога

  blog.sort(Blog.blogSorter);

  ...

}
```

Детали сортировки записей теперь отданы на откуп методу blogSorter() класса Blog.

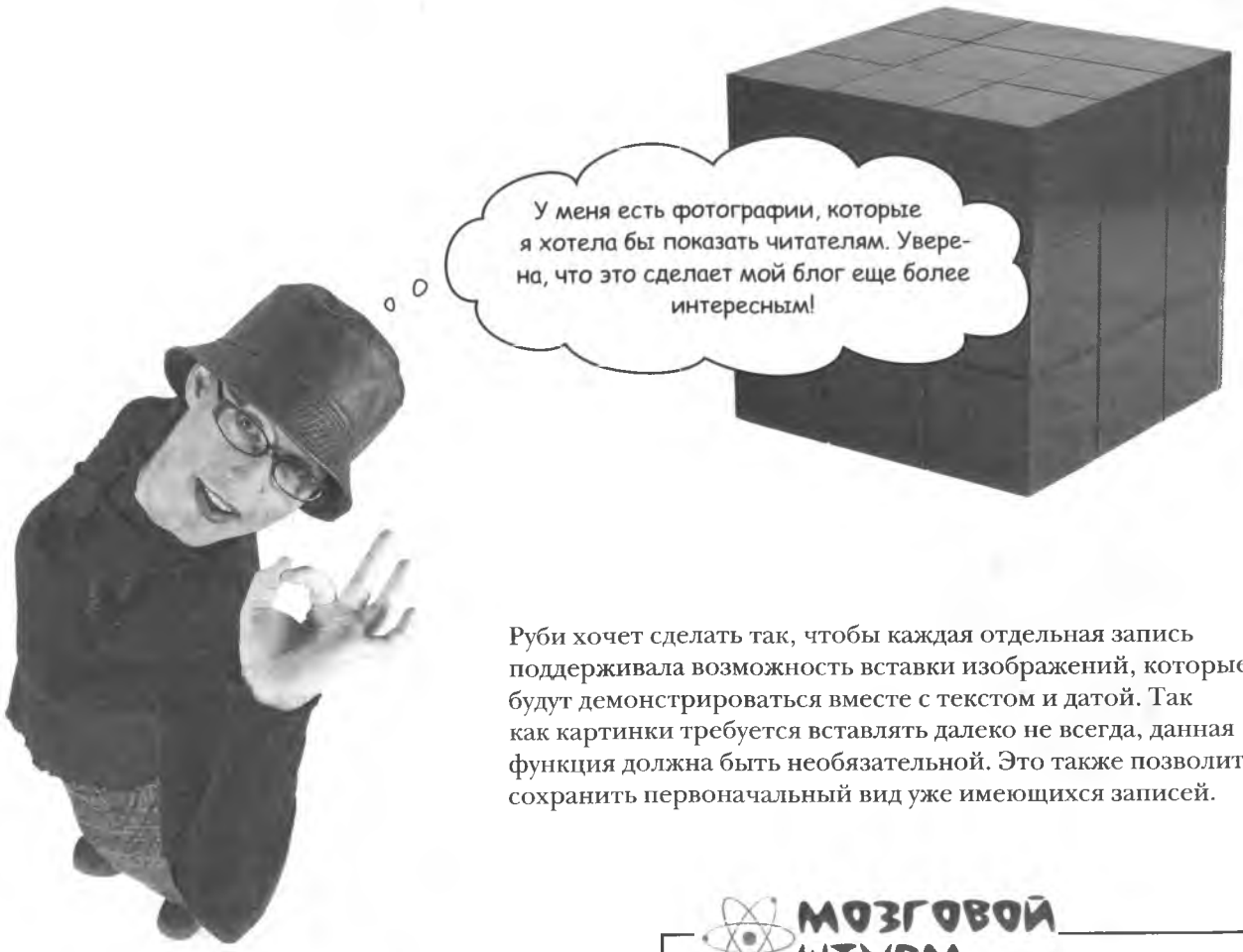
Прелесть этого кода открывается не сразу. Дело в том, что теперь сортировка записей блога выполняется не внешней функцией `showBlog()`, а внутри класса `Blog`, к которому эта процедура логически принадлежит.

Примечателен тот факт, что сортировка по-прежнему инициируется вне класса `Blog` функцией `showBlog()`, и это имеет смысл, так как данная процедура влияет на все записи блога. Но особенности сортировки отдельных записей блога таковы, что эту задачу можно решить средствами класса `Blog`. Хороший ООП-дизайн зачастую аккуратно сочетает объекты и окружающий их код.



В блоге
Картинка стоит тысячи слов

Руби продолжает восторгаться усовершенствованиями своего блога YouCube, но подозревает, что читатели вряд ли смогут разделить ее энтузиазм. Ведь пока что все внесенные изменения никак не повлияли на внешний вид блога. Поэтому Руби решила, что пришло время добавить что-то более заметное невооруженным глазом!



Руби хочет сделать так, чтобы каждая отдельная запись поддерживала возможность вставки изображений, которые будут демонстрироваться вместе с текстом и датой. Так как картинки требуется вставлять далеко не всегда, данная функция должна быть необязательной. Это также позволит сохранить первоначальный вид уже имеющихся записей.

МОЗГОВОЙ ШТУРМ

Каким образом включить поддержку изображений для объекта `Blog`?

Вставка изображений

Чтобы добавить возможность вставки изображений в записи блога YouCube, нам надо понять, каким образом внедрить новую структуру в объект `Blog`, не изменив способа его функционирования. По этому поводу возникает два вопроса:

1. **Каким образом лучше всего сохранить изображение в объекте Blog?**

2. **Как добавить к блогу необязательную возможность вставки изображений?**

Независимо от того, каким образом будет сохранено изображение, оно отображается на странице при помощи тега `<img>`.

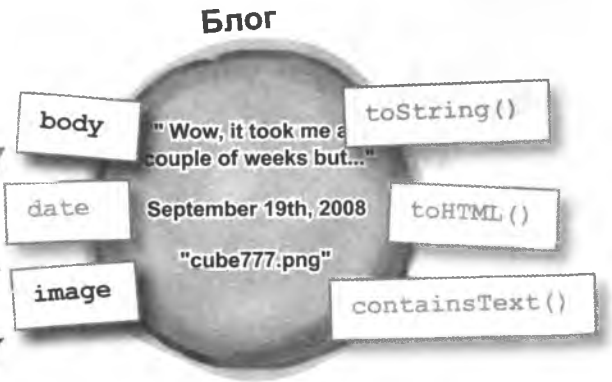
```
<img src="cube777.png" />
```

Достаточно указать имя файла с изображением.

Этот код показывает нам, что с точки зрения блога изображение – это только строка. Разумеется, она ссылается на файл с картинкой, хранящийся где-то на сервере, но для объекта `Blog` она все равно остается строкой.

Для объекта Blog изображение — это не более чем строка.

Поэтому изображения к объекту `Blog` можно добавить как свойство, хранящее строку, аналогичное свойству `body`.

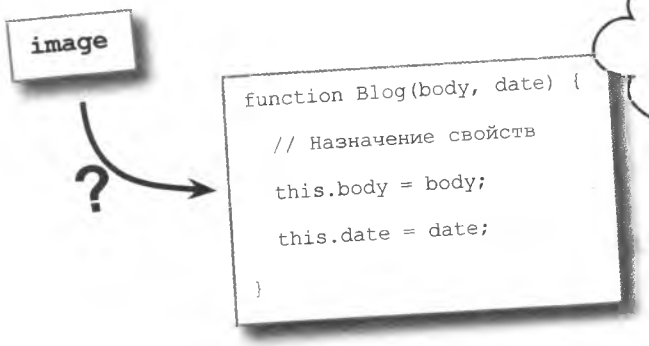


Изображение кубика размером 7x7x7 хранится в файле с именем cube777.png.

Свойства body и image хранятся в объекте Blog, как строки.

Необязательное изображение

2 Итак, в объекте `Blog` изображение сохраняется в виде строкового свойства `image`, но остается вопрос, каким образом сделать эту процедуру необязательной. Чтобы ответить на него, вернемся к конструктору, как к месту создания и инициализации объектов. Именно туда следует поместить особый код, указывающий на необязательность нового свойства.



Отсутствующие аргументы равны нулю.

Непереданный в функцию метод или непереданный конструктору аргумент для любого кода, который попытается его использовать, будет иметь значение `null`. В случае с конструктором это означает приравнивание к `null` связанного с аргументом свойства, что вовсе не плохо. Главное при этом не забыть поместить необязательный аргумент в самый конец списка, чтобы не возникало путаницы. Эта техника работает для любых функций и методов, но особенно она полезна для аргумента `image` в конструкторе `Blog()`.

Возьми в руку карандаш

Перепишите конструктор `Blog()` таким образом, чтобы он начал поддерживать необязательное свойство image.

Возьми в руку карандаш

Решение

Вот как выглядит конструктор `Blog()`, поддерживающий необязательное свойство image.

```
function Blog(body, date, image) {
  // Назначение свойств
  this.body = body;
  this.date = date;
  this.image = image;
}
```

Аргумент image добавлен в самый конец списка передаваемых конструктору аргументов.

Свойство image, и ему присвоен аргумент image.

Часто Задаваемые Вопросы

В: Обязательно ли ставить аргумент `image` на последнее место в списке передаваемых конструктору `Blog()` аргументов?

О: Да, ведь изображение относится к необязательному фрагменту записи. Весь вопрос тут в способе передачи аргументов функциям. Если функция имеет два аргумента, можно передать ей оба, только первый или ни один из них. Но невозможно передать **только** второй аргумент.

Именно поэтому необязательные аргументы ставятся в самый конец списка. Также имеет смысл более важные аргументы ставить в самое начало. Так как аргумент `image` является для конструктора `Blog()` необязательным, мы ставим его в конец списка, где им легко пренебречь.

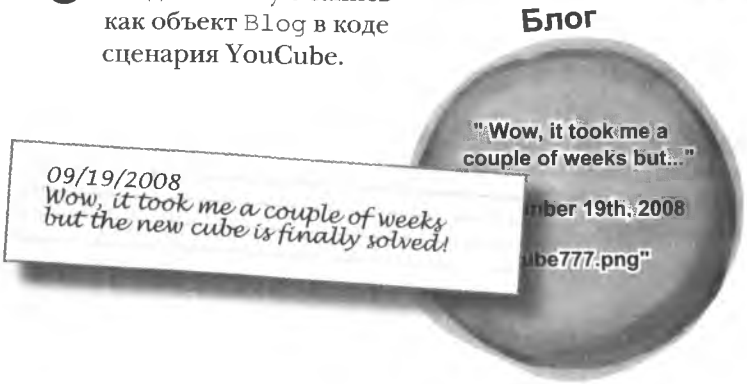
Добавление галереи

Новый блистательный конструктор `Blog()`, поддерживающий вставку изображений, бесполезен, если его не применять. Вот как выглядят этапы создания записи с картинкой:

1. На сервере поместите графический файл в ту же папку, что и страницу YouCube.

cube777.png

2. Создайте новую запись как объект `Blog` в коде сценария YouCube.



В результате мы получим код, который создает новую запись, передавая строковое свойство `image` в последний аргумент конструктора `Blog()`:

```
new Blog("Wow, it took me a couple of weeks but the new cube is finally solved!",
  new Date("09/19/2008"), "cube777.png")
```

Изображение передается в последний аргумент конструктора Blog().

Отображение картинок

Теперь, когда записи блога снабжены картинками, осталось одно, последнее усовершенствование. Весь этот разговор о конструкторах и необязательных аргументах не имеет особого значения, если отображающий запись код не учитывает новое свойство `image`.

Этот код расположен в методе `toHTML()`. Именно данный метод отвечает за HTML-форматирование записей блога, но теперь он должен принимать во внимание свойство `image`, как имеющее значение, так и без него. В данный момент появилось два способа отображения записей, и их выбор зависит от наличия картинки.

Теперь отображение записей блога будет осуществляться в соответствии со следующей логикой.

If (картинка есть)

Отобразить запись с картинкой

Else

Отобразить запись без картинки

Возьми в руку карандаш

В методе `toHTML()` объекта `Blog` отсутствует фрагмент кода, отвечающий за отображение картинок. Впишите его и снабдите примечаниями.

```
if (                    ) {
  blogHTML += "<strong>" + this.date.shortFormat() +
    "</strong><br /><table><tr><td><img src='" + this.image +
    "'/></td><td style='vertical-align:top'>" + this.body + "</td></tr></table><em>" +
    this.signature + "</em></p>";
}
else {
  blogHTML += "<strong>" + this.date.shortFormat() + "</strong><br />" + this.body +
    "<br /><em>" + this.signature + "</em></p>";
}
```

Возьми в руку карандаш

Решение

Вот как выглядит метод `toHTML()` объекта `Blog` после вставки в него фрагмента кода, отвечающего за отображение картинок.

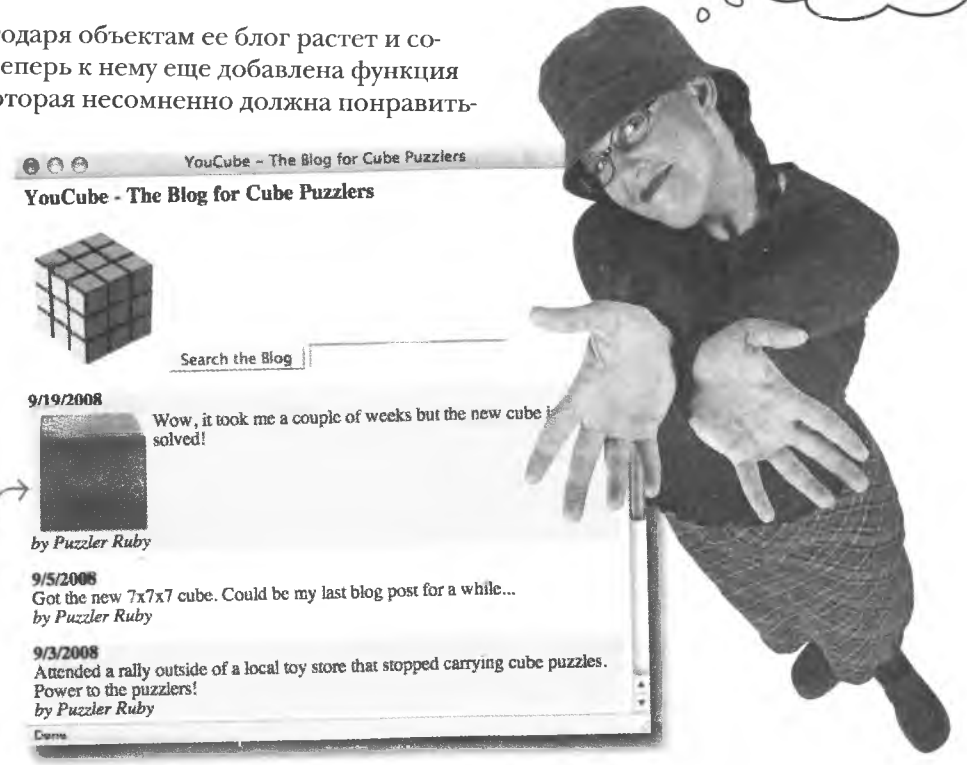
```
if ( this.image ) {
  blogHTML += "<strong>" + this.date.shortFormat() +
    "</strong><br /><table><tr><td><img src='" + this.image +
    "'/></td><td style='vertical-align:top'>" + this.body + "</td></tr></table><em>" +
    this.signature + "</em></p>";
}
else {
  blogHTML += "<strong>" + this.date.shortFormat() + "</strong><br />" + this.body +
    "<br /><em>" + this.signature + "</em></p>";
}
```

Если свойству image присвоена ссылка на изображение, результат проверки условия равен true, и картинка отображается.

В противном случае запись показывается в обычном виде, без картинок.

Блог на основе объектов

Руби в экстазе. Благодаря объектам ее блог растет и совершенствуется, а теперь к нему еще добавлена функция показа картинок, которая несомненно должна понравиться посетителям.

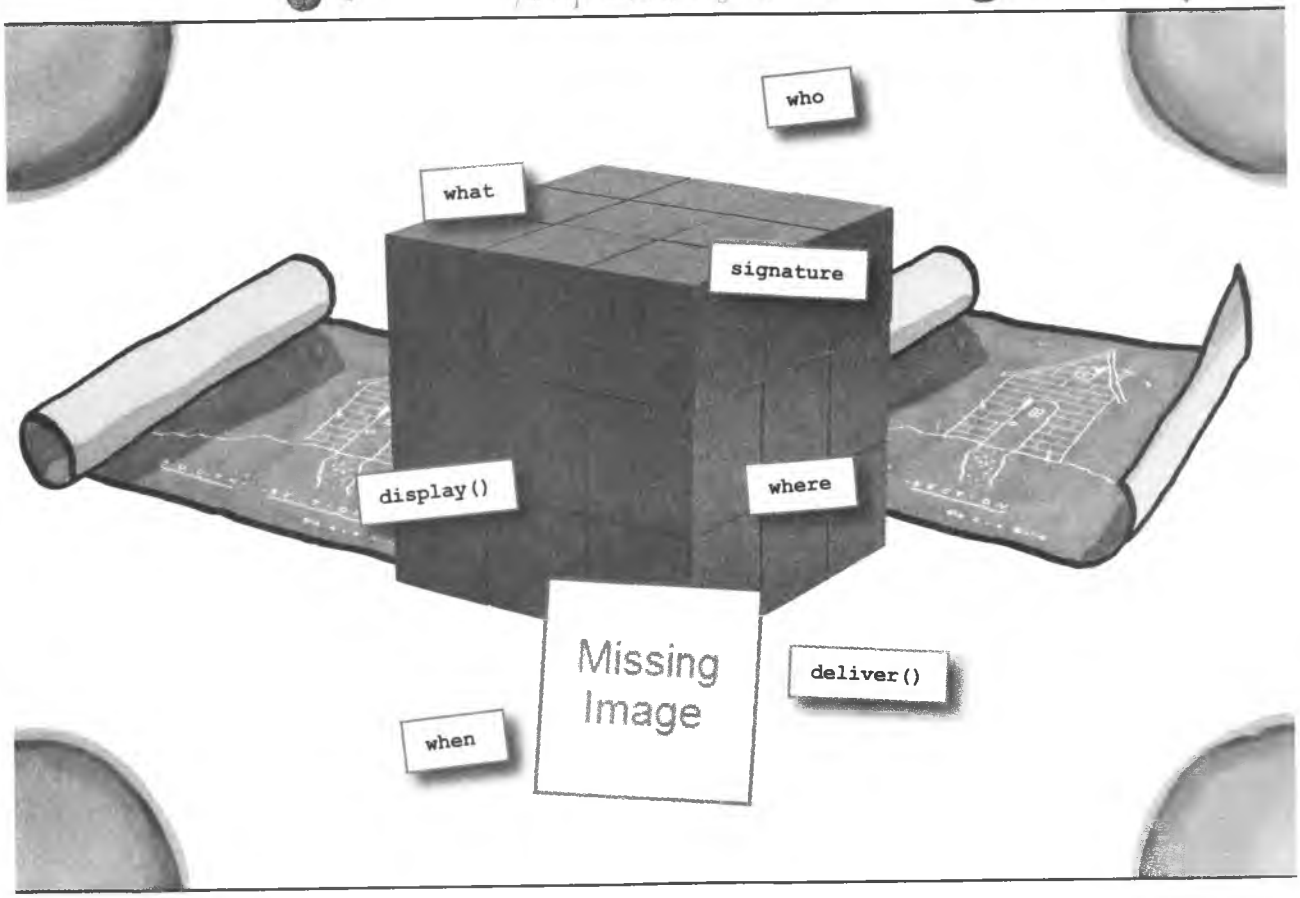


Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Что дают объекты большинству сценариев?

Ум хорошо, а два лучше!



Объекты добавляют к сценариям так много потрясающих вещей, что сложно выбрать что-то одно. Некоторые объекты выделяются из общей массы, что осложняет задачу. Но ответ очевиден!

Когда сценарий не работает

Даже самые лучшие планы в JavaScript иногда не реализуются. И когда это происходит, главное — не паниковать. Лучшие программисты не те, которые никогда не делали ошибок, — на самом деле это просто лгуны. Лучшие — это те, кто может **успешно обнаружить и устранить** ошибку. Отладчики высокой квалификации **нарабатывают хорошую манеру написания кода,** минимизирующую вероятность появления неприятных ошибок. ***Лучше предотвратить, чем потом бороться.*** Тем не менее ошибки то и дело встречаются, и вам нужен арсенал средств борьбы с ними...

Устранение дефектов

Шокирующий факт из жизни сладостей — стандарты на производство шоколадных батончиков не относят к браку около 60 разнообразных дефектов в плитке. А вот бояться брака в коде JavaScript причин нет. Этот код контролируется более тщательно, чем оборудование для производства шоколада. Существует даже специальная рабочая группа для устранения дефектов JavaScript.

BSI:
BUG SCENE INVESTIGATORS

К такой группе недавно присоединился Оуэн в качестве тестировщика JavaScript. Ему не терпится показать свои рабочие навыки, устранив как можно больше ошибок.

На пути к успеху Оуэну предстоит решить несколько задач. Ему предстоит овладеть непростым искусством борьбы с дефектным кодом JavaScript, и только тогда он сможет достичь намеченных высот.

Подсказка

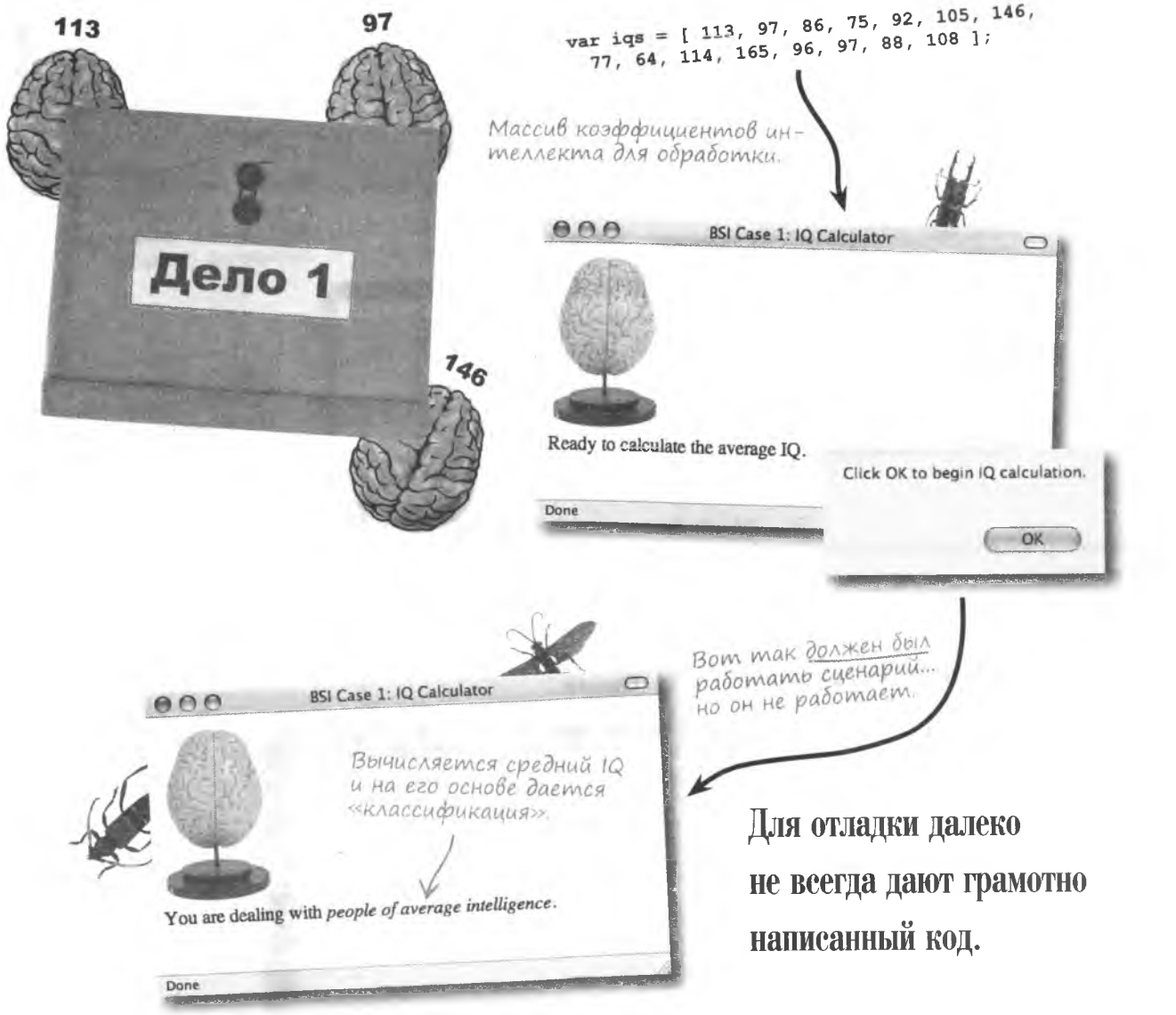
По американским продовольственным стандартам, шоколадный батончик может иметь около 60 разнообразных дефектов. А вот когда дело доходит до ошибок в коде JavaScript, люди в BSI проводят политику абсолютной нетерпимости, и это правильно.

Проблемы с калькулятором для IQ

Первым делом Оуэну поручили разобраться со сценарием, вычисляющим средний IQ на основе массива данных и составляющим группы из пользователей со сходными результатами. Взяв за основу массив чисел, этот сценарий вычисляет среднее и указывает уровень интеллекта этого среднего.

Скачать файлы, с которыми работает Оуэн, можно по адресу http://www.headfirstlabs.com/books/hfjs/.

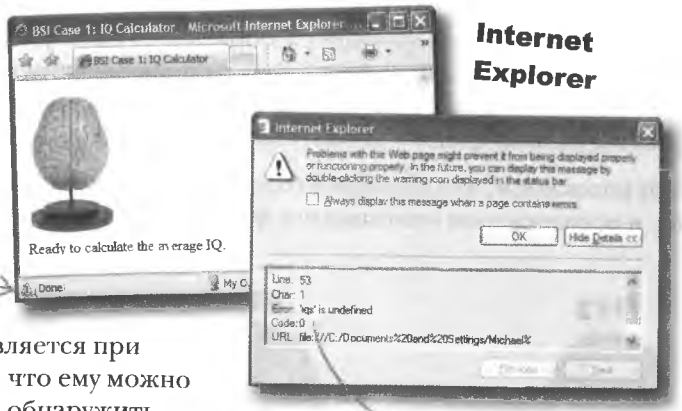
Оуэну рассказали, что сценарий содержит множество ошибок. К сожалению, в основном они описывались фразой, «оно не работает».



Для отладки далеко не всегда дают грамотно написанный код.

Различные баузеры

Оуэн думает, что понять проблему ему поможет прогон сценария в различных браузерах. И начинает с браузера Internet Explorer...

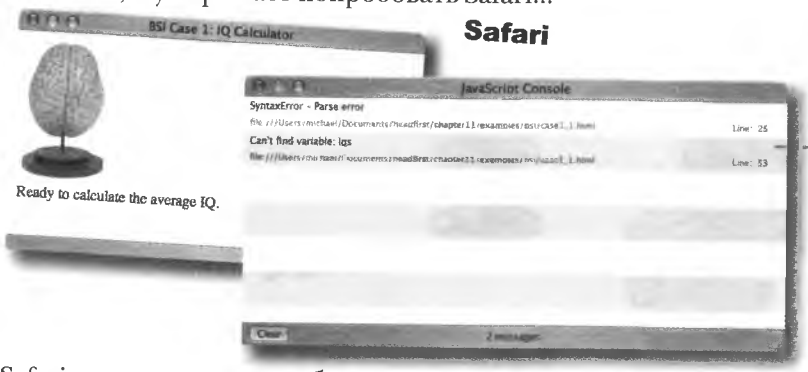


Internet Explorer

Двойной щелчок на желтом значке в нижнем левом углу браузера IE открывает окно ошибок.

В Internet Explorer сообщение об ошибке появляется при первой загрузке страницы, но Оуэн не уверен, что ему можно доверять. Достаточно взглянуть на код, чтобы обнаружить переменную `iqs`, в то время как браузер говорит о ее отсутствии. Зная, что браузеры далеко не всегда правильно сообщают об ошибках, Оуэн решает попробовать Safari...

В коде переменная iqs определена, поэтому ошибка кажется лишенной смысла.



Safari

Найдя строчку, в которую по мнению браузера Safari вкралась ошибка, вы обнаружите, что все в порядке.

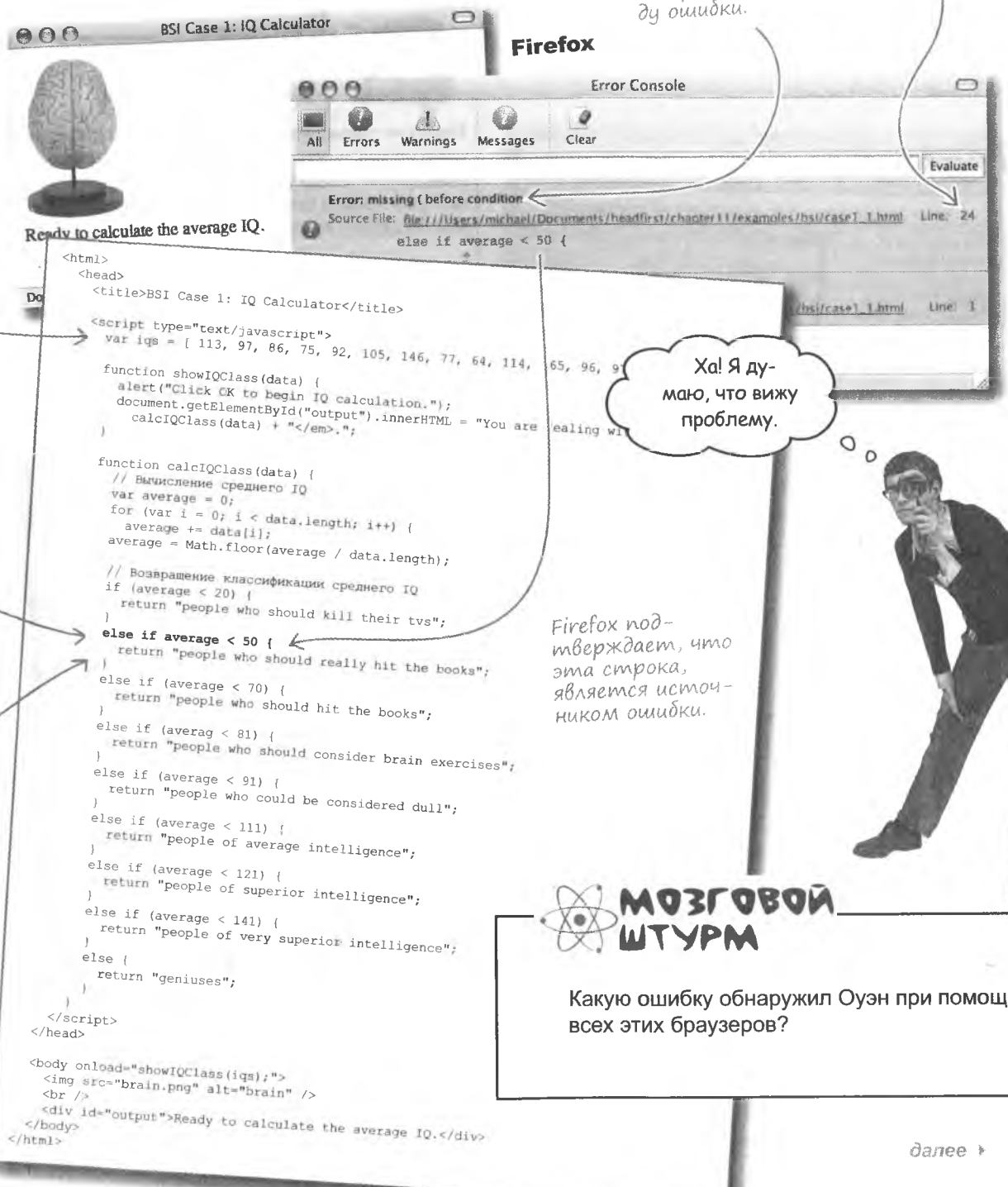
Браузер Safari показывает на ошибку в совсем другой строчке, но Оуэн на первый взгляд не видит там ничего криминального. Поэтому он решает попробовать еще раз, в браузере Opera...

Opera

При том что номер строки дается уже другой, код ошибки совпадает с показанным в Safari.

Окно Error Consoles дает возможность диагностировать проблемы с кодом JavaScript.

Происходит что-то странное. Opera показывает другой номер строки, но при этом воспроизводит код, показанный Safari, что для Оуэна, конечно, хорошо. Другое дело, что он не видит в этом коде ничего криминального. И поэтому решает попробовать еще и Firefox...



МОЗГОВОЙ ШТУРМ

Какую ошибку обнаружил Оуэн при помощи всех этих браузеров?

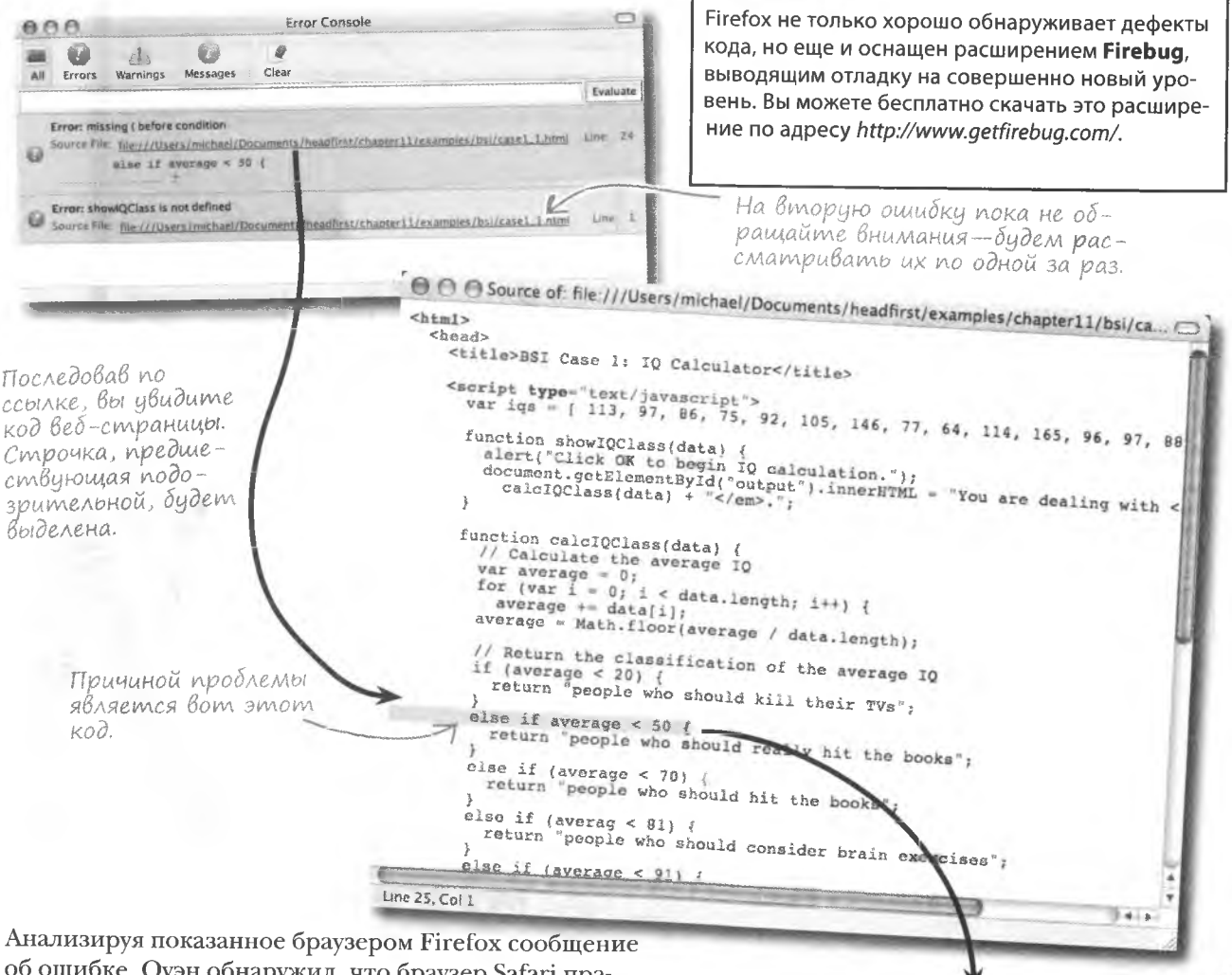
Firefox-спаситель

Обнаружив, насколько точно Firefox описал ошибку, Оуэн решил и дальше пользоваться его помощью. Он щелкнул на ссылке в консоли ошибок и оказался на строчке, предшествующей подозрительному коду.

Firefox считается наиболее подходящим для устранения дефектов браузером, по крайней мере, в настоящее время.

Подсказка

Firefox не только хорошо обнаруживает дефекты кода, но еще и оснащен расширением **Firebug**, выводящим отладку на совершенно новый уровень. Вы можете бесплатно скачать это расширение по адресу *http://www.getfirebug.com/*.



Анализируя показанное браузером Firefox сообщение об ошибке, Оуэн обнаружил, что браузер Safari правильно указал номер ошибочной строки (25). Firefox выделил и упомянул строку 24, но правильно показал, что ошибка содержится в строке 25. И что намного важней, Firefox объяснил, что именно не в порядке с кодом.

Часто Задаваемые Вопросы

В: Где находится консоль в моем браузере?

О: К сожалению, все браузеры разные, и иногда обнаружить консоль для просмотра ошибок JavaScript не так-то просто. Например, браузеры Safari для компьютеров Макинтош дают доступ к консоли только через меню Debug, доступ к которому по умолчанию отсутствует. Для получения доступа введите в приложение Terminal следующую команду (в одну строчку):

```
defaults write com.apple.Safari
IncludeDebugMenu 1
```

Если у вас другой браузер, почитайте документацию, там должно быть написано, как открыть консоль. В браузере Firefox она открывается командой Error Console из меню Tools.

В: Что делает браузер Firefox таким особенным?

О: Разработчики этого браузера проделали большую работу и дали своему детищу большие способности к поиску ошибок. Он превосходит другие браузеры в умении находить дефекты в коде сценариев и указывать на них. Возможно, в будущем и другие браузеры смогут выполнять данную функцию так же хорошо, но в настоящее время именно Firefox является самым мощным инструментом для отладки сценариев JavaScript.

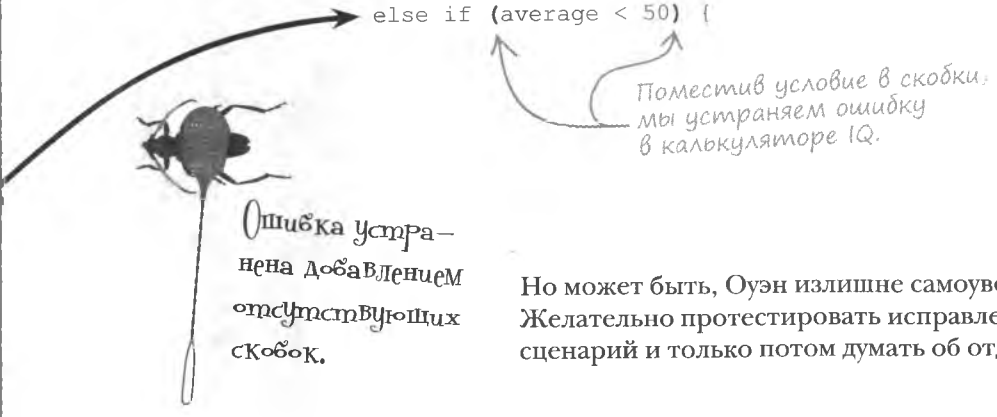
В: На какую ошибку указывал Internet Explorer?

О: Этого, к сожалению, узнать нельзя. Дело в том, что код сценария был загружен некорректно, и поэтому ошибка, о которой сообщает Internet Explorer, — это результат неверной работы интерпретатора JavaScript. О некорректной загрузке можно судить по тому, что переменная `iqs` была указана как «неопределенная», в то время как код ясно показывает процедуру создания `iqs`.

Напрашивается вопрос: содержит ли сценарий и другие ошибки и что на самом деле означает термин «неопределенный»?

Несложная отладка

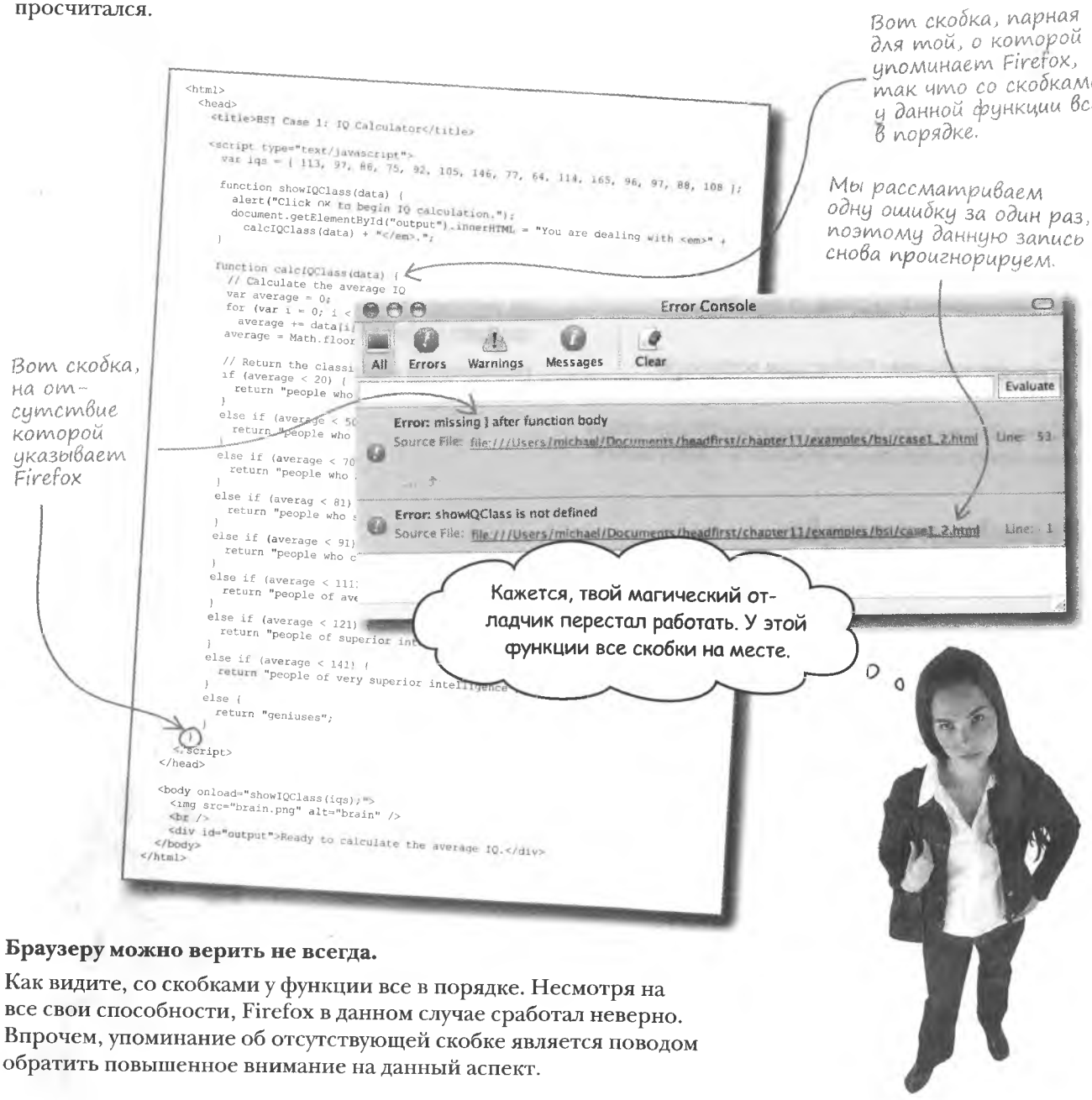
Оуэн счастлив от столь быстрого завершения поиска ошибки в калькуляторе IQ. С легкостью внеся исправления в код, он решает, что эта работа для него подходит. И уверен, что быстро станет в этом деле настоящим профессионалом.



Но может быть, Оуэн излишне самоуверен? Желательно протестировать исправленный сценарий и только потом думать об отдыхе...

Отчет не всегда указывает на ошибку

К сожалению, отладка калькулятора IQ еще не закончена, потому что Firefox снова указывает на ошибку, но уже в другом месте. Оуэн попытался положиться на выдаваемые браузером оценки, но на этот раз, кажется, просчитался.



Браузеру можно верить не всегда.

Как видите, со скобками у функции все в порядке. Несмотря на все свои способности, Firefox в данном случае сработал неверно. Впрочем, упоминание об отсутствующей скобке является поводом обратить повышенное внимание на данный аспект.

БУДЬТЕ интерпретатором JavaScript.

Проверьте скобки в данном сценарии, чтобы понять, что именно с ними не так.

```
<html>
  <head>
    <title>BSI Case 1: IQ Calculator</title>

    <script type="text/javascript">
      var iqs = [ 113, 97, 86, 75, 92, 105, 146, 77, 64, 114, 165, 96, 97, 88, 108 ];

      function showIQClass(data) {
        alert("Click OK to begin IQ calculation.");
        document.getElementById("output").innerHTML = "You are dealing with <em>" +
          calcIQClass(data) + "</em>.";
      }

      function calcIQClass(data) {
        // Calculate the average IQ
        var average = 0;
        for (var i = 0; i < data.length; i++) {
          average += data[i];
        average = Math.floor(average / data.length);

        // Return the classification of the average IQ
        if (average < 20) {
          return "people who should kill their tvs";
        }
        else if (average < 50) {
          return "people who should really hit the books";
        }
        else if (average < 70) {
          return "people who should hit the books";
        }
        else if (averag < 81) {
          return "people who should consider brain exercises";
        }
        else if (average < 91) {
          return "people who could be considered dull";
        }
        else if (average < 111) {
          return "people of average intelligence";
        }
        else if (average < 121) {
          return "people of superior intelligence";
        }
        else if (average < 141) {
          return "people of very superior intelligence";
        }
        else {
          return "geniuses";
        }
      }
    </script>
  </head>

  <body onload="showIQClass(iqs);">
    <img src="brain.png" alt="brain" />
    <br />
    <div id="output">Ready to calculate the average IQ.</div>
  </body>
</html>
```

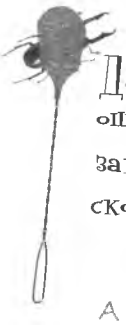
РЕШЕНИЕ ЗАДАЧИ

Вам нужно было проверить скобки в данном сценарии и понять, что именно с ними не так.

```
<html>
  <head>
    <title>BSI Case 1: IQ Calculator</title>

    <script type="text/javascript">
      var iqs = [ 113, 97, 86, 75, 92, 105, 146, 77, 64, 114, 165, 96, 97, 88, 108 ];

      function showIQClass(data) {
        alert("Click OK to begin IQ calculation.");
        document.getElementById("output").innerHTML = "You are dealing with <em>" +
          calcIQClass(data) + "</em>.";
      }

      function calcIQClass(data) {
        // Calculate the average IQ
        var average = 0;
        for (var i = 0; i < data.length; i++) {
          average += data[i];
        average = Math.floor(average / data.length);

        // Return the classification of the average IQ
        if (average < 20) {
          return "people who should kill their tvs";
        } else if (average < 50) {
          return "people who should really hit the books";
        } else if (average < 70) {
          return "people who should hit the books";
        } else if (averag < 81) {
          return "people who should consider brain exercises";
        } else if (average < 91) {
          return "people who could be considered dull";
        } else if (average < 111) {
          return "people of average intelligence";
        } else if (average < 121) {
          return "people of superior intelligence";
        } else if (average < 141) {
          return "people of very superior intelligence";
        } else {
          return "geniuses";
        }
      }
    </script>
  </head>

  <body onload="showIQClass(iqs);">
    <img src="brain.png" alt="brain" />
    <br />
    <div id="output">Ready to calculate the average IQ.</div>
  </body>
</html>
```

После выражения для переменной addition должна стоять закрывающая скобка.

У этой открывающей скобки отсутствует пара!



Для устранения ошибки добавим закрывающуюся скобку.

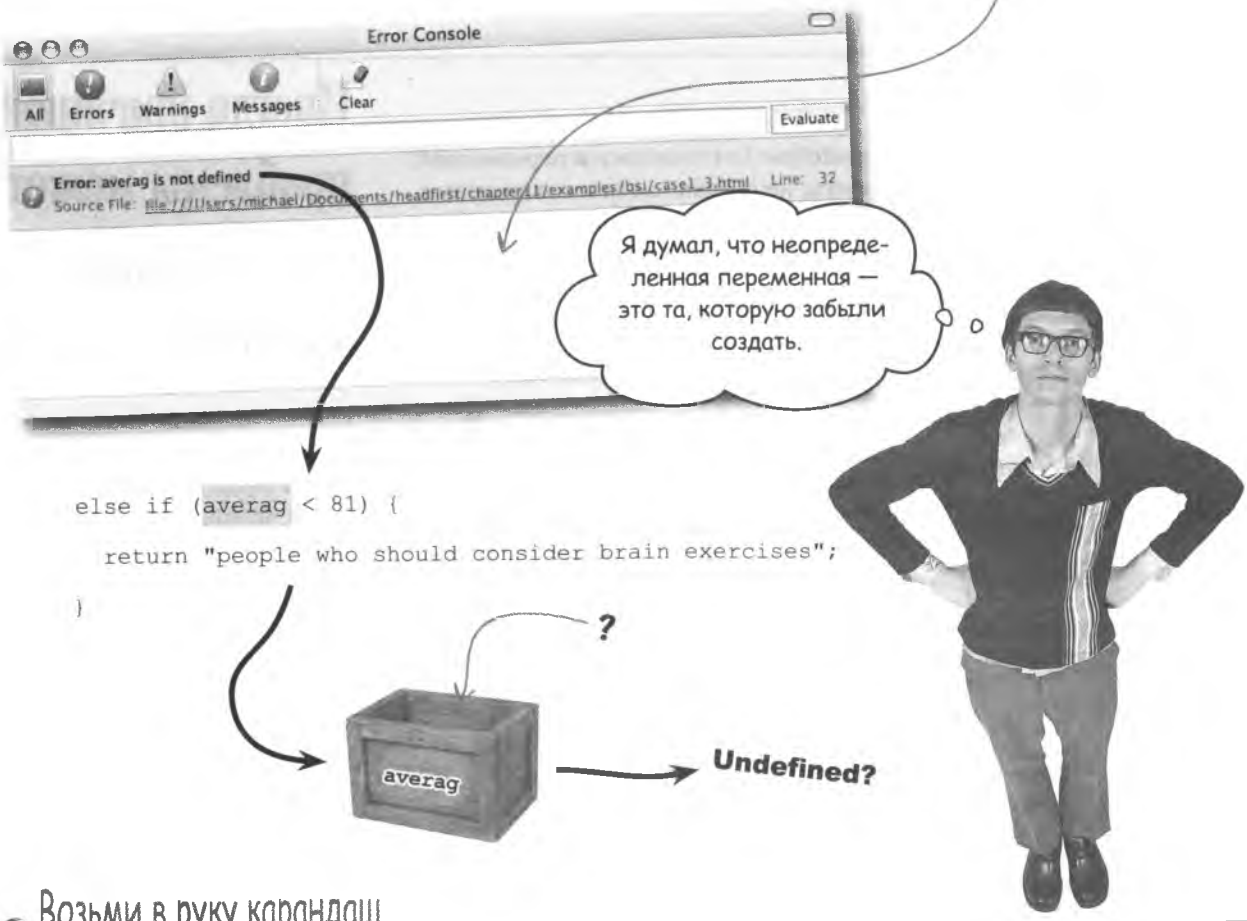
А можно, наоборот, убрать открывающую скобку после цикла for, ведь цикл все равно запускает только одну строчку кода. Скобки тут нужны только для наглядности.

Несовпадающие или отсутствующие скобки являются частой ошибкой в JavaScript. Но ее легко избежать, уделив повышенное внимание синтаксису.

Неопределенные переменные

Кажется, отдых Оуэну не светит, потому что калькулятор IQ обнаруживает все новые и новые ошибки. Теперь Firefox указывает, что переменная «не определена», что напоминает нам фиктивную ошибку, о которой уже докладывал Internet Explorer. Но на этот раз неопределенная переменная называется `averag`, а не `iqs`.

Обратите внимание, что вторая ошибка исчезла сама собой. Иногда внесенные в одном месте исправления способны устранить сразу несколько ошибок.



Возьми в руку карандаш

Как вы думаете, что означает термин undefined в контексте последней обнаруженной Оуэном ошибки?

.....

.....

.....

Возьми в руку карандаш

Решение

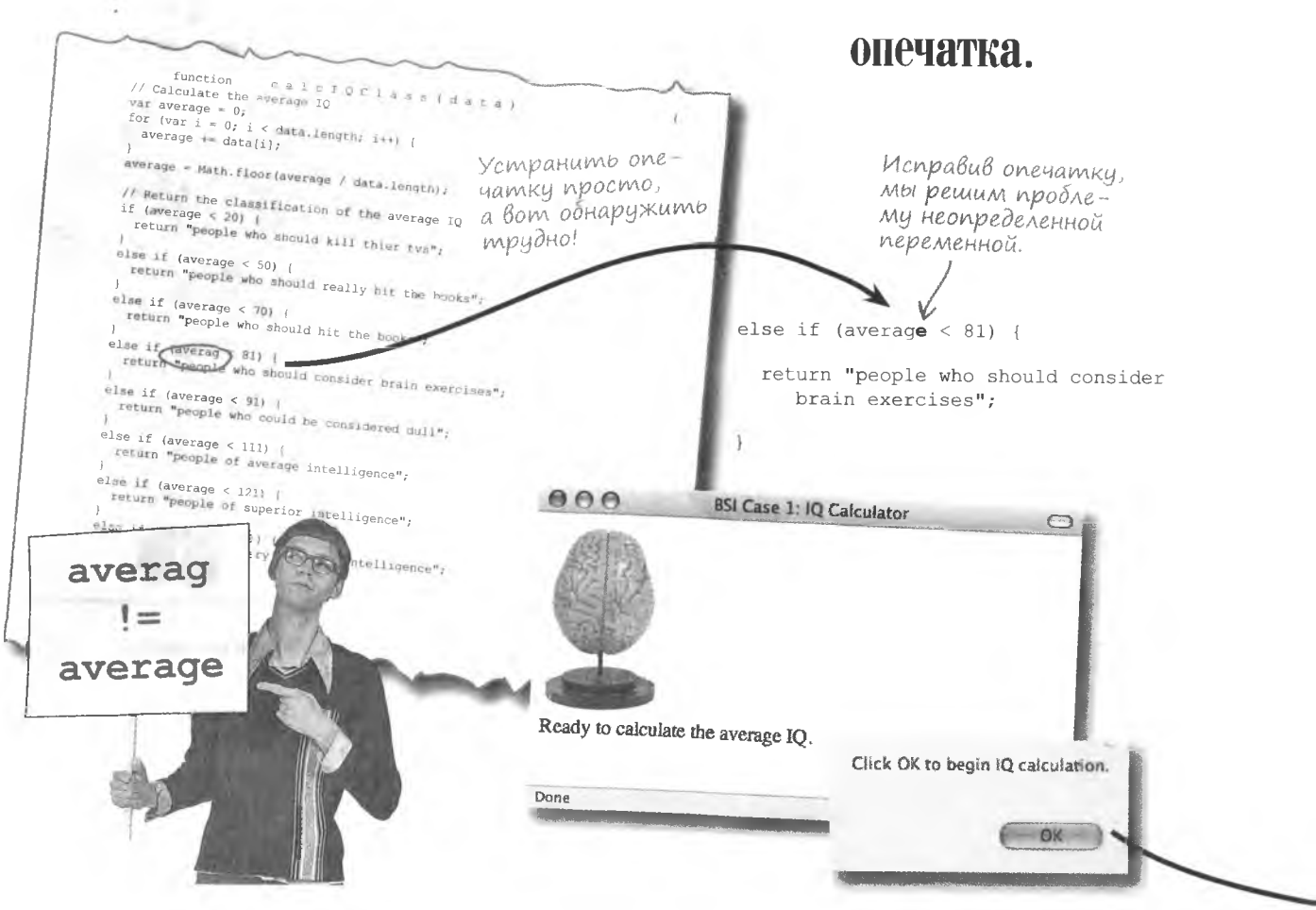
Вот что означает термин undefined в контексте последней обнаруженной Оуэном ошибки.

Термин Undefined относится как к несозданной переменной, так и к переменной, которой не было присвоено начальное значение. Ошибка возникает при ссылке на такую переменную.

Иногда все просто

В данном случае определение undefined относилось к переменной, которую попытались использовать, не создав, хотя и совершенно случайно. Причиной стала опечатка. В результате интерпретатор JavaScript решил, что перед ним совсем другая переменная.

Иногда причиной проблемы может стать обычная опечатка.



Часто Задаваемые Вопросы

В: Есть ли разница между «undefined» и «not defined»?

О: Нет. Термины означают одно и то же, просто некоторые браузеры используют один из них, а некоторые — другой.

В: А в чем тогда разница между «undefined» и `null`?

О: Здесь все сложнее. На техническом уровне между «undefined» и `null` существует разница, но она не настолько существенна, чтобы об этом задумываться. Дело в том, что `null` — это значение, которое можно присвоить переменной. Существует также тип данных `undefined`, к которому автоматически причисляются все переменные, которым пока не присвоено никакого значения. Но переменным никогда автоматически не присваивается значение `null`. Это делается вручную на начальном этапе, чтобы показать, что объект еще не был создан.

Про значения «undefined» и `null` нужно помнить только то, что в логических выражениях, например, при проверке условия в операторе `if` они дают значение `false`. Именно поэтому код вида `if (someObject)` используется для проверки существования объекта перед тем, как осуществить доступ к его членам.

В: А почему опечатка сделала переменную `average` неопределенной? Как это случилось?

О: Несмотря на то что переменная `average` была создана и инициализирована, JavaScript не может установить связи между переменными `averag` и `average` только потому, что их имена **похожи**. С точки зрения JavaScript переменная `averag` могла называться и `shazbot` или `lederhosen`. Другими словами, JavaScript интерпретирует ее как совершенно другую переменную.

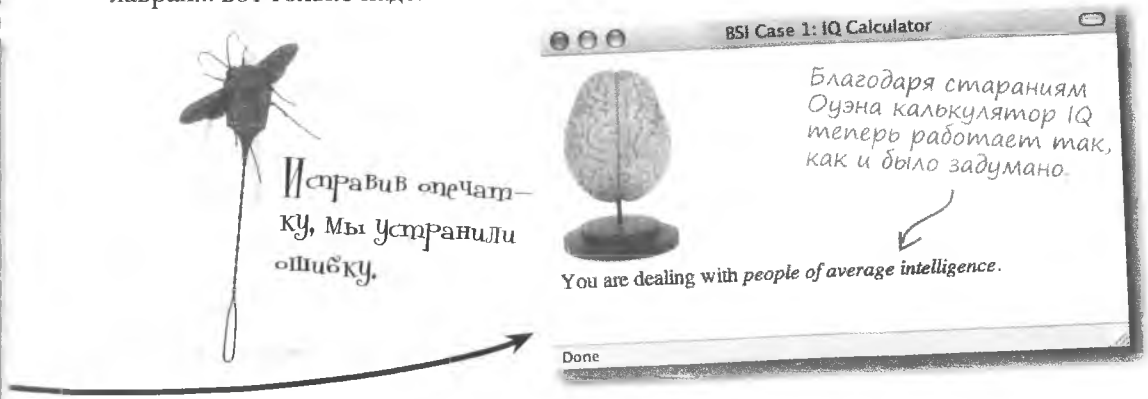
А так как этой новой переменной значение еще не присвоено, невозможно использовать ее в условии оператора `if`. Вы же не можете написать обзор фильма, даже не посмотрев его?

В: Вы шутите? Работая в текстовом редакторе, я постоянно делаю опечатки, и это ни на что не влияет. Почему JavaScript столь чувствителен?

О: К этому вам просто придется привыкнуть. Сценарии пишутся не для людей, а для машин, и машины не прощают ошибок, вне зависимости от того, на каком языке написан сценарий. Он перестанет работать даже из-за одного неверного символа. Существует определенная свобода в расстановке пробелов и знаков переноса строки, но сам код не должен содержать ошибок.

Работа с цифрами

После исправления опечатки калькулятор IQ начал работать корректно, вычисляя среднее из массива чисел и отображая по результатам соответствующую классификацию. Оуэн может считать работу выполненной и почить на лаврах... вот только надолго ли?

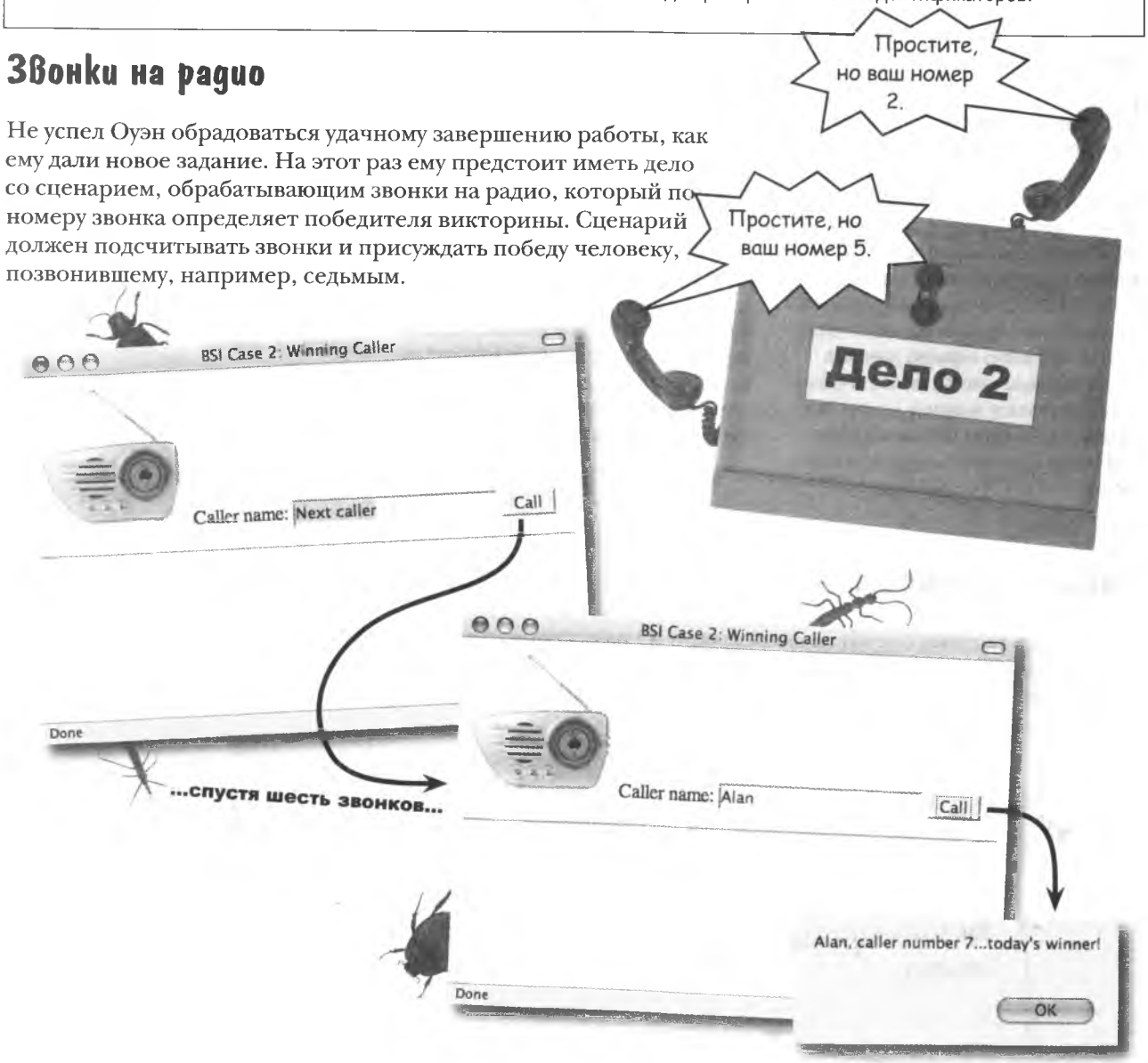


КЛЮЧЕВЫЕ МОМЕНТЫ

- Хотя большинство браузеров и снабжено консолью, на которой отображается информация об ошибках JavaScript, эта информация не всегда достоверна.
- Несмотря на излишне схематичный характер информации об ошибках, она зачастую дает понять, **на что именно** нужно обратить особое внимание.
- Распространенной ошибкой является неверная расстановка скобок вокруг блоков кода — следите, чтобы число открывающих и закрывающих скобок **совпадало**.
- Опечатку легко сделать, но трудно обнаружить — всегда проверяйте имена идентификаторов.

Звонки на радио

Не успел Оуэн обрадоваться удачному завершению работы, как ему дали новое задание. На этот раз ему предстоит иметь дело со сценарием, обрабатывающим звонки на радио, который по номеру звонка определяет победителя викторины. Сценарий должен подсчитывать звонки и присуждать победу человеку, позвонившему, например, седьмым.



Начинаем расследование

Перед испытанием страницы в браузере Оуэн решил взглянуть на ее код (его можно скачать по адресу *http://www.headfirstlabs.com/books/hfjs/*), чтобы понять, каким образом он работает. Ведь иногда можно сразу увидеть, что конкретно идет не так, или по крайней мере понять, как именно должен работать сценарий.

Имя звонящего и номер выигравшего звонка вместе с объектом form передаются в качестве аргумента функции checkWinner().

Если номер звонка совпадает с выигрышным, показываем имя звонящего в отдельном окне.

Счетчику звонков присваивается начальное значение ноль.

Увеличение показаний счетчика на единицу.

Если звонящий не является победителем, удаляем его имя из поля.

```
<html>
  <head>
    <title>BSI Case 2: Winning Caller</title>

    <script type="text/javascript">
      // Общее количество звонков
      var callNum = 0;

      function checkWinner(form, caller, winningNum) {
        // Увеличение номера звонка на единицу
        var callNum;
        ++callNum;

        // Проверка победителя
        if (callNum = winningNum) {
          alert(caller + ", caller number  + callNum + "... today's winner!");
          form.submit();
        }
        else {
          // Удаление информации о предыдущем позвонившем
          var callerField = document.getElementById('caller');
          callerField.value = "Next caller";
          callerField.focus();
          callerField.select();
        }
      }
    </script>
  </head>

  <body>
    <form name="callform" action="radiocall.php" method="POST">
      <img src="radio.png" alt="radio" />
      Caller name: <input id="caller" name="caller" type="text" />
      <input type="button" value="Call"
        onclick="checkwinner(this.form, document.getElementById('caller').value, 7)" />
    </form>
  </body>
</html>
```

Метод focus() устанавливает фокус ввода на элементе страницы.

Метод select() выделяет значение, сохраненное в текстовом элементе.

Сценарий сервера для сохранения информации о победителе.

Щелчок на кнопке Call вызывает функцию проверки победителя checkWinner().

Возьми в руку карандаш

Помогите Оуэну, указав, сколько ошибок, по вашему мнению, содержит данный сценарий.

Ноль **Одна** **Две** **Три** **Четыре** **Пять**

Возьми в руку карандаш

Решение

Вот сколько ошибок содержит код данного сценария.

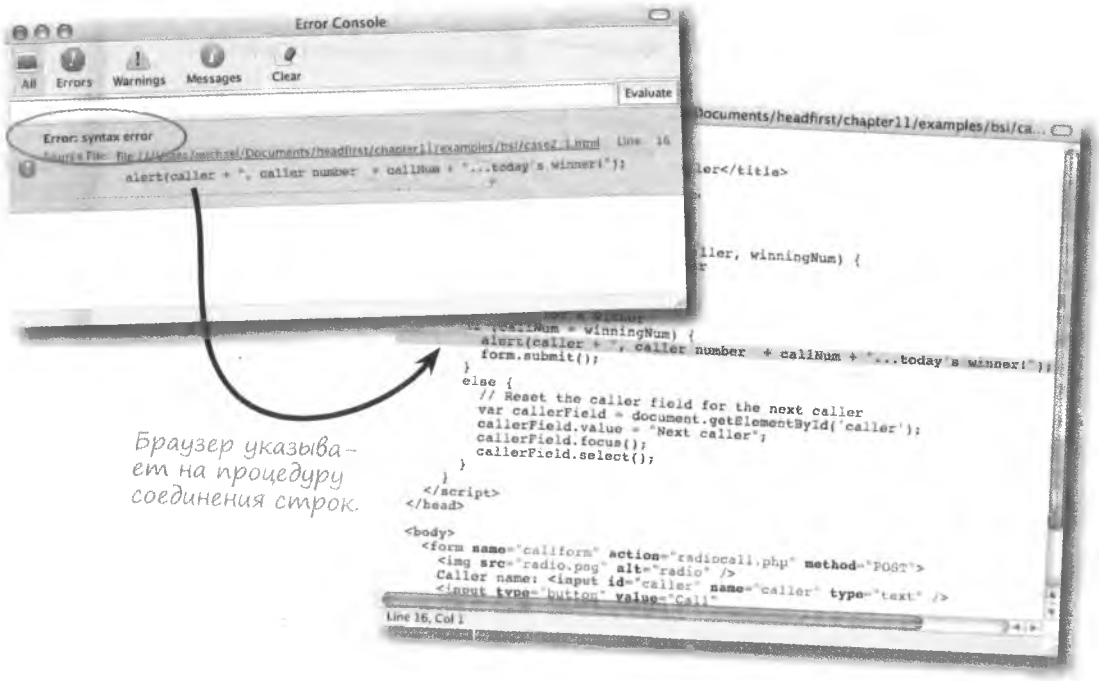
Ноль **Одна** **Две** **Три** **Четыре** **Пять**

Давайте продолжим и поможем Оуэну найти все эти ошибки...

Проверка синтаксиса (ошибка #1)

Получив представление о том, как должен работать этот сценарий, посмотрим, что происходит при попытке запустить его в браузере Firefox. Этот браузер немедленно дает сведения о синтаксических ошибках, то есть о нарушениях правил языка JavaScript.

Браузеры всегда сообщают о синтаксических ошибках.



При включенной функции сообщения об ошибках браузеры всегда информируют о проблемах с синтаксисом. Именно это послужит отправной точкой для наших изысканий.

Аккуратнее со строками

Firefox указал на процедуру соединения строк, и именно ее мы начнем анализировать. Вызывается функция `alert()`, в которой несколько строковых литералов соединяются с переменными `caller` и `callNum`.

```
if (callNum = winningNum)
  alert(caller +                    + callNum +                    );
```

Эти два строковых литерала объединяются с двумя переменными.

В JavaScript важна парность кавычек.

Кавычки всегда должны идти парами, иначе JavaScript не сможет определить, где кончается одна строка и начинается другая. В сценарии, обрабатывающем звонки на радио, в одном из строковых литералов не хватает кавычек. То есть перед нами банальная синтаксическая ошибка.

И достаточно добавить недостающие кавычки в конец строки:

```
if (callNum = winningNum)
  alert(caller + ", caller number " + callNum + "... today's winner!");
```

```
if (callNum = winningNum)
  alert(caller + ", caller number " + callNum + "... today's winner!");
```

Ошибка устранена добавлением парных кавычек.

Кавычки и апострофы

Недостающие кавычки — это только один вариант ошибки, возможный при работе со строками. Дело в том, что выделение строк в JavaScript и выделение атрибутов в HTML возможно как при помощи кавычек, так и при помощи апострофов. А значит, важно не перепутать эти два знака.

```
<input type="button" value="Call"
  onclick="checkwinner(this.form, document.getElementById('caller').value, 7)" />
```

Атрибуты HTML заключаются в кавычки.

Строки JavaScript внутри атрибута выделены при помощи апострофов.

Часто для выделения HTML-атрибутов используют кавычки, в то время как строки JavaScript, расположенные внутри этих атрибутов, выделяются апострофами. Но можно использовать и обратный способ выделения, как показано ниже:

```
<input type='button' value='Call'
  onclick='checkwinner(this.form, document.getElementById("caller").value, 7)' />
```

Современные стандарты XHTML не разрешают выделять атрибуты апострофами.

В этом примере апострофы используются для HTML-атрибутов, в то время как строка JavaScript заключена в кавычки.

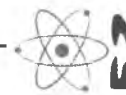
Важно заранее решить, что именно вы будете использовать для выделения определенных фрагментов кода. И так как XHTML, современная версия HTML, требует выделения атрибутов только при помощи кавычек, значит, для выделения строк JavaScript остаются апострофы.

В строках JavaScript и атрибутах HTML кавычки и апострофы могут заменять друг друга.

Проблема возникает, если вам требуются кавычки или апостроф, но этот знак уже используется, например, как показано ниже:

```
alert('It's so exciting!');
```

Будет ли работать этот код?



Что делать, если вам нужно воспользоваться знаком, применяемым для выделения строки?

Esc-символы

Распространенной ошибкой является использование кавычек или апострофов как символов в составе строки. Именно такая синтаксическая ошибка содержится в коде для всплывающего окна. Интерпретатор JavaScript не может распознать, какие апострофы указывают на конец строки, а какие — являются ее частью. К счастью, мы можем легко обозначить «реальность» символа, поместив перед ним обратную косую черту (\). В результате он превратится в так называемый **esc-символ**.

```
alert('It\'s so exciting!');
```

Теперь JavaScript точно знает, что мы хотели поместить в строку символ апострофа, а не указать на конец этой строки. Впрочем, мы могли и обмануть интерпретатор, заменив ограничивающие строку знаки на кавычки.

```
alert("It's so exciting!");
```

Esc-символ больше не нужен.

Esc-символы указывают на символьные литералы в строках.

Впрочем, можно придумать и вот такой вариант кода:

```
alert("They said, "you've won!"");
```

Так как в данном случае строка содержит в качестве литералов и кавычки, и апостроф, остается только прибегнуть к помощи esc-символов. Такой сценарий к тому же намного безопасней, потому что не допускает никаких вариантов трактовки.

```
alert("They said, \"you\'ve won!\"");
```

Esc-символ тут не обязателен, но лучше его поставить.

Упражнение

Устраните проблему, связанную с наличием кавычек и апострофов, в показанном ниже фрагменте кода.

```
var message = 'Hey, she's the winner!';

.....

var response = "She said, "I can't believe I won.""

.....

<input type="button" value="Winner" onclick="givePrize("Ruby");" />

.....
```

Упражнение Решение

Вот каким образом при помощи esc-символов были внесены исправления в код, содержащий множество апострофов и кавычек.

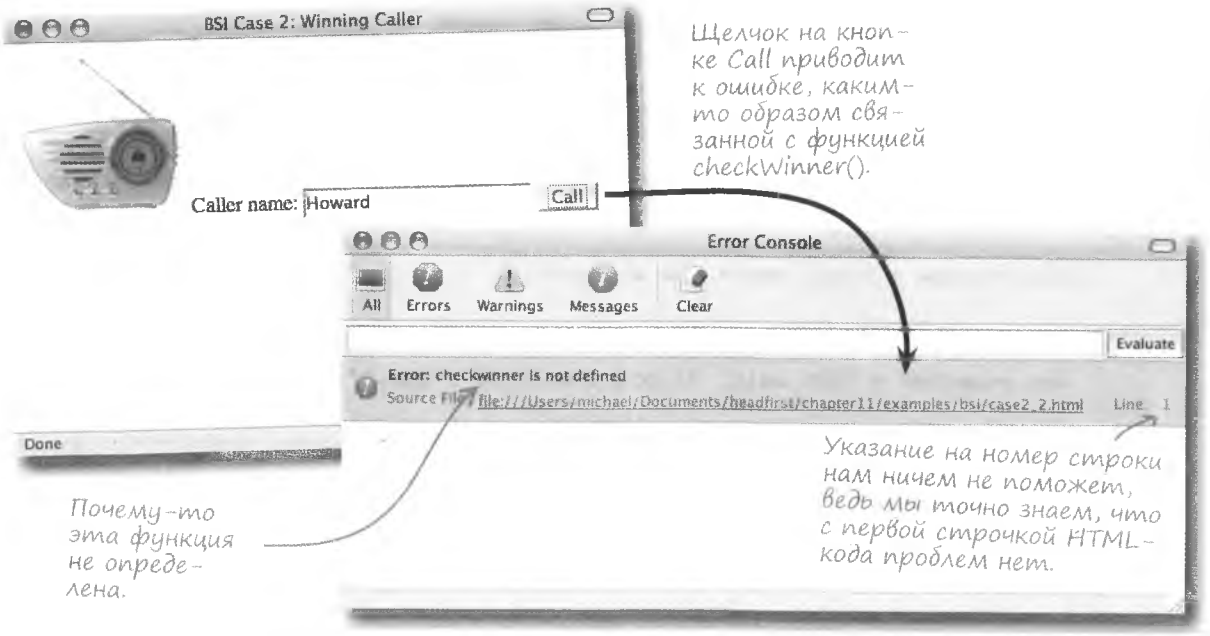
```
var message = 'Hey, she's the winner!';
var message = 'Hey, she\'s the winner!';

var response = "She said, "I can't believe I won.""
var response = "She said, \"I can\'t believe I won.\"";

<input type="button" value="Winner" onclick="givePrize("Ruby");" />
<input type="button" value='Winner' onclick="givePrize('Ruby');" />
```

Здесь esc-символ не обязателен, так как апостроф находится в строке, заключенной в кавычки.

В данном случае esc-символы работать не будут, так как у нас имеется строка JavaScript внутри HTML-атрибута. Проблема решается правильной расстановкой кавычек и апострофов.

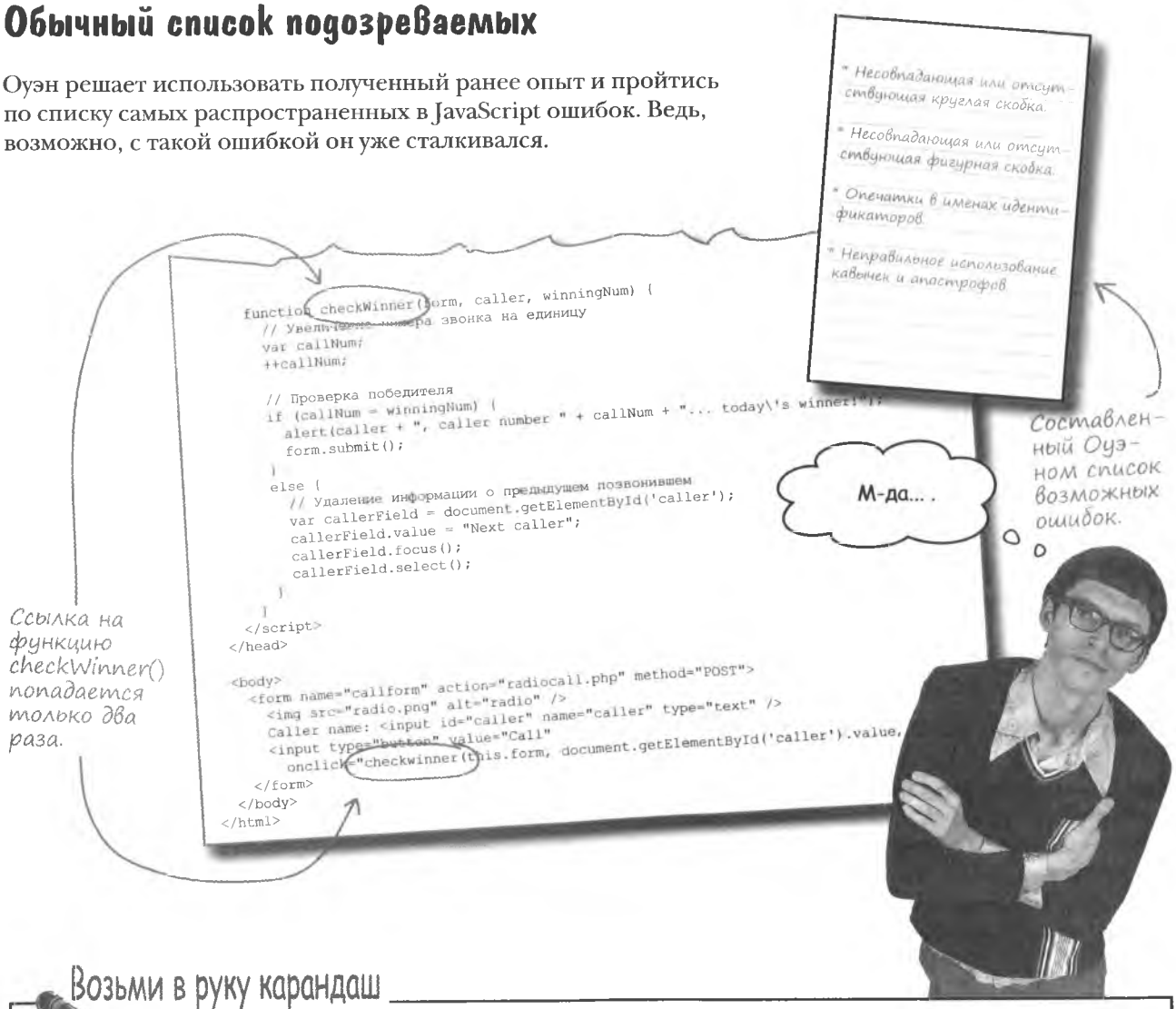
Неопределенность функции (Ошибка #2)

Итак, одна ошибка исправлена, но Оуэн знает, что работа далека от завершения. Работа обрабатывающего звонки сценария начинается гладко, без ошибок, а вот при щелчке на кнопке Call при попытке указать имя звонящего пользователь сталкивается с проблемой. И кажется, она связана с функцией `checkWinner()`.



Обычный список подозреваемых

Оуэн решает использовать полученный ранее опыт и пройтись по списку самых распространенных в JavaScript ошибок. Ведь, возможно, с такой ошибкой он уже сталкивался.



Возьми в руку карандаш

Укажите, какая проблема, с вашей точки зрения, мешает работе сценария обработки звонков.

- ☐ Несовпадающая или отсутствующая фигурная скобка.
- ☐ Несовпадающая или отсутствующая круглая скобка.
- ☐ Опечатка в имени идентификатора.
- ☐ Неправильное использование кавычек и апострофов.
- ☐ Какая-то другая проблема.

Возьми в руку карандаш

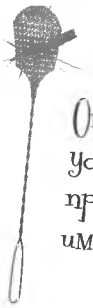
Решение

Вот в чем заключается проблема, мешающая работе сценария обработки звонков.

- ☐ Несовпадающая или отсутствующая фигурная скобка.
- ☐ Несовпадающая или отсутствующая круглая скобка.
- ☑ Опечатка в имени идентификатора.
- ☐ Неправильное использование кавычек и апострофов.
- ☐ Какая-то другая проблема.

Функция checkWinner() вызывается как checkwinner(), потому что в имени по ошибке появилась маленькая w. В результате интерпретатор пытается вызвать функцию, которая не была определена.

Для устранения ошибки достаточно заменить w на W в имени вызываемой функции.



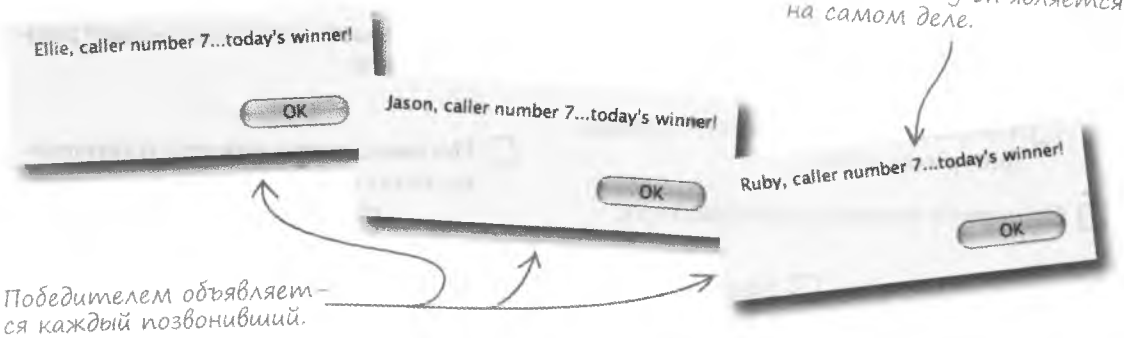
Ошибка #2 устранена исправлением имени функции.

```
<input type="button" value="Call"
  onclick="checkWinner(this.form, document.getElementById('caller').value, 7)" />
```

Побеждают все (Ошибка #3)

Даже после устранения досадной опечатки в имени функции сценарий не работает нужным образом. Хорошо хотя бы то, что браузер не выбрасывает сообщений об ошибках. Вот только выигрышным теперь является каждый звонок — сценарий при этом присваивает ему корректный номер. И если Оуэн не спасет ситуацию, радиостанция разорится на выдаче призов!

Странно то, что каждому звонку присваивается выигрышный номер, вне зависимости от того, каким по счету он является на самом деле.



Победителем объявляется каждый позвонивший.

Отладка с помощью Всплывающих окон

Мы знаем, что определение победителя производится путем сравнения переменной `callNum` с аргументом `winningNum` функции `checkWinner()`. Но что-то с этим кодом не так, поэтому давайте более подробно рассмотрим, что происходит с переменной `callNum`.

```
...
if (callNum = winningNum) {
...
```

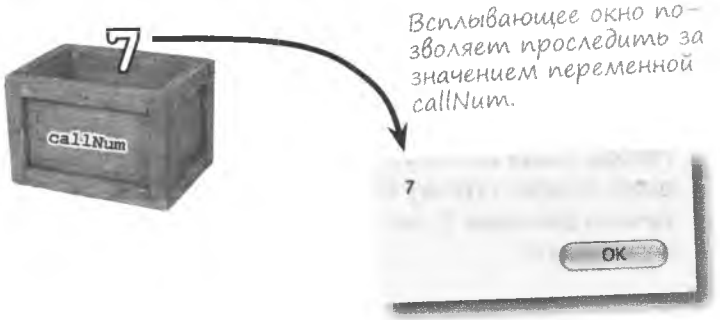
На первый взгляд с кодом все в порядке.

Имеет смысл проследить, как меняется значение переменной callNum по мере работы сценария.

Всплывающие окна являются отличным инструментом для отслеживания значений переменой.

Всплывающие окна как инструмент отладки.

Всплывающие окна годятся не только для отображения информации, предназначенной для конечных пользователей. С их помощью можно также отслеживать значения переменных и проверять корректность вызова отдельных фрагментов кода. В нашем случае при помощи всплывающих окон мы проследим за значениями переменной `callNum`.

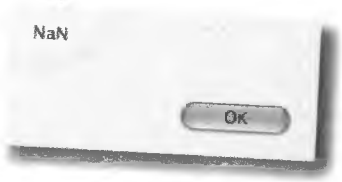


Следим за значением переменной

Термин «**контрольное значение**» означает непрерывное отслеживание переменной в процессе работы программы. Всплывающее окно не позволяет непрерывно следить за переменной, но в нашем случае оно вполне может помочь.

```
alert(callNum);
if (callNum = winningNum) {
  alert(caller + ", caller number " + callNum + "... today\'s
winner!");
  form.submit();
}
```

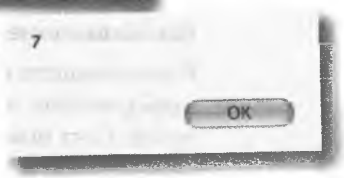
Что-то не так... Переменная callNum должна показывать номер текущего звонка.



Оуэн понял, что с точки зрения сценария переменные `callNum` и `winningNum` равны несмотря на то, что сразу перед оператором `if` переменная `callNum` имеет значение `NaN`. Уже одно это не совсем понятно, но Оуэн решил сначала посмотреть, что происходит внутри оператора `if`.

```
if (callNum = winningNum) {
  alert(callNum);
  alert(caller + ", caller number " + callNum + "... today\'s
winner!");
  form.submit();
}
```

Сразу после проверки условия переменная callNum получает значение 7.



МОЗГОВОЙ ШТУРМ

Теперь, когда вы знаете, что переменная `callNum` всего за одну строчку таинственным образом получила значение 7, как вы думаете, что стало причиной этого?

ОТЛАДКА ПРИ ПОМОЩИ ВСПЛЫВАЮЩЕГО ОКНА

Интервью недели: Всплывающее окно презирает дефекты

Head First: Должен признаться, я слышал о вас разное. Люди говорят, что часто вы их раздражаете. Но при этом вы являетесь лучшим другом отладчиков. Расскажите, кем же вы являетесь на самом деле.

Всплывающее окно: Эти первые просто ненормальные. Мне дают информацию, а я ее отображаю. Какой от этого вред?

Head First: Но вы «всплываете». Точно так же, как многочисленная надоедливая реклама.

Всплывающее окно: То есть, если плотник не умеет обращаться с молотком, виноват молоток?

Head First: Вы хотите сказать, что вас используют некорректно?

Всплывающее окно: Именно так. Если меня заставляют отображать всю эту глупую рекламу, мне это не нравится. Но выбора у меня нет. Впрочем, я думал, вы хотите узнать, как именно я помогаю процессу отладки.

Head First: Да, извините. Так как же вы это делаете?

Всплывающее окно: Очень просто. Скажем, переменной нечаянно присваивают какое-то не имеющее смысла значение. Программисту нужно отследить, как она меняется по мере работы сценария. И он просит меня отображать ее значения.

Head First: Но каким же образом вы умудряетесь узнать значение переменной в разные моменты времени? Наверное, это сложно...

Всплывающее окно: Вовсе нет. Достаточно вызвать меня в нужных точках сценария.

Head First: Понимаю. А сталкивались ли вы с проблемами в процессе такой работы?

Всплывающее окно: Должно признаться, что не так-то просто работать, если переменная меняется много раз, например, в процессе выполнения цикла.

Head First: Почему же?

Всплывающее окно: Как вы помните, я всплываю и убрать меня можно только щелчком. А если я всплываю много раз, нужна целая серия щелчков.

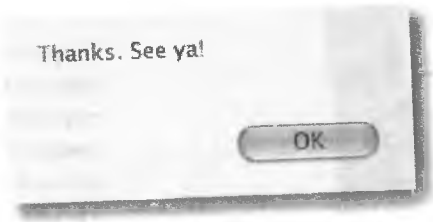
Head First: Тут вы правы. Но еще я слышал, что вы можете быть полезны даже в случаях, когда не требуется отслеживать данные.

Всплывающее окно: О, да. Существуют ситуации, когда неясно, в какой момент вызывается фрагмент кода и вызывается ли вообще. Именно моя помощь позволяет узнать о том, что вызов кода все-таки прошел успешно.

Head First: И во всех этих процедурах отладки вы только временный участник?

Всплывающее окно: Именно так. Но я не возражаю. В конце концов, у меня есть постоянная работа, а помощь в отладке это так, небольшая подработка.

Head First: Спасибо за потраченное на нас время и за объяснение вашей роли в деле борьбы с дефектами кода. Надеюсь, мы еще не раз встретимся.



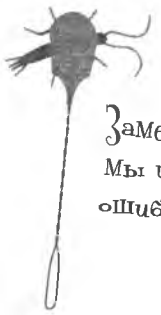
Некорректная логика

Оуэн натолкнулся на логическую ошибку. То есть код совершенно корректен с точки зрения языка JavaScript, но делает совсем не то, что должен делать. В данном случае вместо == введено =, и в результате вместо сравнения двух переменных переменной `callNum` присваивается значение переменной `winningNum`.

С виду корректный код на самом деле несет в себе трудно обнаруживаемую ошибку.

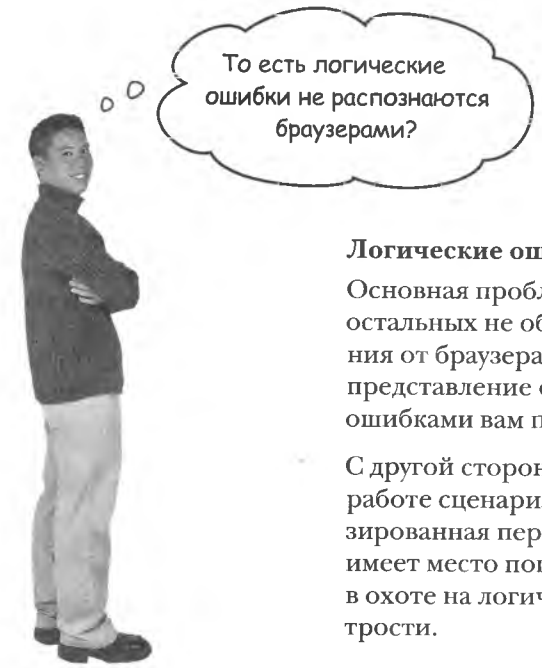
```
...
if (callNum = winningNum) {
...
```

```
...
if (callNum == winningNum) {
...
```



Заменив = на ==, мы избавимся от ошибки #3.

Такие ошибки не обнаруживаются браузерами. Интерпретатор JavaScript выполняет свою работу, возвращая присвоенное переменной `winningNum` значение, которое затем автоматически преобразуется в значение `true` при проверке условия оператором `if`. Другими словами, код написан корректно, просто он делает не то, что нам нужно.



Логические ошибки — это как полет ниже уровня радара.

Основная проблема в том, что логические ошибки в отличие от остальных не обнаруживаются автоматически. Пусть сообщения от браузера и выглядят пугающе, по крайней мере, они дают представление о том, где искать ошибку. А вот с логическими ошибками вам приходится бороться уже без подсказок.

С другой стороны, такие ошибки приводят к некорректной работе сценария. Например, если причиной стала неинициализированная переменная, сообщение «undefined» покажет, что имеет место попытка сослаться на такую переменную. Так что в охоте на логические ошибки также есть свои тонкости и хитрости.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Синтаксическими называют ошибки, нарушающие правила языка JavaScript. Они обнаруживаются браузером.
- Количество открывающих и закрывающих апострофов и кавычек должно совпадать.
- В HTML-атрибутах обработчиков событий, содержащих код JavaScript, нужно использовать как кавычки, так и апострофы.
- Всплывающие окна можно использовать для отслеживания значения переменных.
- В операторе проверки условия часто используют = вместо ==, что становится причиной ошибки.

Часто Задаваемые Вопросы

В: В роли esc-символов могут выступать только кавычки и апострофы?

О: Нет, в JavaScript поддерживаются и другие esc-символы. Например, вот так `\t` в строку вставляется табуляция. Знак переноса строки представлен символом `\n`. А так записывается обратная косая черта `\\`.

Эти символы часто применяются для форматирования текста во всплывающих окнах. Символы `\t` и `\n` выравнивают текст и разбивают его на строки.

В: Почему ограничено использование esc-символов в HTML-атрибутах?

О: Дело в том, что HTML-атрибуты не подчиняются правилам JavaScript в том, что касается символов, ограничивающих значение. Вы можете использовать esc-символы в строках JavaScript, входящих в HTML-атрибут, но те же самые символы при этом могут применяться для обозначения границ атрибута.

С точки зрения HTML атрибут — это значение, которое появляется между кавычками или апострофами.

И ничего больше. Какой бы символ вы ни применяли для обозначения начала атрибута, следующий аналогичный символ будет указывать на его конец. Ведь HTML не обрабатывает esc-символы из JavaScript как значение атрибута. Вы можете вставлять эти символы в HTML-атрибуты при условии, что они не конфликтуют с символами, указывающими на границы атрибута. Ведь значения атрибутов обработчиков событий не интерпретируются, как код JavaScript.

В: Существуют ли отладчики JavaScript, предоставляющие подробный контроль процесса отладки?

О: Да, такие отладчики бывают. И наверное, вам имеет смысл попробовать их в деле. Однако хорошая манера написания кода вкупе с изученными в этой главе методиками отладки способна помочь вам в создании сценариев, не содержащих ошибок.

В: Что именно происходит при ссылке на неопределенную переменную или функцию?

О: Неопределенной называется переменная, которая или не была создана, или же ей забыли присвоить начальное значение. В обоих случаях ее значение воспринимается как неопределенное. Поэтому попытки прочитать его и проделать с ним какие-то операции не имеют смысла, и JavaScript создает сообщение об ошибке. Аналогичная ситуация возникает, если при вызове функции интерпретатор JavaScript не может ее обнаружить. Вызов неопределенной функции не имеет смысла, и вы снова получаете сообщение об ошибке.

В: Почему переменная callNum получала значение `NaN` перед проверкой условия в операторе `if`?

О: Мы этого не знаем. Можно только сказать, что с кодом все еще что-то не в порядке. А значит, нужно продолжать отладку...

Беседа у камина

Синтаксическая и логическая ошибки обожают плохие сценарии.

Синтаксическая ошибка:

Эй, я тебя знаю. Наслышана о твоем коварстве. Но любишь ли ты плохо написанные сценарии так же, как люблю их я?

Нет, нет. Я предпочитаю сценарии, проблемные даже с виду. Я там выделяюсь. Мои сестры разбросаны по всему сценарию, и браузер гарантированно будет возмущен.

Не могу не оценить твое хитроумие, но ведь ты допускаешь работу сценария. А я на это пойти не могу. Сценарий вообще не должен работать. Никак.

Это да. Но к сожалению, причиняемый нами вред весьма ограничен. Портить вебстраницы, не давая им корректно работать весело, но больше никуда у нас доступа нет. Вот если бы нас пустили на жесткий диск, полный важной информации!

Логическая ошибка:

О да! Нет ничего лучше сценария, который на первый взгляд выглядит корректно, но на самом деле переполнен ошибками.

И что тут веселого? Все знают, что удары исподтишка более эффективны. Пусть пользователь думает, что все в порядке, в то время как то там, то здесь начнут возникать небольшие проблемы. И если как следует постараться, пользователь вообще задумается о том, работает ли его браузер.

А ведь это мысль. Жаль, что я сама не додумалась останавливать работу сценария так же, как и ты. Или даже вызывать падение браузера. Вот было бы здорово!

Да, это было бы потрясающе. А ты уверена, что нам никак нельзя туда проникнуть?

Синтаксическая ошибка:

Увы, интерпретатор JavaScript держит нас крепко.

Нет. А как он работает?

И как ты выходишь сухой из воды?

У меня в арсенале тоже есть подобные трюки. Скажем, когда программист забывает поставить в конце оператора точку с запятой. Это не критично для интерпретатора, и все работает, пока каждый оператор находится на своей строке. Но вот попытки «оптимизировать» код и соединить операторы в одну строчку позволяют мне как следует повеселиться!

Но если интерпретатор вдруг что-то замечает, в работу включаюсь я и ошеломляю сообщением об ошибке. Эй, а ведь нам имеет смысл работать в команде. Так мы сможем причинить намного больше вреда.

Я следую за тобой!

Логическая ошибка:

Ничего. Веселиться это нам не мешает. Я рассказывала тебе про маленький фокус с операторами = и ==?

Программист хочет написать == и сравнить два значения, но случайно вводит = и вместо этого выполняет присваивание. А потом сидит часами, пытаясь отыскать корень зла. И даже интерпретатор JavaScript не может ему помочь, потому что технически код совершенно корректен.

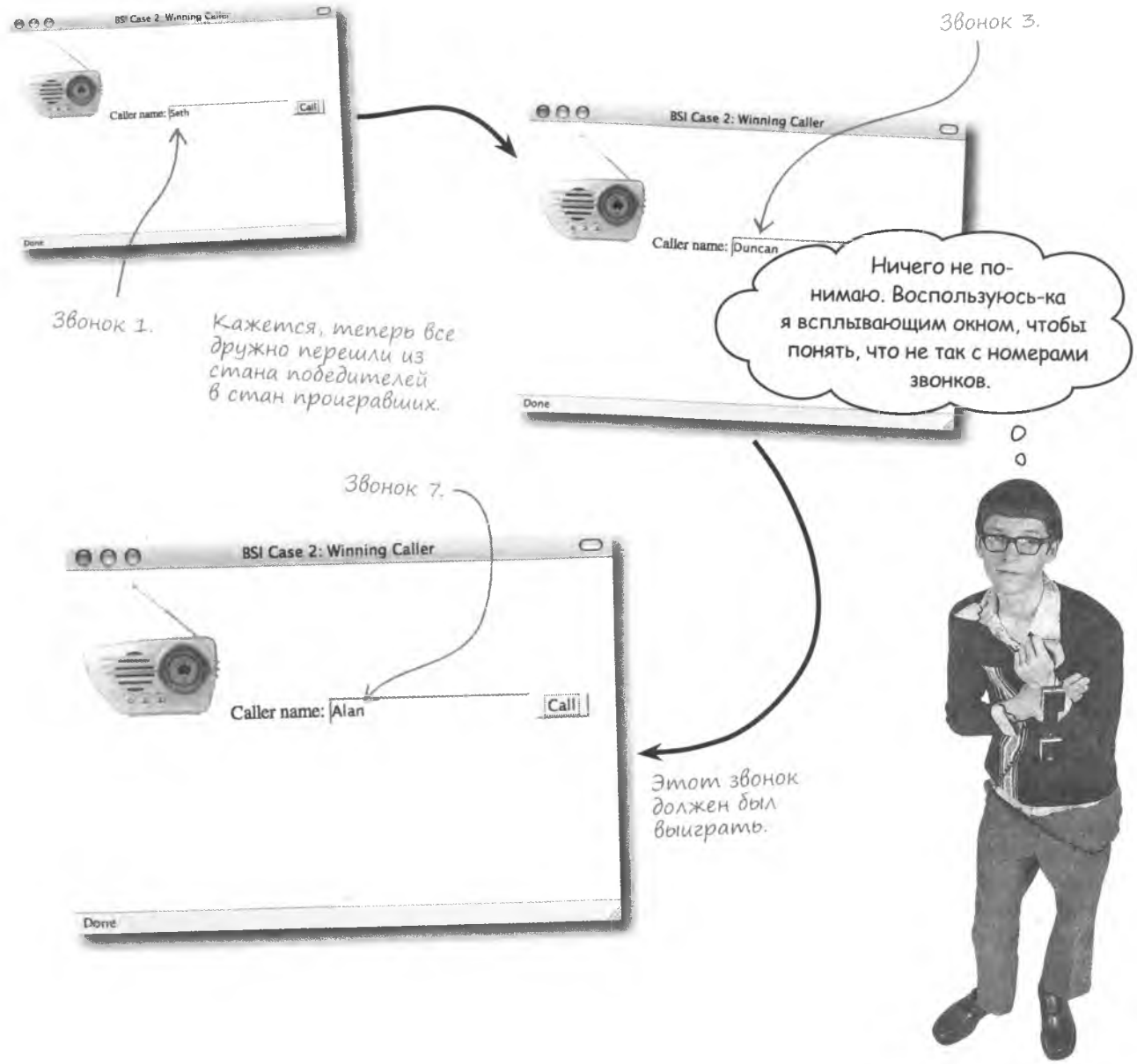
У меня много способов. Это так здорово, не нарушая закона, причинять массу неприятностей.

Ты напомнила мне одну из моих любимых шуток. Обожаю, когда программист решает поменять аргументы уже написанной функции. При этом он часто забывает обновить все вызовы этой функции. Интерпретатор ничего не замечает, и программист получает неожиданные результаты из-за неверного списка аргументов.

Согласна! Давай так и сделаем.

Проигрывают все! (Ошибка #4)

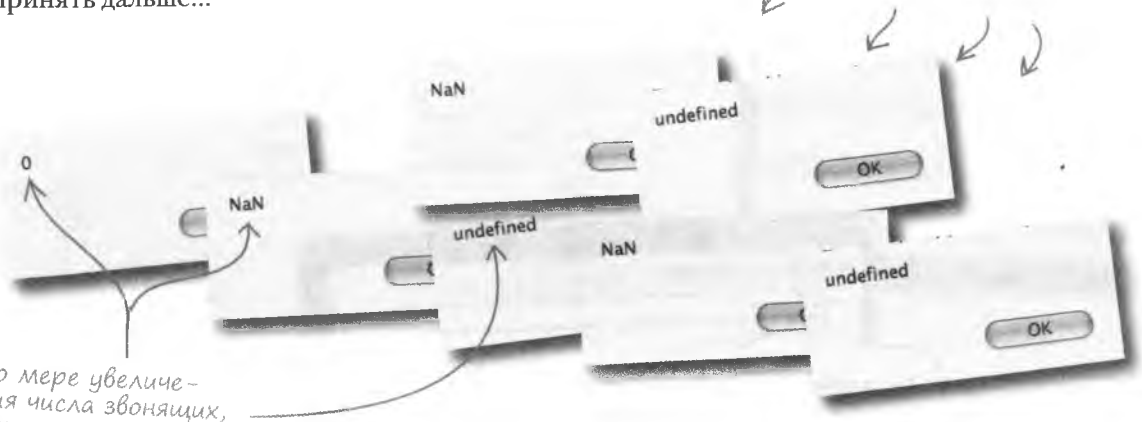
Оуэн начинает понимать, что работа отладчика далеко не так проста, как ему сначала показалось. Исправив логическую ошибку в операторе `if`, он обнаружил, что теперь сценарий вообще не объявляет победителя. Если раньше у нас выигрывал каждый позвонивший, то теперь все проигрывают. Это может негативно повлиять на самооценку слушателей, поэтому ошибку следует устранить как можно скорее.



Атака всплывающих окон

Оуэн пытается проследить за значением переменной `callNum`. Он попытался вставить несколько всплывающих окон в разные части кода, в результате оказался завален окнами с практически бесполезной информацией. И теперь он просто не знает, что предпринять дальше...

При наличии повторяющегося кода попытки отслеживать значение переменной при помощи всплывающего окна приводят к появлению большого числа окон.

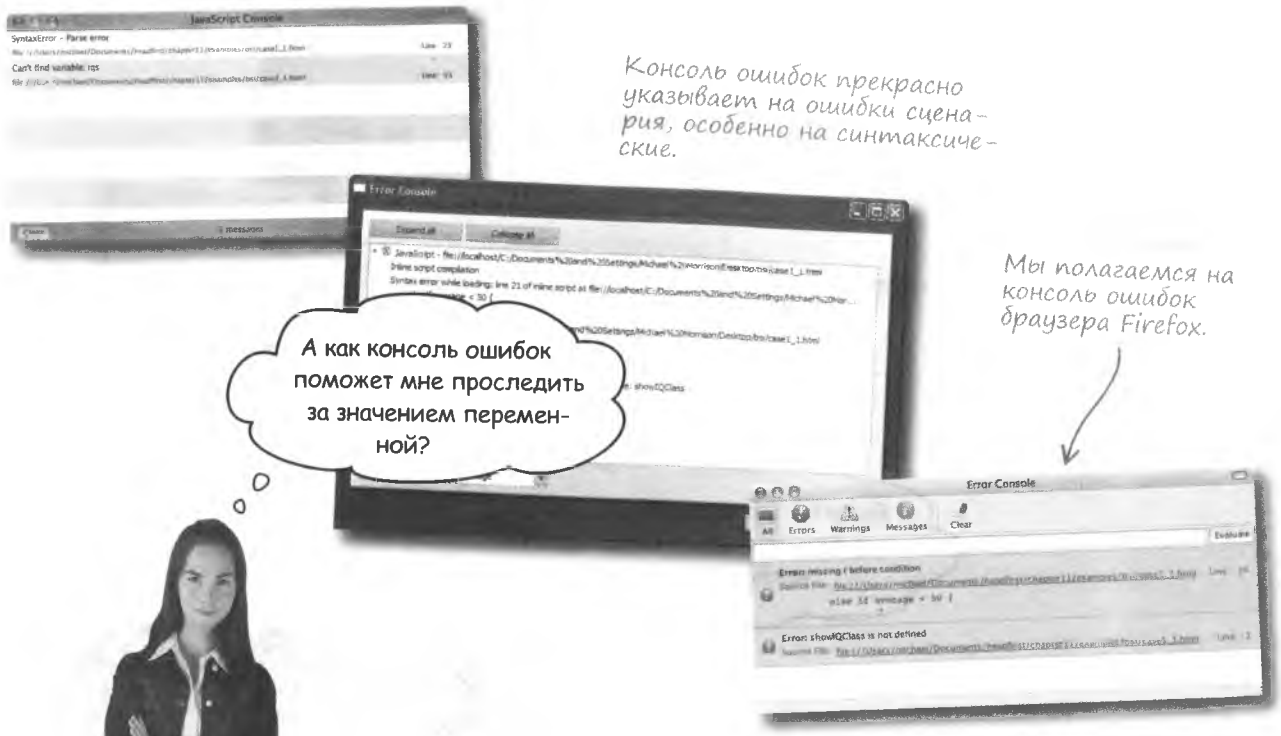


По мере увеличения числа звонящих, переменная callNum демонстрирует полный спектр странных значений.

Было бы здорово, если бы следить за переменными можно было без всплывающих окон...

Отладочная консоль браузера

Большинство браузеров имеют отладочную консоль, на которой отображается информация о содержащихся в сценарии ошибках. Этот инструмент позволяет понять, что именно не так с вашим сценарием. Как вы уже убедились, лучше всего использовать консоль браузера Firefox.

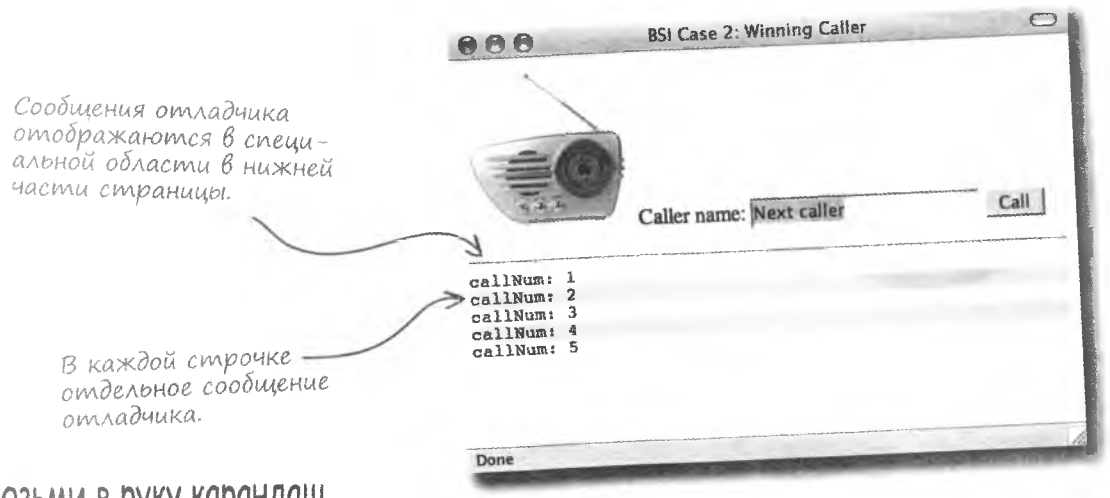


Консоль ошибок не позволяет следить за переменными.

К сожалению, при всех своих достоинствах эту задачу консоль ошибок решить не поможет. Но вам никто не мешает создать свою собственную консоль, предназначенную именно для отслеживания значений переменных.

Пользовательская консоль

Такая задача, как создание собственной консоли, может сначала показаться сложной, но ведь нам всего лишь нужно по запросу отображать определенный текст. Просто отладочная панель должна делать это не в отдельном окне, а непосредственно на странице. Всплывающее окно неудобно потому, что пользователю постоянно приходится щелкать на кнопке OK, чтобы его закрыть.



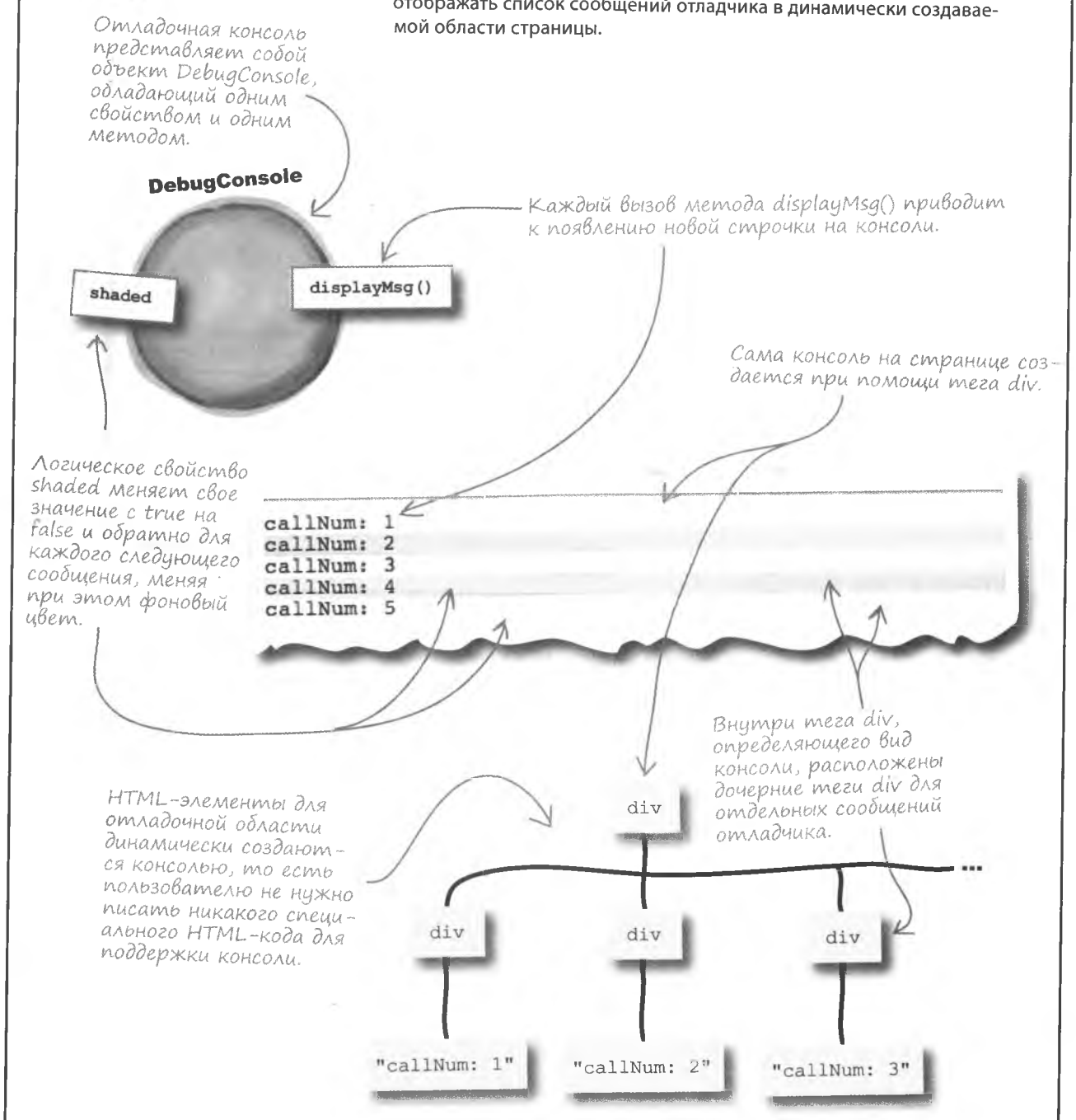
Возьми в руку карандаш

Представьте конструкцию консоли, позволяющей отображать список сообщений отладчика в динамически создаваемой области страницы. Нарисуйте, из каких компонентов она должна состоять и как они соединяются друг с другом и со специальным объектом JavaScript, который и представляет собой консоль.

Возьми в руку карандаш

Решение

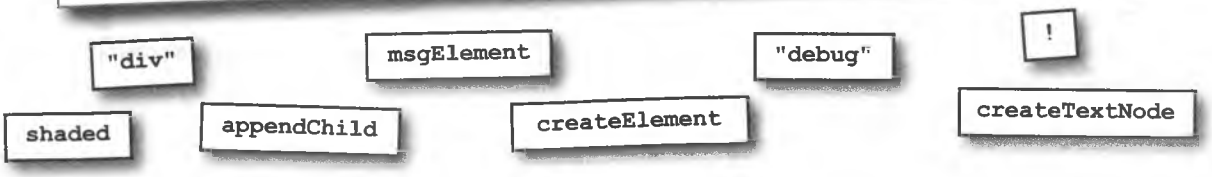
Вот из каких компонентов должна состоять консоль, позволяющая отображать список сообщений отладчика в динамически создаваемой области страницы.



Магниты JavaScript

В коде отладочной консоли отсутствуют несколько фрагментов. Заполните пробелы магнитами таким образом, чтобы получить объект `DebugConsole`.

```
function DebugConsole() {
  // Создание области отладки

  var consoleElem = document.                    (        );

  consoleElem.id = "debug";
  consoleElem.style.fontFamily = "monospace";
  consoleElem.style.color = "#333333";

  document.body.                  (consoleElem);

  consoleElem.                  (document.                          ("hr"));


  // Создание свойства, управляющего изменением фонового цвета

  this.           = false;

}

DebugConsole.prototype.displayMsg = function(msg) {
  // Создание сообщения
  var msgElement = document.createElement("div");

  msgElement.appendChild(document.                          (msg));

  msgElement.style.backgroundColor = this.shaded ? "#EEEEEE" : "#FFFFFF";

  var consoleElem = document.getElementById(          );

  consoleElem.appendChild(                );

  // Изменение значения свойства, управляющего изменением фонового цвета

  this.shaded =      this.shaded;

}
```



Решение задачи с магнитами

Вот каким образом нужно было расположить магниты, чтобы получить код объекта `DebugConsole`.

```
function DebugConsole() {
  // Создание области отладки

  var consoleElem = document.createElement("div");

  consoleElem.id = "debug";
  consoleElem.style.fontFamily = "monospace";
  consoleElem.style.color = "#333333";

  document.body.appendChild(consoleElem);

  consoleElem.appendChild(document.createElement("hr"));

  // Создание свойства, управляющего изменением фонового цвета

  this.shaded = false;
}

DebugConsole.prototype.displayMsg = function(msg) {
  // Создание сообщения
  var msgElement = document.createElement("div");

  msgElement.appendChild(document.createTextNode(msg));

  msgElement.style.backgroundColor = this.shaded ? "#EEEEEE" : "#FFFFFF";

  var consoleElem = document.getElementById("debug");

  consoleElem.appendChild(msgElement);

  // Изменение значения свойства, управляющего изменением фонового цвета

  this.shaded = !this.shaded;
}
```

Тег div консоли добавляется в тело документа, поэтому консоль появляется в нижней части страницы

Первый дочерний элемент консоли — горизонтальная черта, отделяющая ее от остальной страницы.

Начальное значение false соответствует белому цвету фона.

Сообщения добавляются на консоль в дочерних тегах div.

Для простоты восприятия фоновые цвета сообщений чередуются.

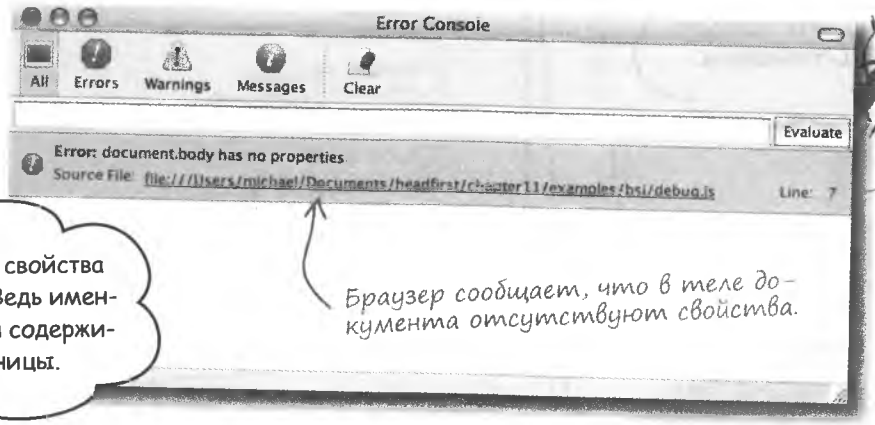
Отладка отладчика

Оуэн не может дождаться момента, когда новая отладочная консоль позволит ему понять, что же не так со сценарием обработки звонков на радио. Он импортирует в страницу файл **debug.js** и создает объект `DebugConsole` в заголовке.

```
<script type="text/javascript">

  // Глобальная переменная отладочной консоли

  var console = new DebugConsole();

  ...
```

Этот код создает в заголовке страницы объект DebugConsole как глобальную переменную.

К сожалению, планам не суждено сбыться. Первая попытка использования консоли показывает, что Оуэн только увеличил число проблем, добавив совершенно новые ошибки.



Куда же пропали свойства тела документа? Ведь именно в них хранится содержимое веб-страницы.

Браузер сообщает, что в теле документа отсутствуют свойства.

Кажется, в сценарии появилась совсем новая ошибка...

Ставшая причиной ошибки строчка кода добавляет к телу документа дочерний узел (`div`), что по идее не должно создавать никаких проблем.

```
document.body.appendChild(console);
```

Получается, что чего-то не хватает и это что-то связано с новым объектом `DebugConsole`.

МОЗГОВОЙ ШТУРМ

Что могло стать причиной отсутствия свойств в теле документа?

Ожидание загрузки

Проблема с отладочной консолью связана с временем загрузки страницы и моментом, когда код сценария получает доступ к ее содержимому.

Заголовок страницы загружается перед ее телом, поэтому содержимое последнего в этот момент недоступно.

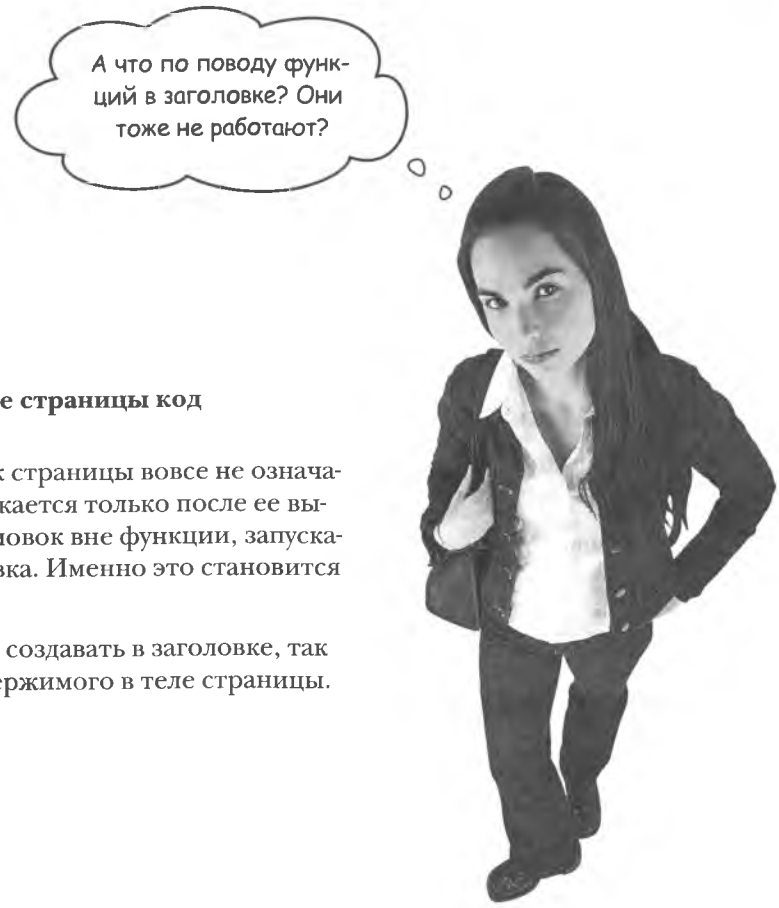
HTML-элементы страницы загружаются только при загрузке тела, после загрузки заголовка.

Код сценария, **запускаемый** в заголовке страницы, не имеет доступа к ее HTML-элементам.

```
<html>
  <head>
    <title>BSI Case 2: Winning Caller</title>

    <script type="text/javascript" src="debug.js"></script>

    <script type="text/javascript">
      // Глобальная переменная отладочной консоли
      var console = new DebugConsole();

      // Общее число звонков
      var callNum = 0;

      function checkWinner(form, caller, winningNum) {
        // Увеличение номера звонка на единицу
        var callNum;
        ++callNum;

        // Проверка победителя
        if (callNum == winningNum) {
          alert(caller + ", caller number " + callNum + "... today\'s winner!");
          form.submit();
        }
        else {
          // Удаление имени звонящего перед следующим звонком
          var callerField = document.getElementById('caller');
          callerField.value = "Next caller";
          callerField.focus();
          callerField.select();
        }
      }
    </script>
  </head>
  <body>
    <form name="callform" action="radiocall.php" method="POST">
      <img src="radio.png" alt="radio" />
      Caller name: <input id="caller" name="caller" type="text" />
      <input type="button" value="Call"
        onclick="checkWinner(this.form, document.getElementById('caller').value, 7)" />
    </form>
  </body>
</html>
```

Код JavaScript в заголовке не имеет доступа к содержимому страницы.

Так как сначала загружается заголовок и только потом все остальное, коду сценария не должен требоваться доступ к HTML-элементам в теле страницы. Это ограничение может показаться странным, но оно имеет смысл, если вспомнить, что практически весь код **запускается** именно в заголовке.



Не весь содержащийся в заголовке страницы код выполняется там же.

Поместить код функции в заголовок страницы вовсе не означает запустить ее там – функция запускается только после ее вызова. А вот код, помещенный в заголовок вне функции, запускается сразу же после загрузки заголовка. Именно это становится причиной проблем.

Объект `DebugConsole` нельзя было создавать в заголовке, так как его конструктор зависит от содержимого в теле страницы.

Возьми в руку карандаш

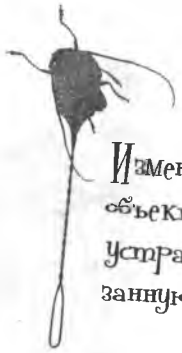
Когда и где следует создать объект `DebugConsole`, чтобы гарантировать его доступ к элементам страницы?

Возьми в руку карандаш

Решение

Вот когда следует создать объект `DebugConsole`, чтобы гарантировать его доступ к элементам страницы.

Браузер указывает на окончание загрузки событием onload. Именно в ответ на данное событие имеет смысл создавать объект DebugConsole. Но объявление переменной console следует оставить в заголовке страницы, ведь объект является глобальным — конструктор объекта будет вызван только после события onload.



Изменив место создания объекта DebugConsole, мы устраним ошибку, связанную с консолью.

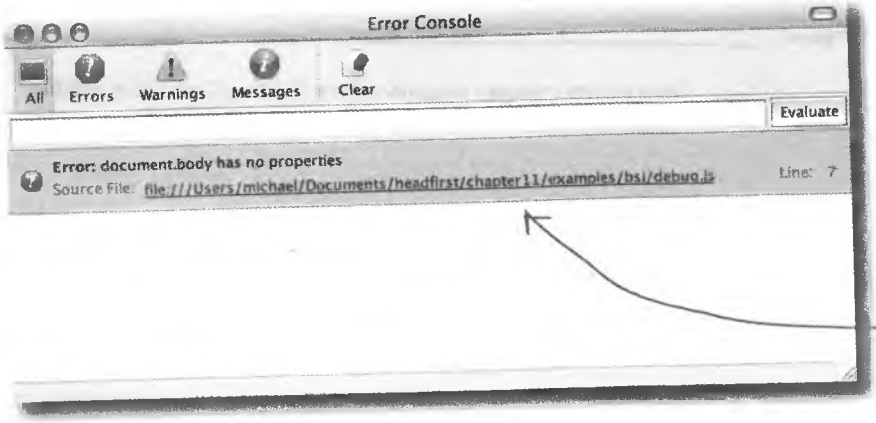
```
<body onload="console = new DebugConsole();">
```

Объект DebugConsole теперь появляется в ответ на событие onload.

Самая противная ошибка

Проблема с незагружающимся телом документа относится к **ошибкам при выполнении** – такие ошибки появляются только при определенных условиях во время работы сценария. Под условиями может подразумеваться и ввод пользователем определенных данных или определенное число повторений цикла. Ошибки при выполнении сложно обнаружимы, потому что их невозможно предсказать. Иногда проблемой является даже воспроизведение ошибки, на которую жалуется пользователь.

Ошибки при выполнении возникают при определенных условиях во время работы сценария.



Проблема с консолью отладки относилась к ошибкам при выполнении и была вызвана попытками доступа к еще незагруженным данным. Подобные проблемы возникают только в процессе работы сценария.

Три самых популярных типа ошибок

Вместе с ошибками при выполнении лидируют в JavaScript еще два типа уже изученных нами ошибок: синтаксические и логические. Они могут появляться в сценариях как вместе, так и по отдельности! Для их успешного распознавания и устранения нужно понимать разницу между ними.

При выполнении

Возникают в процессе работы сценария, например, при вводе пользователем данных, которые невозможно обработать, или при попытке доступа к еще не созданному объекту.

Синтаксические

Возникают при нарушении правил языка JavaScrip и делают код непонятным для интерпретатора.

Логические

Вызваны неверной логикой и в основном связаны с кодом, который должен выполнять одну операцию, но делает что-то другое. Иногда могут быть обусловлены тем, что программист с самого начала неверно понял задачу.

```
<html>
  <head>
    <title>BSI Case 2: Winning Caller</title>

    <script type="text/javascript" src="debug.js"></script>

    <script type="text/javascript">
      // Debug console global variable
      var console = new DebugConsole();

      // Total number of calls
      var callNum = 0;

      function checkWinner(form, caller, winningNum) {
        // Increment the call number
        var callNum;
        ++callNum;

        console.displayMsg("callNum: " + callNum);

        // Check for a winner
        if (callNum = winningNum) {
          alert(caller + ", caller number " + callNum + "... today's winner!");
          form.submit();
        }
        else {
```

Упражнение

Какому типу ошибки соответствует каждое из описаний.

Отсутствие скобок вокруг условия в операторе `if`. .....

Переменной `counter` не присвоено начальное значение 0. .....

Цикл, повторяющийся после последнего элемента массива. .....

Отсутствие закрывающей фигурной скобки у функции. .....

Упражнение
Решение

Вот каким типам ошибок соответствуют данные описания:

Отсутствие скобок вокруг условия в операторе `if`.	*Синтаксическая*
Переменной `counter` не присвоено начальное значение 0.	*Логическая*
Цикл, повторяющийся после последнего элемента массива.	*Выполнения*
Отсутствие закрывающей фигурной скобки у функции.	*Синтаксическая*

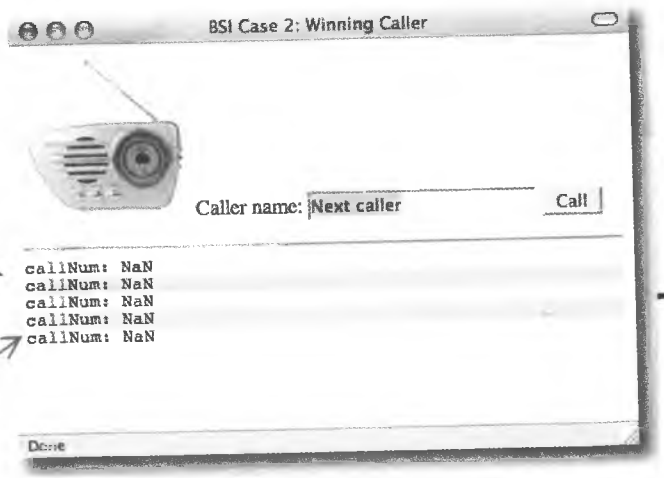
Это не-число

Теперь, когда консоль отладки готова и корректно работает, пришло время выяснить, что же не так с нашей переменной `callNum`. Эту проблему Оуэну давно уже следовало решить. Переменная `callNum` отображается как `NaN`, то есть имеет нечисловое значение. Вот только почему?

```
console.displayMsg("callNum: " + callNum);
```

Эта строчка задает слежение за переменной callNum.

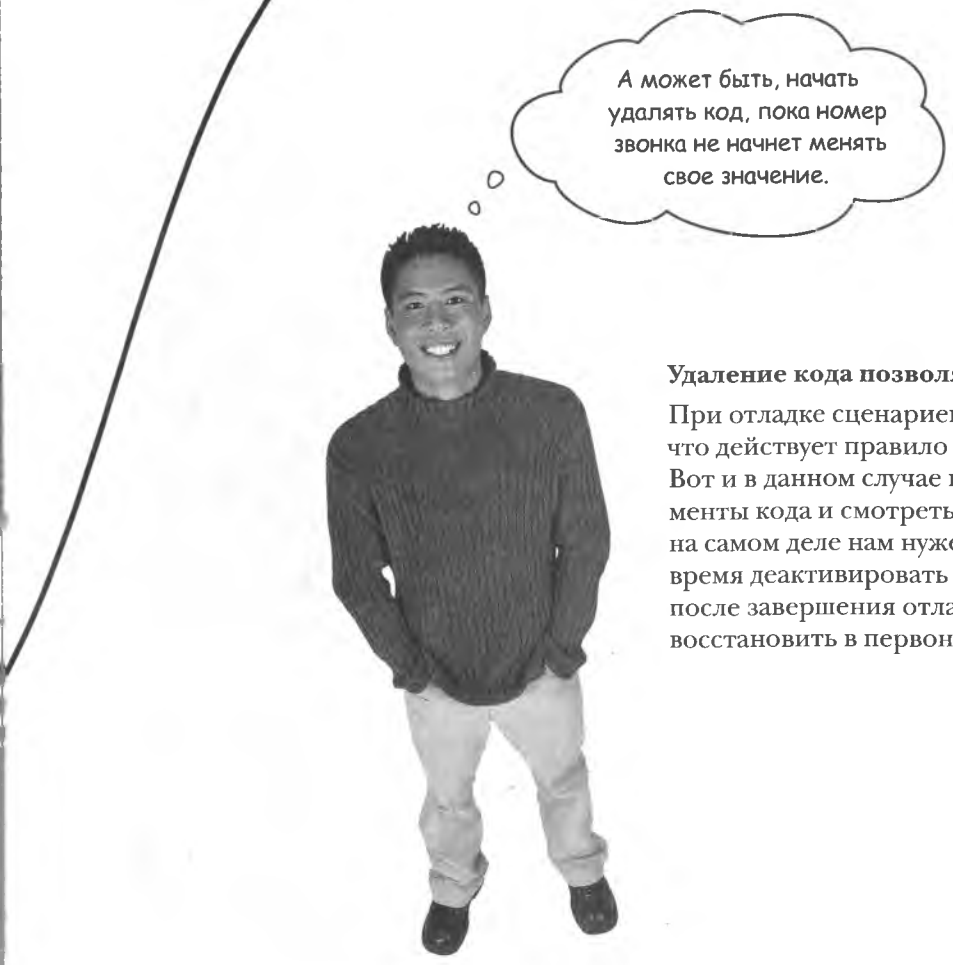
По крайней мере, консоль отладки работает!



Когда слежения недостаточно

Иногда отслеживание значений переменной приносит больше вопросов, чем ответов. Ну почему переменная `callNum` не является числом? Почему ее значение не увеличивается на единицу при каждом следующем звонке? Зачем нужна консоль отладки, если она всего лишь подтверждает то, что мы и так знаем, то есть тот факт, что у нас проблема. И как нам найти причину такого поведения?

А теперь что?



Удаление кода позволяет упростить сценарий.

При отладке сценариев JavaScript бывает и так, что действует правило «чем меньше, тем лучше». Вот и в данном случае имеет смысл удалять фрагменты кода и смотреть, что получается. Хотя на самом деле нам нужен всего лишь способ на время деактивировать код, а не удалять его. Ведь после завершения отладки сценарий желательно восстановить в первоначальном виде.

Комментарии

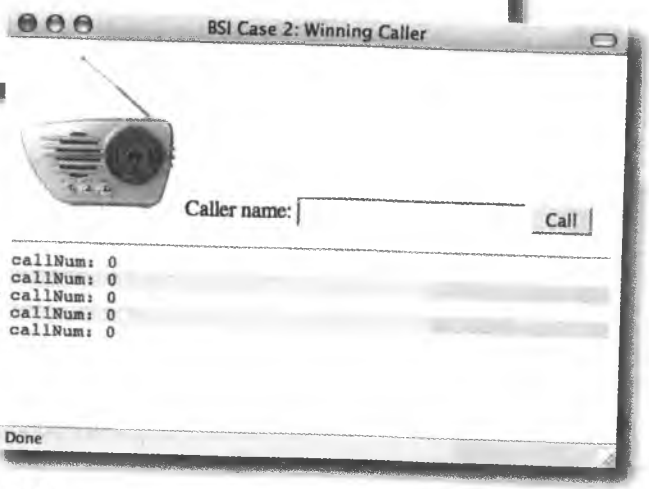
Деактивировать код в процессе отладки легче всего, превратив его в комментарии. Именно этот способ дает возможность изолировать ошибку.

Комментарии позволяют на время деактивировать фрагменты кода.

```
function checkWinner(form, caller, winningNum) {
  console.displayMsg("callNum: " + callNum);

/*
  // Increment the call number
  var callNum;
  ++callNum;

  // Check for a winner
  if (callNum == winningNum) {
    alert(caller + ", caller number " + callNum + "... today\'s
winner!");
    form.submit();
  }
  else {
    // Reset the caller field for the next caller
    var callerField = document.getElementById('caller');
    callerField.value = "Next caller";
    callerField.focus();
    callerField.select();
  }
*/
}
```

Этот многострочный комментарий отключает весь код функции за исключением строчки, отображающей сообщение отладчика.

Переменная callNum теперь имеет значение 0, значит, причина проблемы в отключенном коде.



Теперь номер звонка показан как 0. Значит, именно фрагмент изолированного кода становится причиной появления значения NaN.

МОЗГОВОЙ ШТУРМ

Что произойдет, если вернуть только строчку, увеличивающую значение переменной на единицу?

Проблема кажется решена

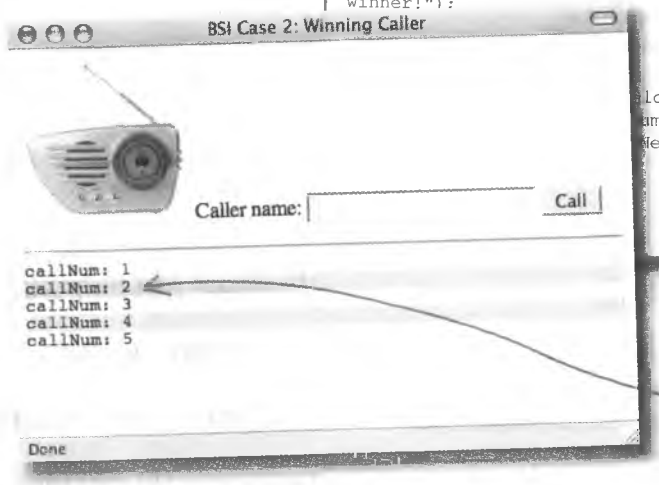
Перейдя от многострочных комментариев к однострочным, можно выборочно подойти к отключению кода. После добавления строчки, увеличивающей значение переменной `callNum` на единицу, все начинает работать как нужно. Значит, проблема содержится где-то в отключенных строчках кода.

Однострочные комментарии позволяют отключать и включать обратно отдельные строчки кода.

```
function checkWinner(form, caller, winningNum) {
  console.displayMsg("callNum: " + callNum);

  // Increment the call number
//  var callNum;
  ++callNum;

  // Check for a winner
//  if (callNum == winningNum) {
//    alert(caller + ", caller number " + callNum + "... today\'s
winner!");
```

Снятие значка комментария с этой строчки наконец-то заставляет переменную callNum корректно функционировать.

```
ld for the next caller
ument.getElementById('caller');
Next caller";
```



Теперь значение переменной callNum увеличивается на единицу при каждом звонке, как и предполагалось с самого начала.

Возьми в руку карандаш

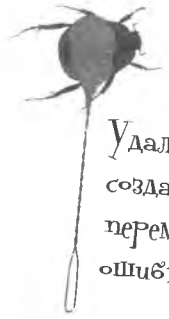
Что именно было не так с переменной callNum и как можно исправить эту ошибку?

Возьми в руку карандаш

Решение

Вот что было не так с переменной callNum.

В функции checkWinner() случайно создана еще одна, локальная переменная callNum. Она «перекрывает» глобальную переменную callNum. Но, так как эта новая переменная не была инициализирована, увеличение ее на единицу и сравнение с номером-победителем дает в результате значение «not a number». Для исправления ошибки достаточно удалить строчку, в которой создается эта переменная.



Удаление строчки, создающей локальную переменную, устраняет ошибку #4.

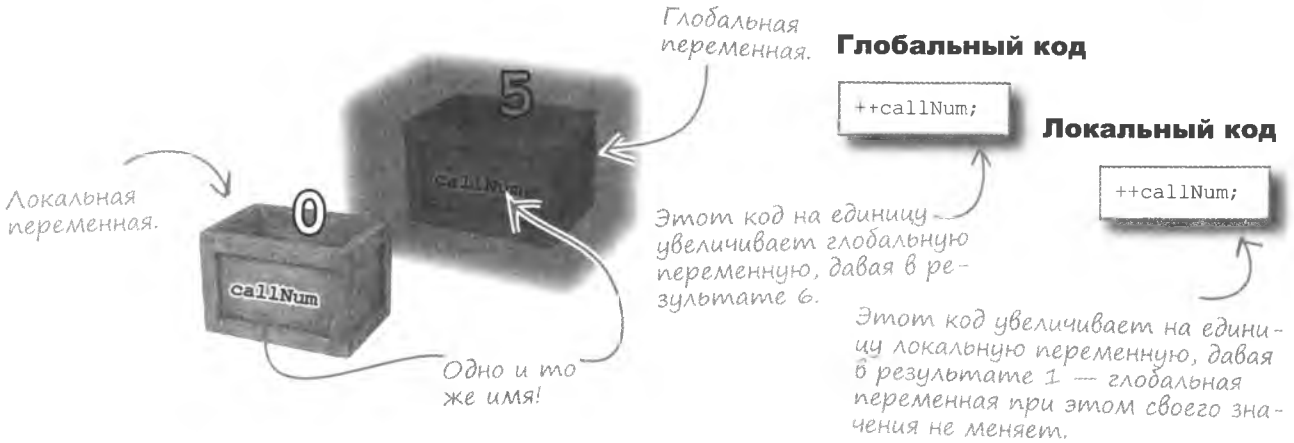
```
// Увеличение номера звонка на единицу
var callNum;
++callNum;
```

Убрав эту строчку, в которой случайно создается локальная переменная callNum, мы заставляем функцию использовать глобальную переменную callNum, которая предполагалась с самого начала.

Дважды объявленные переменные

Причиной ошибки в сценарии обработки звонков стала **дважды объявленная** переменная `callNum`. Такая проблема возникает, когда имя локальной переменной дублирует имя глобальной. При этом все изменения локальной переменной никак не отражаются на состоянии глобальной — она как бы на время скрывается от сценария.

Двойное объявление возникает, когда локальная и глобальная переменные создаются под одним именем.



Часто Задаваемые Вопросы

В: А как понять, какие строчки кода следует превратить в комментарии?

О: Вы научитесь отвечать на этот вопрос по мере роста вашего опыта в отладке сценариев JavaScript. Впрочем, обычно имеет смысл изолировать большую часть или весь код вокруг проблемной области. В случае больших проблем можно начать с превращения в комментарии всего кода. Не забудьте также удалить теги, импортирующие внешний код.

Если вы уже нашли содержащий ошибку фрагмент, используйте другой подход. По очереди превращайте в комментарии строчки, пока ошибка не исчезнет. То есть вместо отключения всего кода и постепенного его включения до **появления** ошибки вы по одной выключаете строчки, пока ошибка не **исчезнет**.

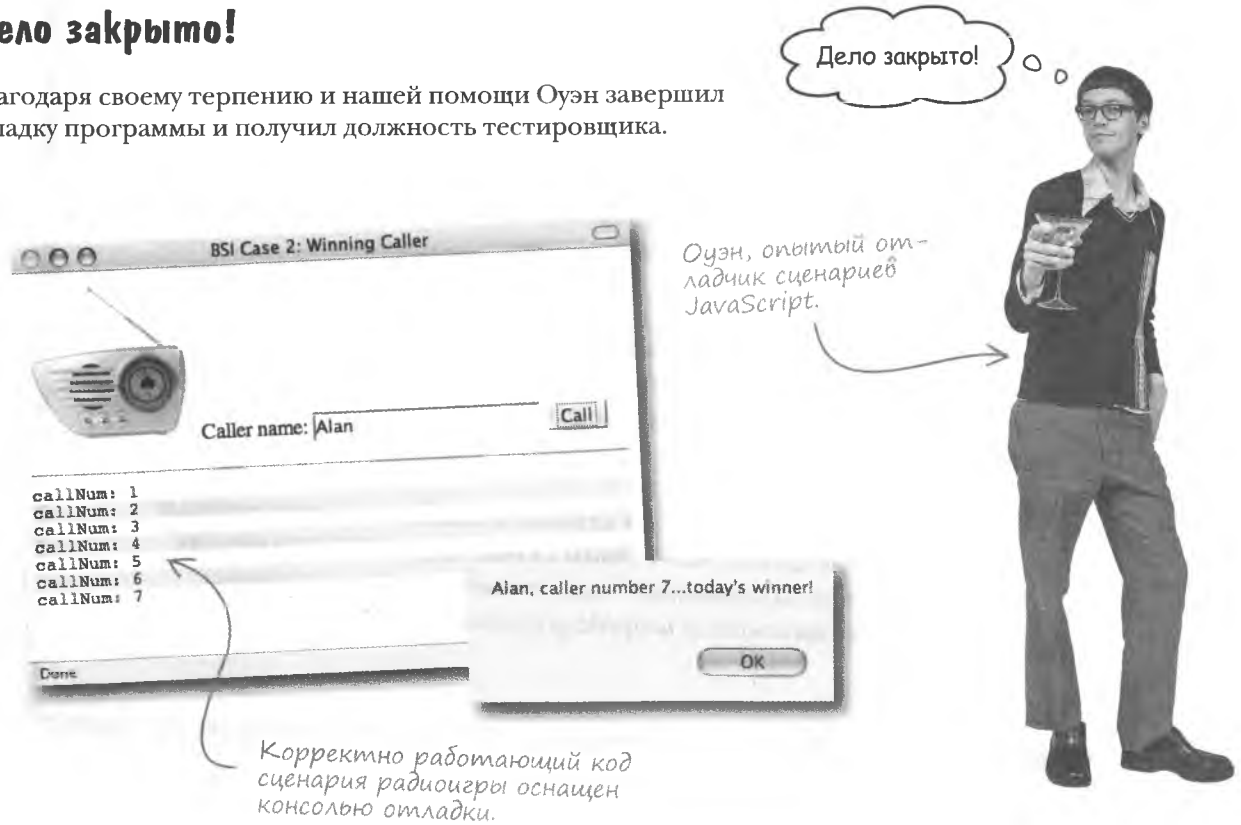
Первый подход применяют в случаях, когда непонятно, где именно локализована ошибка, в то время как второй используется после обнаружения дефектного фрагмента.

В: Можно ли намеренно дважды объявить переменную?

О: Это все равно что спросить, можно ли **намеренно** сломать себе ногу? И ответ на это — нет. То, что вы самостоятельно причиняете себе боль и страдания, не делает их приемлемыми и допустимыми. Кроме того, код зачастую и так содержит достаточно ошибок, чтобы еще вносить их специально. Поэтому двойного объявления переменных следует избегать в любой ситуации.

Дело закрыто!

Благодаря своему терпению и нашей помощи Оуэн завершил отладку программы и получил должность тестировщика.



Контрольная таблица Оуэна

Убедись, что все скобки имеют пару.

Убедись, что блоки кода заключены в скобки, — аккуратные отступы помогают определить границы блока.

Избегай опечаток в именах переменных и функций — в обоих случаях непоследовательность в именах становится причиной проблем.

Будь последователен в использовании кавычек и апострофов и при необходимости аккуратно смешивай их в HTML-атрибутах.

Символы со специальным значением не забывай заменять esc-символами, например (\") для кавычек и (\') для апострофов.

Никогда не используй оператор = там, где нужен оператор ==. Хотя JavaScript и не считает это ошибкой, но код будет работать вовсе не так, как тебе нужно.

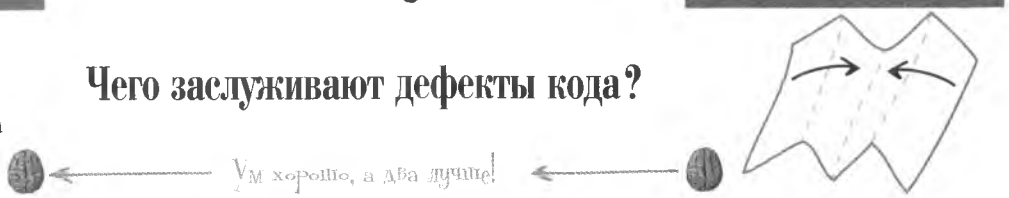
Перед ссылкой на объект убедись, что он уже создан, — это особенно касается элементов веб-страницы, которые создаются только после появления события onload.

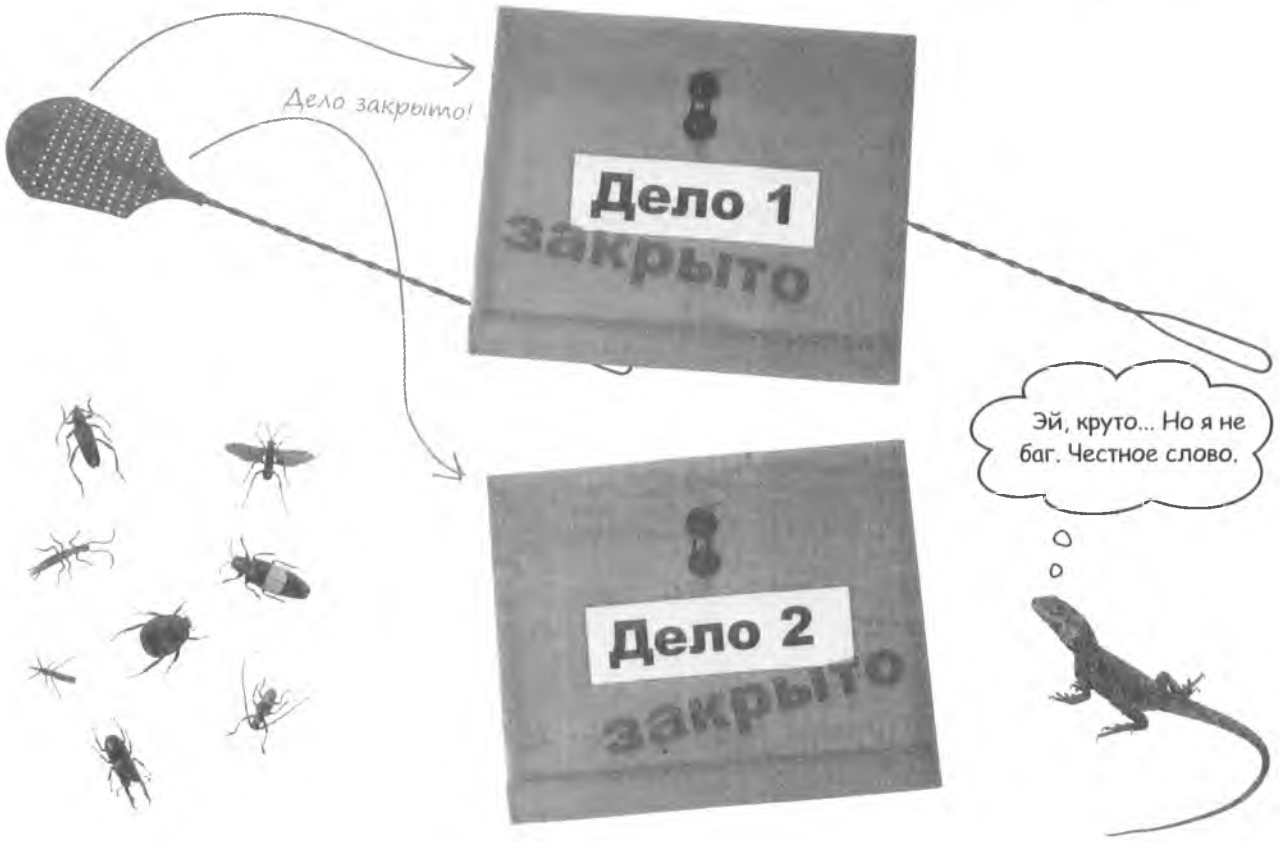
Никогда не присваивай локальным и глобальным переменным одинаковые имена, ведь в этом случае локальная переменная начинает использоваться вместо глобальной, и поведение сценария становится непредсказуемым.

Вкладка

Согните страницу по вертикали, чтобы совместить два мозга и решить задачу.

Чего заслуживают дефекты кода?

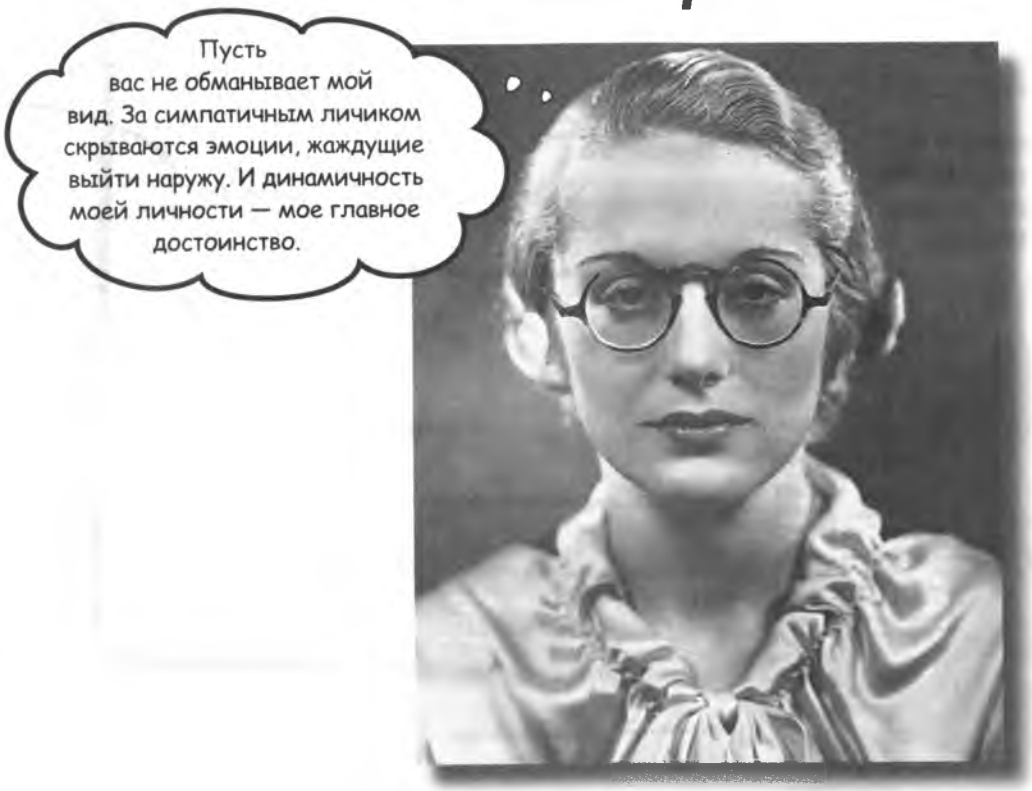




Подставлять другую щеку — неправильный подход, который может истощить ваше терпение. Отвращение к борьбе с дефектами ослабит ваш код — а это уже проблема.

12 Динамические данные

Удобные веб-приложения

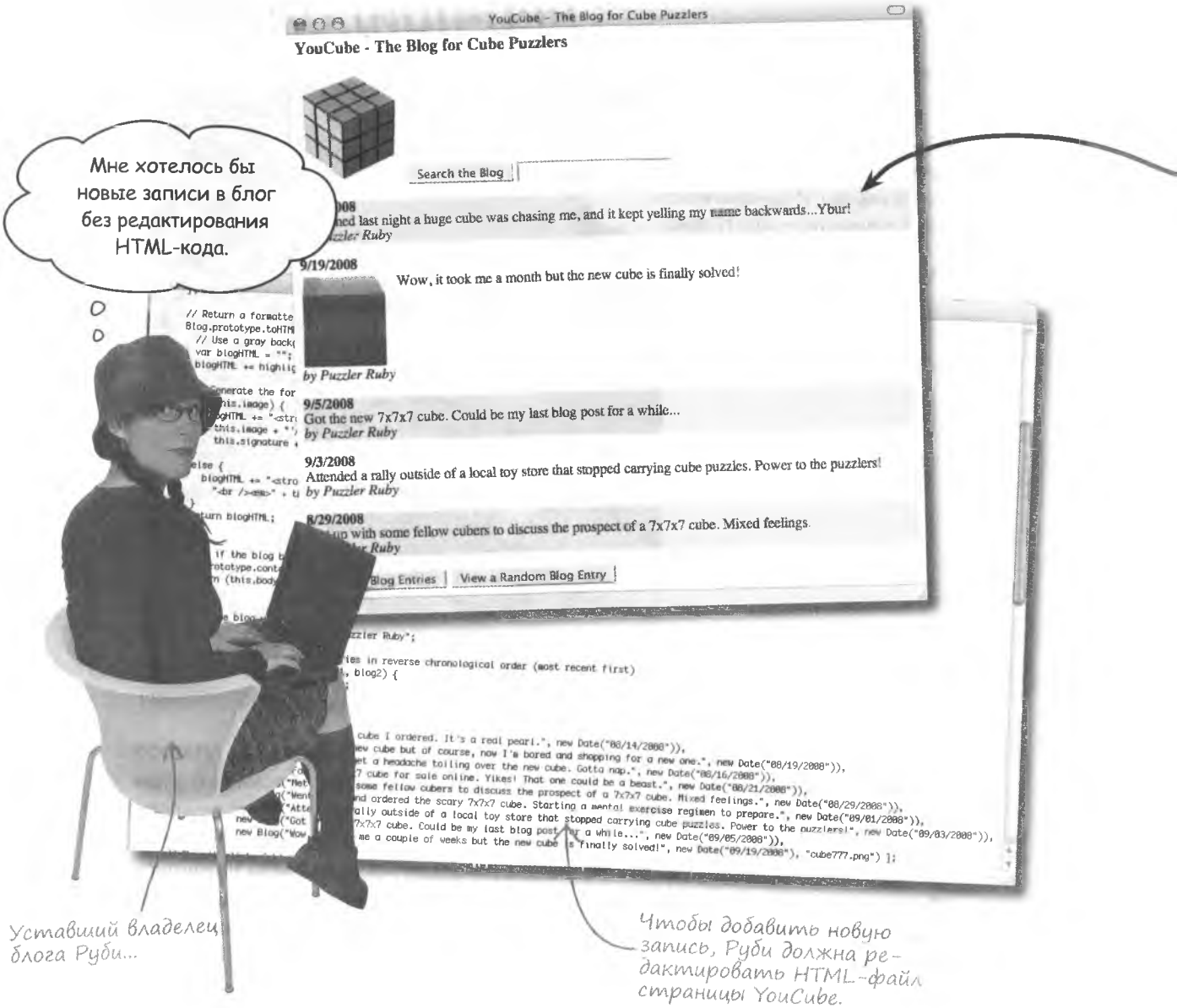


Современный Интернет очень отзывчив, страницы умеют реагировать на каждый каприз пользователя. Именно об этом мечтают многие разработчики. JavaScript играет важную роль в осуществлении этой мечты при помощи технологии **Ajax**, позволяющей эффективно менять «чувствительность» страниц. Благодаря Ajax страницы научились **быстро загружаться и динамически сохранять данные, отвечая на действия пользователя в реальном времени** без необходимости перезагрузки браузера.

Жажда динамических данных

Помните Руби, фанатку головоломок и автора блога? Руби обожает свой блог YouCube, написанный с применением JavaScript, но она устала редактировать HTML-файл при добавлении каждой новой записи. Поэтому она хотела бы отделить записи в блог от HTML-кода, описывающего страницу, чтобы сосредоточиться только на создании новых постов.

Добавление в блог YouCube новых записей не должно требовать редактирования кода страницы.

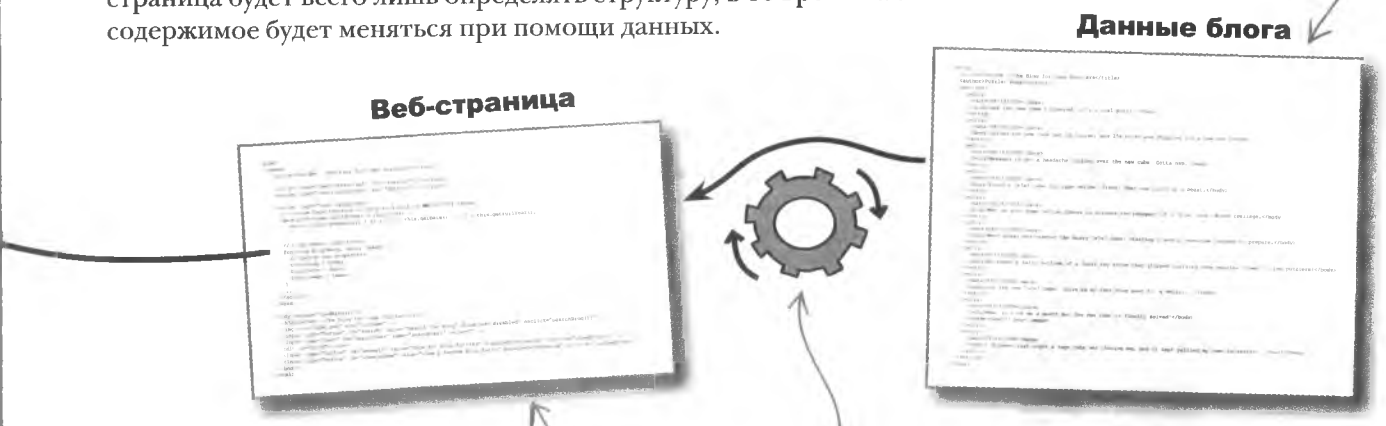


Файлы с новой версией блога Руби можно скачать тут http://www.headfirstlabs.com/books/hfjs/.

Блог, управляемый данными

Руби на пороге новых открытий. Версия блога, разделяющая содержимое и структуру страницы, связана с динамическими данными, которые в реальном времени вставляются в страницу, в то время как сама страница обрабатывается браузером. Соответственно, нам нужен блог, **управляемый данными**. Его веб-страница будет всего лишь определять структуру, в то время как содержимое будет меняться при помощи данных.

Данные блога хранятся в отдельном файле, который можно редактировать независимо от веб-страницы.

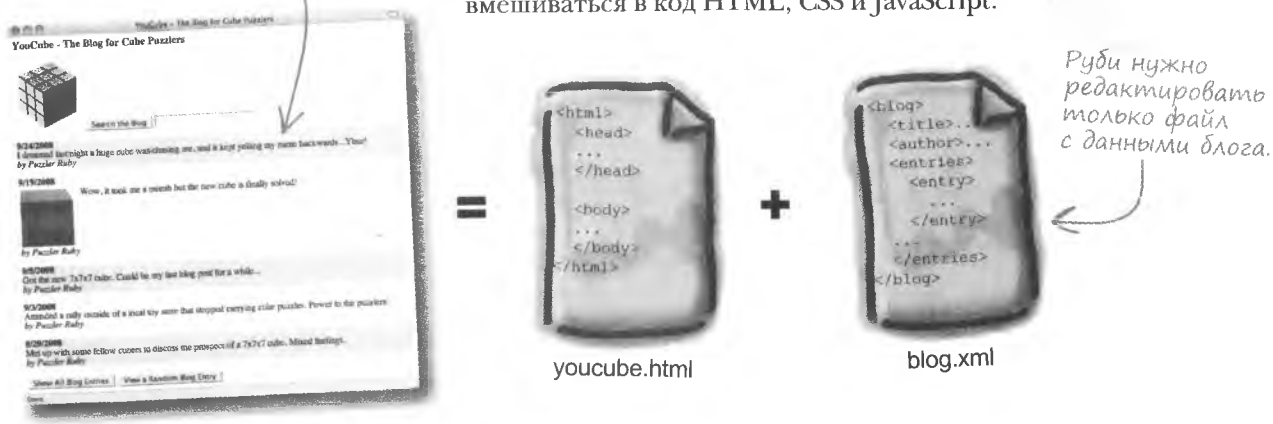


Веб-страница содержит HTML-код, определяющий ее структуру, а также код JavaScript для вставки в страницу динамических данных блога.

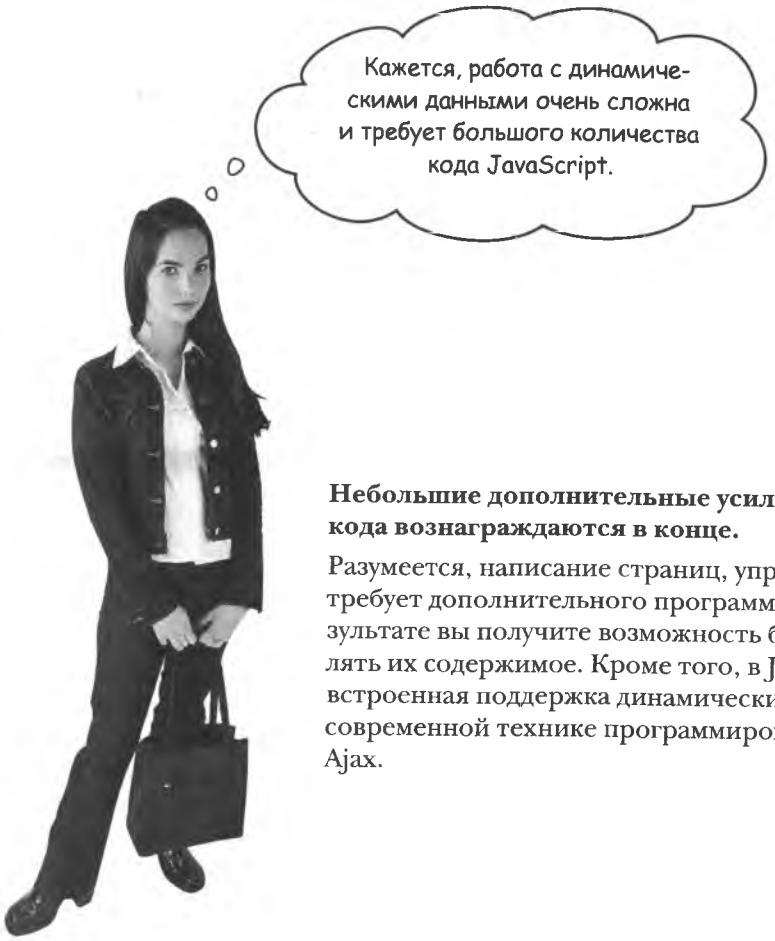
JavaScript отвечает за обработку данных блога и вставку их в веб-страницу.

С помощью JavaScript строчки данных блога динамически вставляются в HTML-код и генерируют окончательный вариант страницы блога YouCube, выглядящий точно так же, как и раньше. Но управляемая данными страница состоит из двух частей: структура страницы и данные блога. Благодаря тому, что записи содержатся в отдельном файле, Руби больше не понадобится вмешиваться в код HTML, CSS и JavaScript.

Записи блога вставляются в страницу из отдельного файла.



Руби нужно редактировать только файл с данными блога.



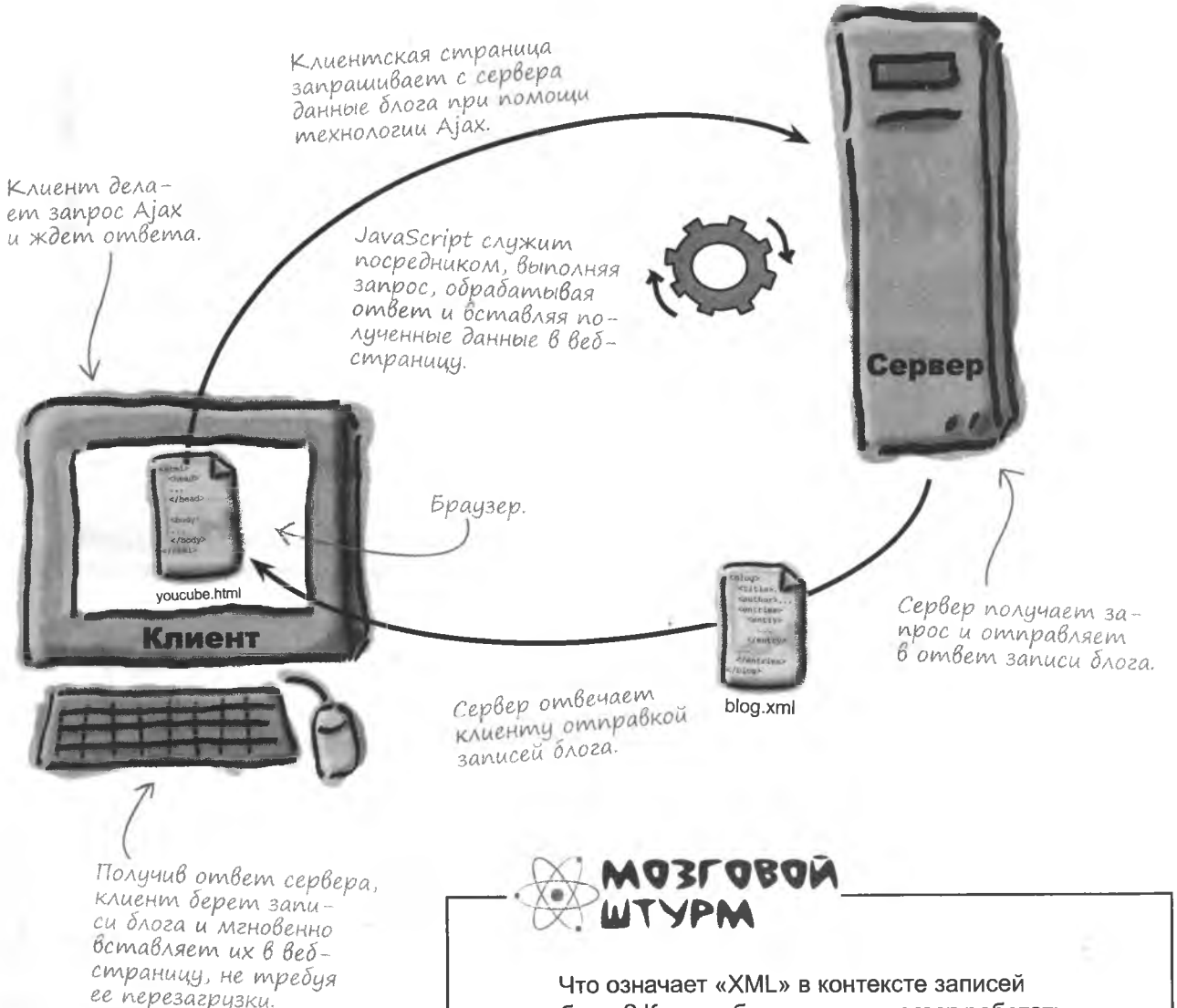
Небольшие дополнительные усилия по написанию кода вознаграждаются в конце.

Разумеется, написание страниц, управляемых данными, требует дополнительного программирования, но в результате вы получите возможность быстро и легко обновлять их содержимое. Кроме того, в JavaScript существует встроенная поддержка динамических данных благодаря современной технике программирования, называемой Ajax.

Ajax как возможность для общения

Ajax дает возможность работать с динамическими данными благодаря постоянному обмену небольшими сообщениями между браузером клиента и сервером. Сценарий может запросить данные с сервера, например набор записей блога, а сервер доставляет их при помощи технологии Ajax. Сценарий же берет полученные данные и динамически вставляет их в страницу.

Ajax позволяет веб-страницам динамически получать данные с сервера.



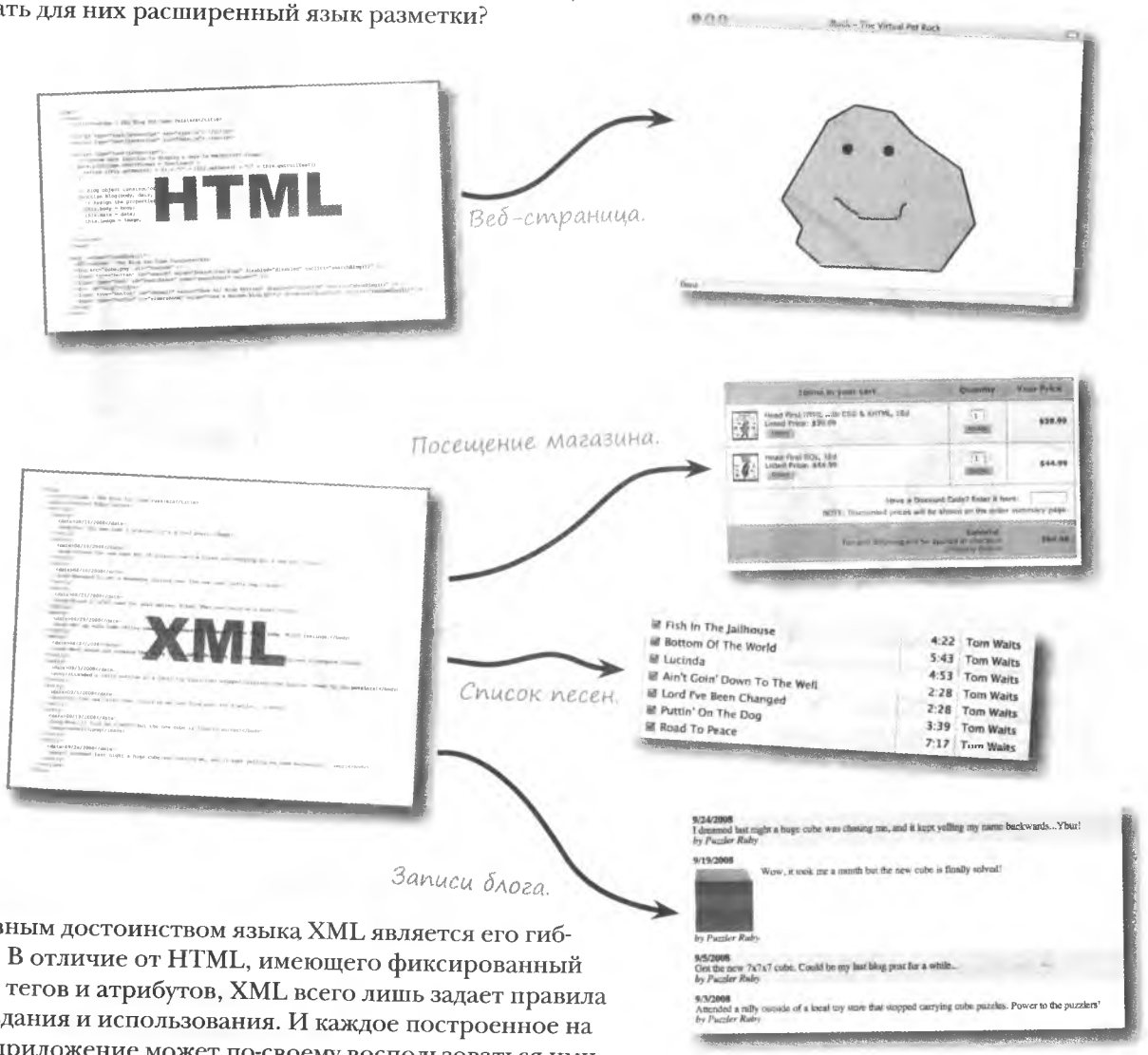
МОЗГОВОЙ ШТУРМ

Что означает «XML» в контексте записей блога? Каким образом он помогает работать с динамическими данными?

HTML для любых целей: XML

XML — язык разметки для форматирования данных любого типа.

Буквы «ML» в аббревиатуре HTML расшифровываются как **«язык разметки»** и указывают на то, что язык HTML использует атрибуты и теги для создания гипертекста («HT»). XML — это еще один язык разметки, используемый для создания всего, что вам будет угодно. Именно на это указывает буква «X»! Ведь существуют различные типы данных, которые имеет смысл сохранить при помощи тегов и атрибутов. Так почему бы не создать для них расширенный язык разметки?



Основным достоинством языка XML является его гибкость. В отличие от HTML, имеющего фиксированный набор тегов и атрибутов, XML всего лишь задает правила их создания и использования. И каждое построенное на XML приложение может по-своему воспользоваться ими для представления определенной информации.

Форматирование с помощью XML

Прелесть языка XML состоит в возможности самостоятельно создавать теги и атрибуты, подгоняя язык под свои цели. Существуют и предустановленные варианты XML, заточенные на решение определенных проблем, и ими вполне можно воспользоваться при необходимости. Но сложно воспротивиться искушению создать свой собственный вариант языка.

Как и код HTML, код XML состоит из иерархии элементов.

```
<movie>
  <title>Gleaming the Cube</title>
  <releaseDate>01/13/1989</releaseDate>
  <director>Graeme Clifford</director>
  <summary>A skateboarder investigates the death of his adopted brother.</summary>
</movie>
```

Каждый фрагмент информации о фильме хранится между своими собственными тегами.

Информация о фильме заключена в тег <movie>.

Несмотря на то что раньше вы никогда не видели такого языка разметки, ведь он полностью создан пользователем, названия тегов дают возможность понять назначение данных. Более того, теги связаны с заключенной внутри них информацией — вполне логично использовать тег `<director>` для хранения сведений о режиссере фильма!

Упражнение

Соедините показанные ниже теги с соответствующим описанием и укажите, к какому языку — HTML или XML — принадлежит каждый тег.

..... `<itunes:author>`	Выделение жирным текста на странице.
..... `<span>`	Заголовок новостей.
..... `<title>`	Элемент управления вводом данных.
..... `<strong>`	Преобразованный в речь текст для звонящих по телефону.
..... `<input>`	Автор цифровой записи iTunes.
..... `<prompt>`	Выравнивание содержимого веб-страницы.

Упражнение Решение

Вот какое описание соответствует каждому из тегов.

XML	<itunes:author>	Выделение жирным текста на странице.
HTML		Заголовок новостей.
XML	<title>	Элемент управления вводом данных.
HTML		Преобразованный в речь текст для звонящих по телефону.
HTML	<input>	Автор цифровой записи iTunes.
XML	<prompt>	Выравнивание содержимого веб-страницы.

XML — это всего лишь текст

Как и HTML, XML-данные — это всего лишь текст, соответственно, хранятся они в текстовых файлах. Но файлы XML имеют расширение .xml, в то время как для HTML-файлов используется расширение .html или .htm.

XML + HTML = XHTML

Несмотря на различные расширения файлов, XML и HTML связаны друг с другом, и эта связь называется XHTML. Это современная версия HTML, следующая более строгим правилам языка XML. Например, каждому открывающему тегу должен соответствовать закрывающий. В HTML это правило для таких тегов, как, например, <p>, соблюдать не обязательно.

XHTML — это версия HTML, придерживающаяся более строгих синтаксических правил XML.

HTML

```
This is a paragraph of text in HTML.<p>
```

В HTML тег <p> часто используется сам по себе, указывая на начало или конец параграфа.

XHTML

```
<p>This is a paragraph of text in XHTML.</p>
```

В XHTML каждому открывающему тегу должен соответствовать закрывающий.

Другим важным отличием HTML и XHTML являются пустые теги, такие как
, которые теперь комплектуются пробелом и косой чертой, указывающей на отсутствие закрывающего тега.

HTML

```
This is just a sentence.<br>
```

Вот так в HTML выглядит тег переноса строки.

XHTML

```
This is just a sentence.<br />
```

В XHTML этот тег содержит еще и пробел с косой чертой на конце.

В отличие от HTML в XHTML в кавычки заключаются все значения атрибутов.

HTML

```
<a href=home.html>Go home</a>
```

Отсутствие кавычек вокруг значения атрибута нарушает правила XHTML.

XHTML

```
<a href="home.html">Go home</a>
```

Все значения атрибутов в XHTML должны быть заключены в кавычки.

Хотя для нужд Руби XHTML не требуется, он прекрасно иллюстрирует некоторые синтаксические правила языка XML, на пользовательской версии которого и будет написан новый вариант блога.

Беседа у камина

HTML и XML конкурируют за данные.

Последняя версия HTML была переформулирована с помощью XML и получила название XHTML.

XHTML:

Знаешь, ты для меня все запутал. Я был опорой Интернета, а теперь из-за тебя люди стали путаться.

Но вот со мной тебе не повезло, потому что браузеры до сих пор отображают только HTML-код. А что делать с тобой, они не представляют.

О чем ты говоришь? Кого заботят данные, не имеющие внешнего вида?

И все это можно увидеть только благодаря мне!

Вот как... То есть ты утверждаешь, что мы фактически работаем в паре?

Приятно было это узнать!

XML:

Не моя вина, что ты думаешь только о вебстраницах. Я просто имею более широкие взгляды и потому представляю данные любого типа.

Вот такой я загадочный парень. Я — существо без лица. Я не имею внешнего вида. И когда мне нужно показать себя, прибегаю к твоим услугам.

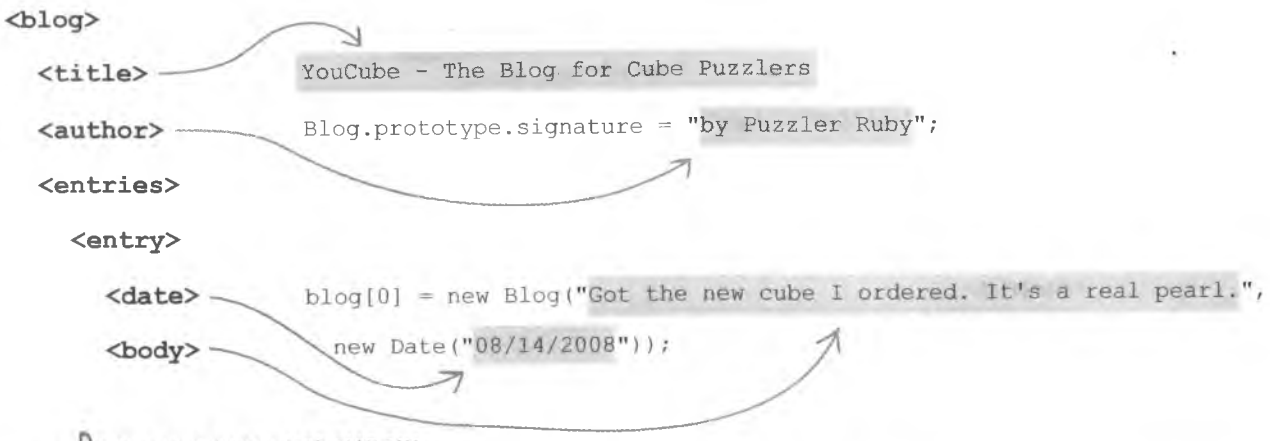
Вот только не надо выходить из себя. Весь мир давно работает с данными, которые часто не видны. Банковские транзакции, политические опросы, погодные условия и многое другое.

Это так, но вот только каким образом это все хранится перед тем, как отобразится браузером? Совсем не в виде абзацев и таблиц. Они сохраняются с моей помощью, плюс я облегчаю обработку данных.

Именно так! Я не имею понятия о том, как выглядят данные, зато фокусируюсь на их значении. И пока люди пользуются браузерами, для отображения данных мне нужен ты.

XML и данные блога YouCube

XHTML является замечательным приложением XML, быстро улучшающим структуру и надежность веб-страниц. Но для блога YouCube Руби понадобится пользовательский вариант языка XML, моделирующий нужные ей данные. Давайте посмотрим, как представить записи блога при помощи тегов XML.



Возьми в руку карандаш

Изобретите ваш вариант языка XML для хранения записей блога и используйте его для написания примера кода. Вам потребуются такие элементы, как заголовок (title), дата (date), автор (author) и собственно запись (entry).

```
<blog>
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
</blog>
```

Возьми в руку карандаш

Решение

Вот вариант языка XML для хранения записей блога.

Блог находится между тегами <blog>.

```
<blog>
  <title>YouCube - The Blog for Cube Puzzlers</title>
  <author>Puzzler Ruby</author>
  <entries>
    <entry>
      <date>11/14/2007</date>
      <body>Got the new cube I ordered. It's a real pearl.</body>
    </entry>
  </entries>
</blog>
```

Тег <title> содержит заголовок блога.

Угадайте, как называется тег, содержащий имя автора блога?

Набор записей хранится в теге <entries>.

Каждая отдельная запись заключена в тег <entry>.

И дата, и тело записи блога заключены в свои собственные теги.

Часто Задаваемые Вопросы

В: А почему не хранить записи блога в виде обычного, неформатированного текста?

О: Вы можете хранить их в таком виде, но потом будет неимоверно сложно разбить информацию на набор записей, каждая из которых связана с отдельной датой. XML структурирует данные предсказуемым образом, позволяя легко выделять отдельные записи, не говоря уже о заголовке или имени автора блога.

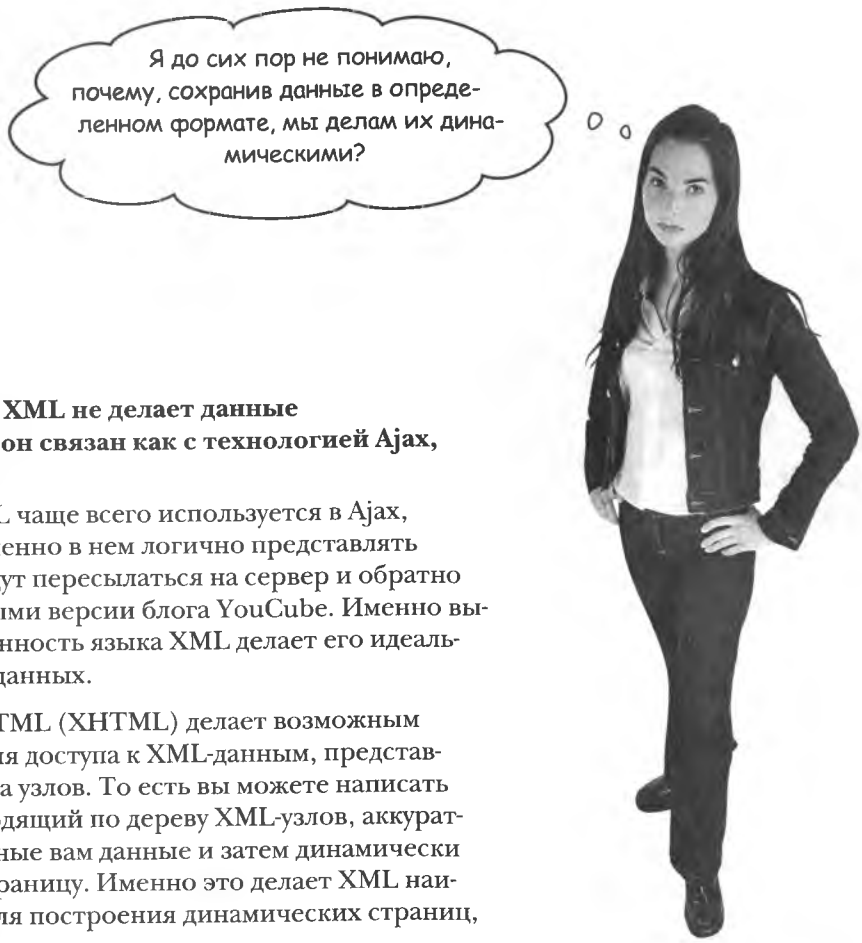
В: Насколько для записи данных блога в формате XML нужен тег `<entries>`?

О: НЕ являясь обязательным, этот тег делает данные более структурированными и легкими для понимания. Например, в предыдущем примере без тега `<entries>` оказалась бы невозможной поддержка множественных тегов `<entry>`, остались бы только теги `<title>` и `<author>`. Тег `<entries>` предполагает наличие набора записей и делает более очевидным способ использования данных.

В: Как XML связан с Ajax?

О: Название Ajax образовалось как сокращение от «**A**synchronous **J**avaScript **A**nd **X**ML» (асинхронный JavaScript и XML), так что XML непосредственно связан с Ajax. В настоящее время роль технологии Ajax расширена настолько, что для нее не всегда требуется XML. Но именно этот язык формирует основу большинства Ajax-приложений, предоставляя замечательный механизм моделирования данных.

Как вы увидите в этой главе, связь между Ajax и XML обнаруживается также в способе поддержки технологии Ajax языком JavaScript. В качестве формата данных для обработки запросов Ajax JavaScript не ограничен языком XML, но именно этот язык упрощает обработку этих запросов. Так что хотя некоторые пуристы и утверждают, что XML никак не связан с Ajax, на самом деле они идут рука об руку.



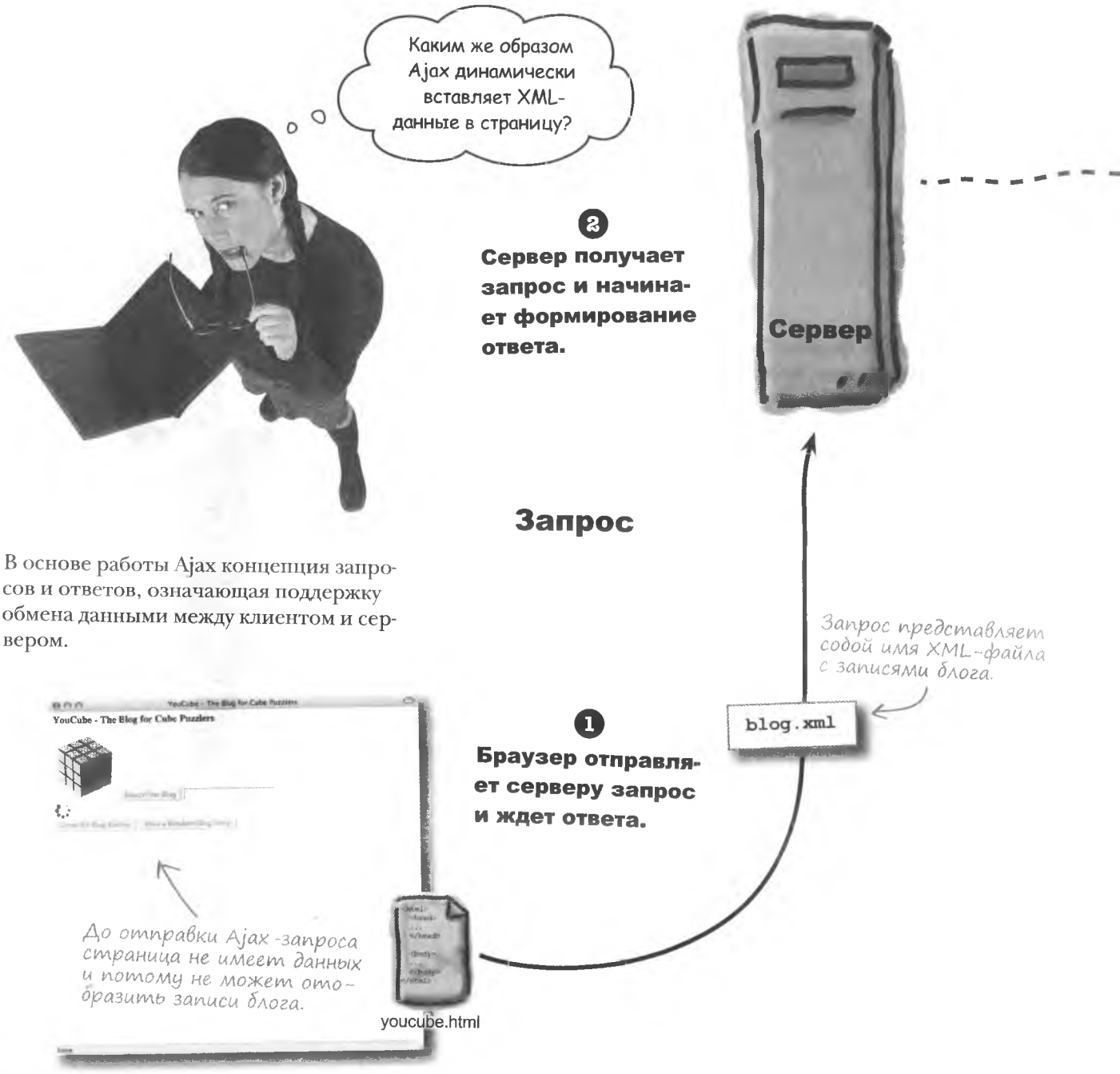
Сам по себе формат XML не делает данные динамическими, но он связан как с технологией Ajax, так и с DOM.

Именно формат XML чаще всего используется в Ajax, и, следовательно, именно в нем логично представлять данные, которые будут пересылаться на сервер и обратно в управляемой данными версии блога YouCube. Именно высокая структурированность языка XML делает его идеальным для пересылки данных.

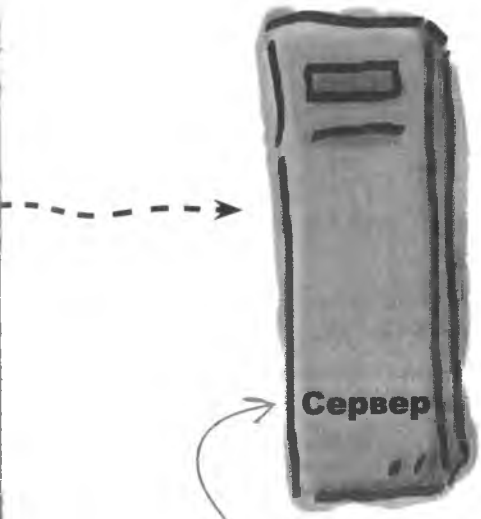
А сходство XML с HTML (XHTML) делает возможным применение DOM для доступа к XML-данным, представленным в виде дерева узлов. То есть вы можете написать код JavaScript, проходящий по дереву XML-узлов, аккуратно изолировать нужные вам данные и затем динамически встроить их в веб-страницу. Именно это делает XML наилучшим решением для построения динамических страниц, управляемых данными.

Добавим к блогу Ajax

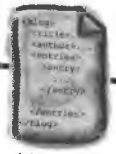
Имея на руках документ XML с записями блога, Руби готова динамически вставить их на страницу YouCube при помощи Ajax.



3 **Сервер создает ответ для браузера, упаковав данные в файл blog.**



В ответ возвращается содержимое XML-файла с записями блога.

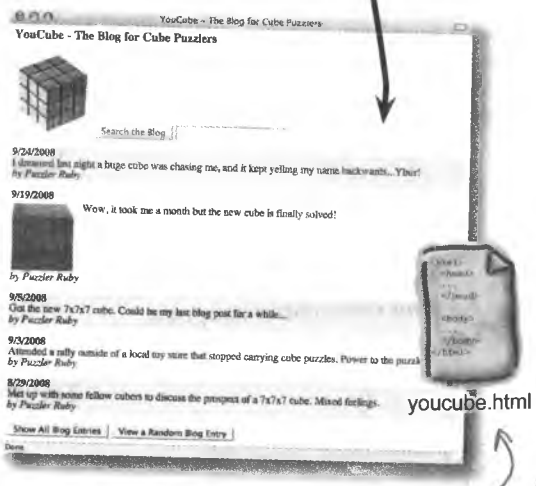


Ответ

Иногда для обработки запросов Ajax и подготовки ответных данных требуется сценарий на стороне сервера.

4 **Браузер распаковывает полученные XML-данные и аккуратно вставляет их в страницу.**

После вставки XML-данных в HTML-код страницы они становятся видимы в браузере.



На странице работает код JavaScript, отвечающий за создание запроса Ajax и обработку ответа.

МОЗГОВОЙ ШТУРМ

Какой именно код JavaScript отвечает за обработку запросов Ajax и ответов на них?

Интерфейс XMLHttpRequest

В JavaScript существует встроенный объект `XMLHttpRequest`, инициирующий запросы Ajax и обрабатывающий ответы. Он достаточно сложен и содержит набор методов и свойств, которые и осуществляют поддержку технологии Ajax.

Объект XMLHttpRequest имеет и другие свойства и методы, но тут перечислены самые важные из них.

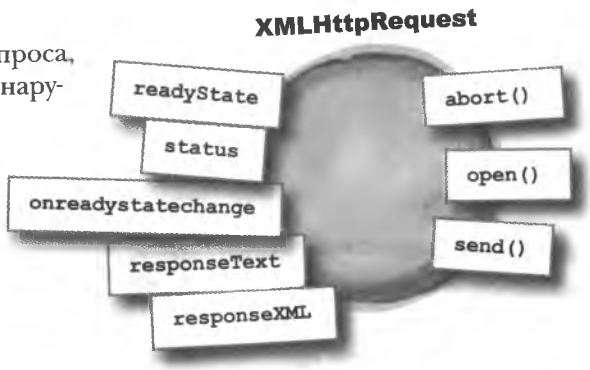
readyState

Текущее состояние объекта: 0 (неинициализирован), 1 (открыт), 2 (отправка данных), 3 (получение данных), 4 (данные загружены).

status

HTTP-код статуса запроса, например 404 (не обнаружено) или 200 (OK).

Эти свойства определяют, был ли получен корректный ответ на запрос Ajax.



abort()

Отмена текущего запроса.

Этот метод применяется, чтобы отменить сделанный запрос.

open()

Подготовка запроса путем указания его типа, URL и других параметров.

send()

Отправка запроса на сервер для обработки.

Эти два метода используются в паре для подготовки запросов Ajax и их отправки на сервер.

onreadystatechange

Ссылка на функцию, вызываемую при изменении состояния запроса.

Это уникальное свойство содержит ссылку на специальный обработчик события, который вызывается при изменении состояния Ajax-запросов — именно он обрабатывает ответы.

responseText

Текст ответа на запрос.

responseXML

Текст ответа на запрос в виде дерева XML-узлов.

Эти два свойства хранят данные, полученные с сервера в ответ на запрос.

Применение XMLHttpRequest

Объект XMLHttpRequest — потрясающе мощный и удивительно гибкий. Но мощность и гибкость сопровождаются сложностью, в результате даже базовые запросы Ajax требуют изрядного количества кода JavaScript. Частично за это в ответе несовместимость различных браузеров. Удручает также возможность легко запутаться в параметрах настройки объекта даже в случае, когда вам требуется только быстрое динамическое перемещение данных.

Объект XMLHttpRequest является мощным, но сложным в использовании инструментом.

Рассмотрим в качестве примера код создания объекта XMLHttpRequest, работающего с различными браузерами:

```
var request = null;
if (window.XMLHttpRequest) {
  try {
    request = new XMLHttpRequest();
  } catch(e) {
    request = null;
  }
// Пробуем версию ActiveX (IE)
} else if (window.ActiveXObject) {
  try {
    request = new ActiveXObject("Msxml2.XMLHTTP");
  // Пробуем объект ActiveX более старой версии IE
  } catch(e) {
    try {
      request = new ActiveXObject("Microsoft.XMLHTTP");
    } catch(e) {
      request = null;
    }
  }
}
```

Различные подходы к созданию объекта XMLHttpRequest, так как он по-разному поддерживается разными версиями браузера IE.

Оператор try-catch является усовершенствованным механизмом обработки ошибок исполнения в JavaScript.

Подсказка

Создание объекта XMLHttpRequest затруднено тем, что для каждого браузера требуется своя реализация. К счастью, набор методов и свойств одинаков для всех браузеров — учитывать разницу в браузерах требуется только при создании объекта.

Теперь, когда объект XMLHttpRequest готов, нужно задать функцию обработки запроса и затем создать сам запрос.

```
request.onreadystatechange = handler;
request.open(type, url, true); // всегда асинхронный (true)
```

Эта функция вызывается после ответа сервера на запрос.

Открытый запрос готов к отправке, здесь же указывается его тип (GET или POST).

При открытии запроса следует указывать его тип ("GET" или "POST"), адрес URL сервера, а также является ли этот запрос асинхронным или нет. Асинхронные запросы выполняются в фоновом режиме, не заставляя сценарий ждать, именно поэтому почти все запросы Ajax являются асинхронными.

Получение и отправка

Тип запроса Ajax крайне важен, поскольку отражает не только содержимое запроса, но также и его **назначение**. Первый тип, его еще принято называть **методом** запроса, называется `GET` и применяется для получения данных с сервера, не влияя на них. Второй тип запроса, `POST`, обычно связан с отправкой данных на сервер, что приводит к некоторому изменению его состояния.

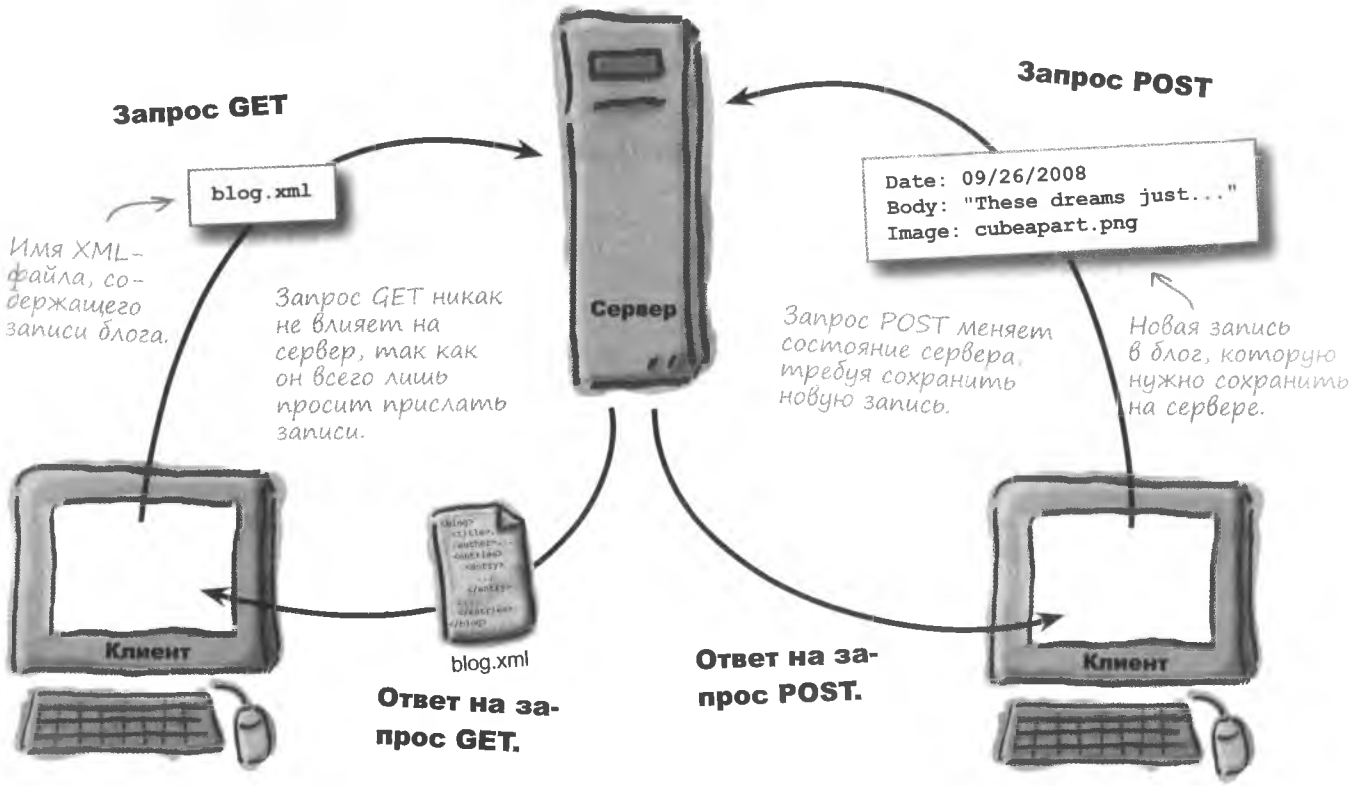
В Ajax используются те же два типа запросов, которые применяются для отправления HTML-форм, то есть GET и POST.

GET

Используется для получения данных и не меняет ничего на сервере. Небольшие фрагменты данных могут быть при необходимости отправлены на сервер в виде URL. Запрос `GET` прекрасно подходит для получения записей блога из хранящегося на сервере XML-файла.

POST

Отправляет на сервер данные, инициируя изменения, например сохранение данных в базу. При этом возможна отправка данных в ответ на запрос. Запрос `POST` идеально подходит для добавления новых записей в блог при помощи веб-формы.



Запрос с объектом XMLHttpRequest

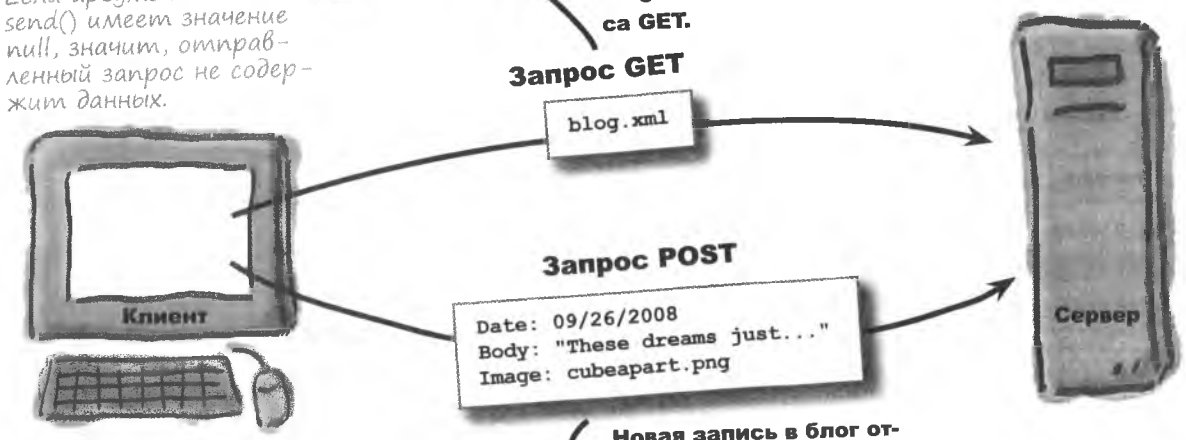
Определившись с типом запроса и указав этот тип в момент открытия, мы подходим к задаче отправки запроса на сервер для последующей обработки. Код отправки запроса зависит от того, какой тип вы выбрали, GET или POST.

Тип GET и URL указываются в момент открытия запроса.

```
request.open("GET", "blog.xml", true); // всегда асинхронный (true)
request.send(null);
```

Если аргумент метода send() имеет значение null, значит, отправленный запрос не содержит данных.

Записи блога в формате XML запрошены из расположенного на сервере файла blog.xml при помощи запроса GET.



Новая запись в блог отправляется на сервер при помощи запроса POST.

Запрос включает отправляемые на сервер данные, поэтому нужно указывать их тип.

При открытии запроса указывается его тип POST и URL сервера.

```
request.open("POST", "addblogentry.php", true); // всегда асинхронный (true)
request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded; charset=UTF-8");
request.send("09/26/2008&These dreams just...&cubeapart.png");
```

Отправляемые вместе с запросом данные передаются методу send() в качестве аргумента.

Не волнуйтесь, если вы пока не поняли разницу между запросами GET и POST.

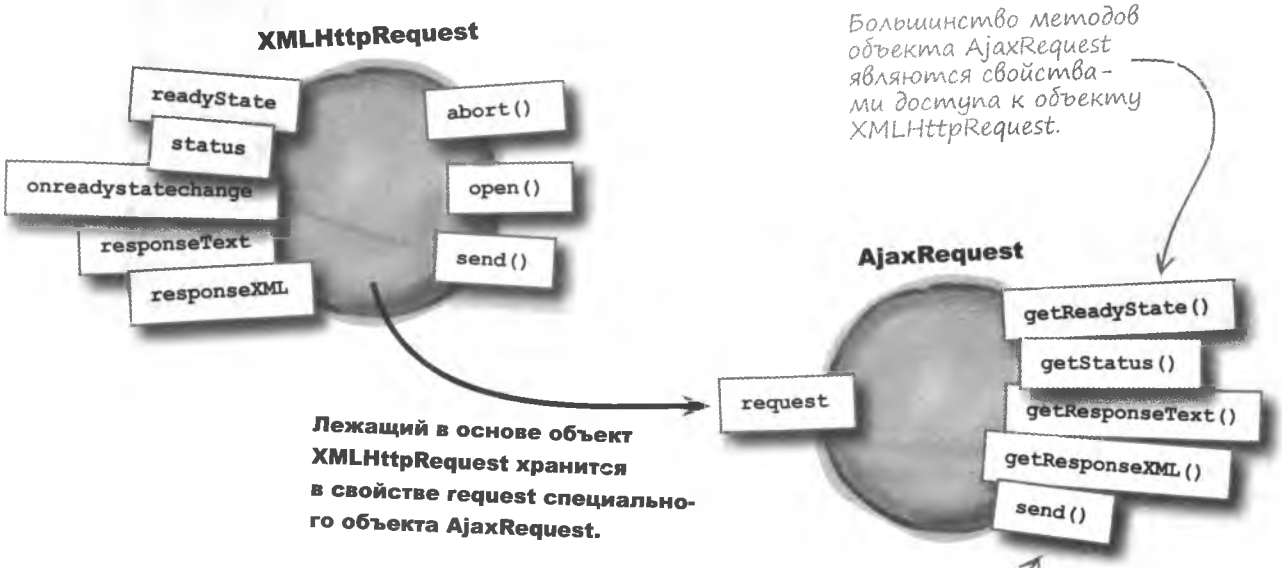
Все станет намного понятнее по мере того, как мы используем полученные знания на практике для обновления нашего блога YouCube.

Упростим задачу

Специальный объект AjaxRequest упростит создание Ajax-запросов.

Объект `XMLHttpRequest` является крайне мощным инструментом, но пользоваться им непросто, в чем вы уже успели убедиться. Впрочем, требуется и изрядное количество «стандартного» кода, который применяется в любом Ajax-приложении. Поэтому для облегчения работы с объектом `XMLHttpRequest` пишутся сторонние библиотеки. Многие из них расширяют функции JavaScript, что требует изучения дополнительного материала.

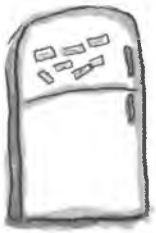
Упростив задачу до предела, создадим для нашего блога YouCube специальный объект, который поможет работать с объектом `XMLHttpRequest`. Это даст нам возможность сфокусировать внимание на технологии Ajax, вместо того чтобы бороться с объектом `XMLHttpRequest` или настраивать стороннюю библиотеку. Наш объект `AjaxRequest` сделает работу с объектом `XMLHttpRequest` намного проще.



Конструктор объекта `AjaxRequest` крайне упрощен, по сравнению с конструктором объекта `XMLHttpRequest`. Вот как просто выглядит код создания объекта `AjaxRequest`, способного инициировать запросы Ajax во всех современных браузерах:

```
var ajaxReq = new AjaxRequest();
```

Конструктор AjaxRequest автоматически берет на себя все сложности создания базового объекта XMLHttpRequest.



Магниты JavaScript

Специальный объект `AjaxRequest` является оболочкой стандартного объекта `XMLHttpRequest`, предоставляя более простой интерфейс для отправки Ajax-запросов и обработки ответов на них. Но в методе `send()` этого объекта не хватает ряда фрагментов. Воспользуйтесь магнитами, чтобы восстановить код метода.

```
AjaxRequest.prototype.send = function(type, url, handler, postDataType, postData) {
  if (this.request != null) {
    // Удаление предыдущего запроса
    this.request.abort();

    // Добавим параметр dummy для переписывания кэша браузера
    url += "?dummy=" + new Date().getTime();

    try {

      this.request.onreadystatechange = ..... ;

      this.request.open(.....,....., true); // всегда асинхронный (true)

      if (type.toLowerCase() == "get") {
        // Отправка запроса GET; без данных

        this.request.send(.....);

      } else {
        // Отправка запроса POST; последний аргумент содержит данные

        this.request.setRequestHeader("Content-Type", ..... );

        this.request.send(..... );

      }
    } catch(e) {
      alert("Ajax error communicating with the server.\n" + "Details: " + e);
    }
  }
}
```

handler

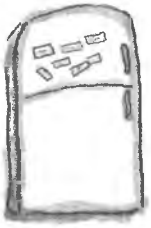
url

null

type

postDataType

postData



Решение задачи с магнитами

Вот как выглядит метод `send()` **объекта** `AjaxRequest`.

Метод send() отправляет запросы с указанным набором аргументов.

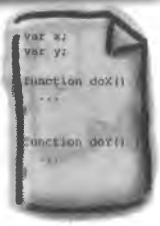
```
AjaxRequest.prototype.send = function(type, url, handler, postDataType, postData) {
  if (this.request != null) {
    // Удаление предыдущего запроса
    this.request.abort();

    // Добавим параметр dummy для переписывания кэша браузера
    url += "?dummy=" + new Date().getTime();

    try {
      this.request.onreadystatechange = handler;

      this.request.open(type, url, true); // всегда асинхронный (true)

      if (type.toLowerCase() == "get") {
        // Отправка запроса GET; нет данных

        this.request.send(null);

      } else {
        // Отправка запроса POST; последний аргумент содержит данные

        this.request.setRequestHeader("Content-Type", postDataType);

        this.request.send(postData);
      }
    } catch(e) {
      alert("Ajax error communicating with the server.\n" + "Details: " + e);
    }
  }
}
```

Функция handler вызывается для обработки ответов сервера.

Аргумент type метода send() определяет, будет ли это запрос GET или POST.

Данные отсылаются на сервер только в случае запроса типа POST.

Этот код хранится во внешнем файле ajax.js вместе с конструктором и другими методами объекта AjaxRequest.



ajax.js

Анализ запросов Ajax

Специальный объект `AjaxRequest` состоит из конструктора и набора методов, один из которых нам особенно полезен. Это метод `send()`, подготавливающий и отправляющий на сервер запросы Ajax. Все эти запросы принадлежат или к типу `GET` или к типу `POST`, аналогично запросам отправки форм в HTML. Но в случае с Ajax запросы не сопровождаются полной перезагрузкой страницы.


```
send(type, url, handler, postDataType, postData)
```

type

Тип запроса, GET или POST.

url

Адрес URL сервера (в случае с блогом YouCube это blog.xml). При необходимости данные пакуются в этот URL.

postDataType

Тип отправляемых данных (только для запросов типа POST).

handler

Функция обратного вызова, используемая для обработки ответов.

postData

Отправляемые данные (только для запросов типа POST). Эти данные можно отправлять в различных форматах.

Все запросы Ajax имеют одинаковую структуру, хотя у запросов типа `GET` отсутствует два последних аргумента. Соответственно, самыми важными для самых простых запросов Ajax являются первые три аргумента метода `send()`. В качестве примера рассмотрим запрос данных XML из находящегося на сервере файла movies.xml:

Тип запроса.

URL запрашиваемого файла.

```
ajaxReq.send("GET", "movies.xml", handleRequest);
```

Предполагается, что объект AjaxRequest уже создан и хранится в переменной ajaxReq.

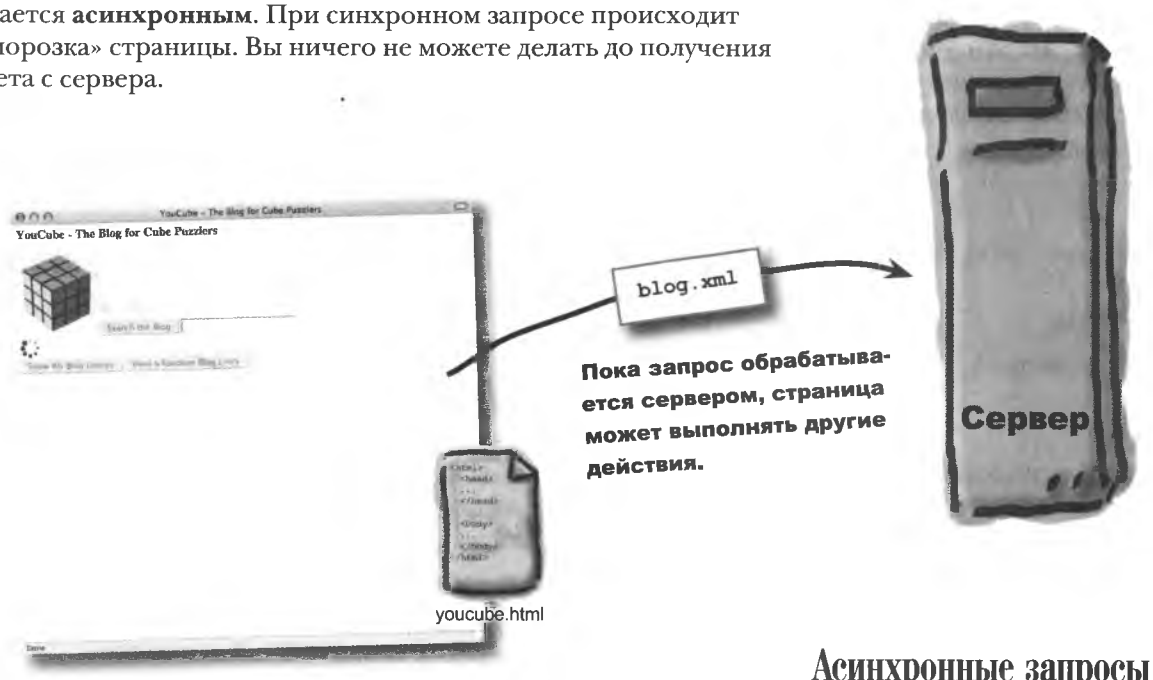
Эта функция будет вызываться для обработки ответа на запрос.

Не волнуйтесь об обработке запросов.

Пока вам достаточно понимать, что существует **специальная функция**, вызываемая после получения ответа на запрос.

Выполнение запросов

Пока метод `send()` отправляет запрос на сервер, веб-страница может выполнять другие операции. Именно поэтому запрос называется **асинхронным**. При синхронном запросе происходит «заморозка» страницы. Вы ничего не можете делать до получения ответа с сервера.



Тот факт, что страница не замораживается в процессе обработки запроса, не означает, что пользователь действительно может сделать что-нибудь продуктивное. Все зависит от особенностей страницы. В случае с блогом YouCube успешный просмотр записей целиком зависит от скорости получения ответа на запрос Ajax.

Асинхронные запросы Ajax не замораживают страницу во время своей обработки.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Объект `XMLHttpRequest` является **стандартным** и предназначен для обработки запросов Ajax.
- **Специальный** объект `AjaxRequest` позволяет работать с Ajax, избегая непосредственных обращений к объекту `XMLHttpRequest`.
- Все запросы Ajax делятся на два типа, `GET` и `POST`, и определяются отправляемыми на сервер данными.
- Метод `send()` объекта `AjaxRequest` открывает запросы Ajax и отправляет их на сервер.

Часто Задаваемые Вопросы

В: Обязателен ли для выполнения запросов объект `AjaxRequest`?

О: Нет. Для отправки запросов и обработки ответов можно воспользоваться непосредственно объектом `XMLHttpRequest`. Но зачем это делать, если использовать объект `AjaxRequest` удобнее и проще? Он создается для удобства и упрощает работу с Ajax, взяв на себя всю сложную работу по формированию запросов.

В: Чем запросы/ответы на них Ajax отличаются от запросов/ответов в HTTP?

О: Запросы HTTP и ответы на них используются браузерами для получения HTML-страниц с веб-серверов. Запросы Ajax во многом с ними сходны. Вот их основные различия: запрос Ajax может быть и не связан с доставкой HTML-данных. Более того, основным достоинством Ajax является именно возможность запросить данные **любого типа**.

Важную роль также играет размер запрашиваемых данных. Ajax вовсе не обязан запрашивать за один раз целую страницу или документ. Вполне можно ограничиться пересылкой фрагмента данных. Именно благодаря этому Ajax позволяет динамически редактировать страницы. При этом вставка новой информации происходит без перезагрузки.

В: То есть Ajax позволяет динамически разобрать страницу по кусочкам?

О: Да! При этом важна не только сама способность собирать страницу из фрагментов. Важно **распределение во времени** этой сборки. Запросы Ajax и обработка ответов на них происходит в реальном времени, не прерывая использования страницы. Другими словами, пользователю не приходится перезагружать страницу для обновления ее небольшого фрагмента. Подгрузка этого фрагмента происходит в фоновом режиме.

В: Каким образом со всем этим связаны запросы `GET` и `POST`?

О: Тип `GET` или `POST` определяет особенность обработки запроса на сервере. Однако способность время от времени динамически запрашивать данные не зависит от типа запроса. Основное различие между запросами `GET` и `POST` в том, меняется или нет состояние сервера в ответ на получение новых данных. Скажем, если данные сохраняются в базе, их отправка осуществляется при помощи запроса типа `POST`. В противном случае можно ограничиться запросом типа `GET`.

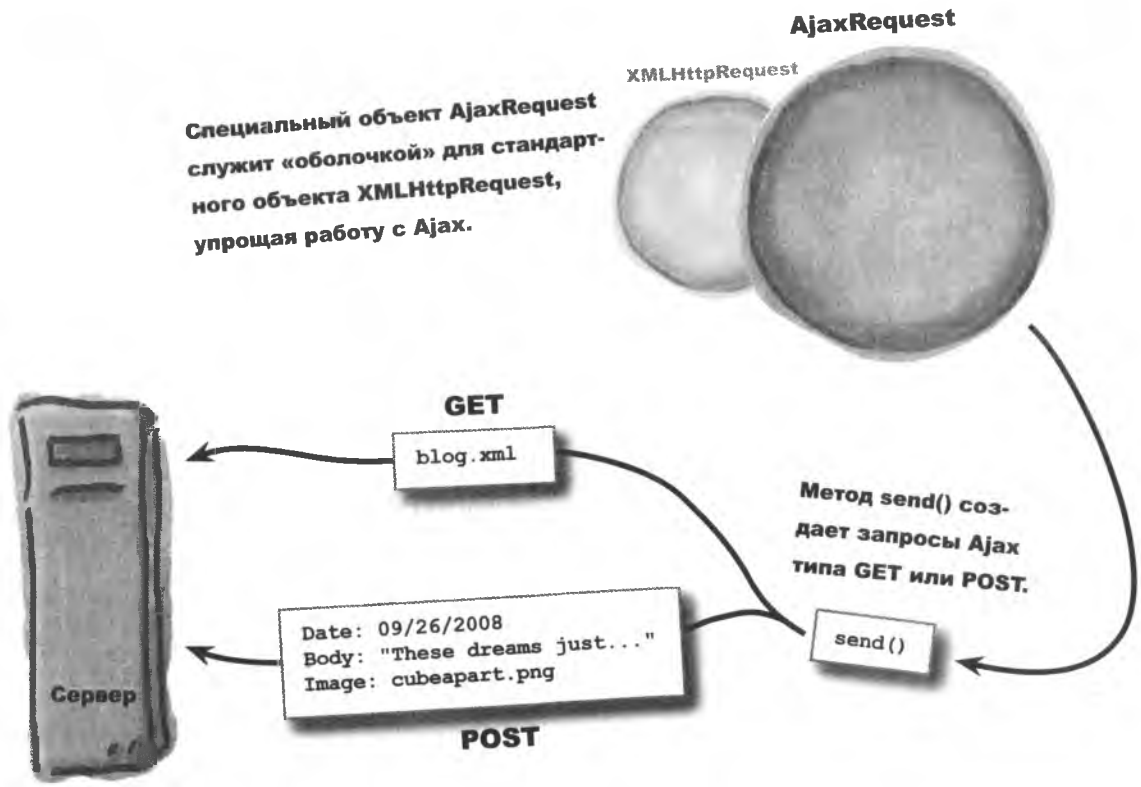
Кто и что делает?

Совместите фрагменты кода с описанием.

`XMLHttpRequest`	Запрашивает данные, ничего не меняя на сервере.
`GET`	Отправляет запрос на сервер и получает ответ.
`send()`	Отправляет данные на сервер, меняя его состояние.
`AjaxRequest`	Стандартный объект JavaScript для работы с Ajax.
`POST`	Специальный объект для упрощения работы с Ajax.

КТО И ЧТО ДЕЛАЕТ?

Вот какое описание соответствует каждому из фрагментов кода.



Создание запросов

Вне зависимости от того, каким образом используется Ajax и к каким данным он пытается получить доступ, обмен данными начинается с запроса. Поэтому первая задача, которую нужно решить Руби для превращения блога YouCube в приложение, управляемое данными, — это запросить файл XML с записями блога.

1. Создание объекта. `AjaxRequest`.
2. Запрос `GET` для получения с сервера файла blog.xml.
3. Обработка запроса?

По поводу шага 3 Руби пока не уверена, поэтому она решает сфокусироваться на первых двух этапах.

Возьми в руку карандаш

Напишите код создания объекта `AjaxRequest` и воспользуйтесь этим объектом для отправки запроса записей блога.

.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит код создания объекта `AjaxRequest` и соответствующий запрос записей блога.

```
1 var ajaxReq = new AjaxRequest();
2 ajaxReq.send("GET", "blog.xml", handleRequest);
```

Нам потребуется запрос GET для получения данных с сервера.

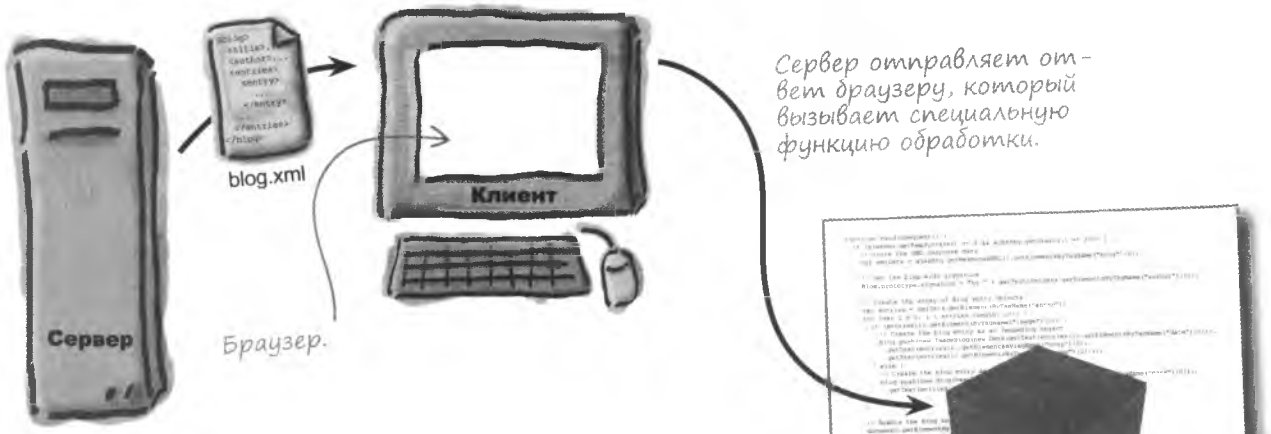
Файл формата XML указан, как URL запроса.

Для обработки запроса нам потребуется специальная функция handleRequest().

Закончишь — вызови меня

После отправки запроса Ajax роль браузера меняется — он не ждет ответа с сервера. Благодаря тому, что запросы Ajax обычно являются **асинхронными**, пользователь может продолжить работу со страницей, а ожидание ответа происходит в фоновом режиме. Как только обработка запроса на сервере закончена, ответ обрабатывается кодом JavaScript при помощи функции **обратного вызова**.

Сценарий на стороне клиента обрабатывает ответ, полученный на запрос Ajax, при помощи специальной функции обратного вызова.

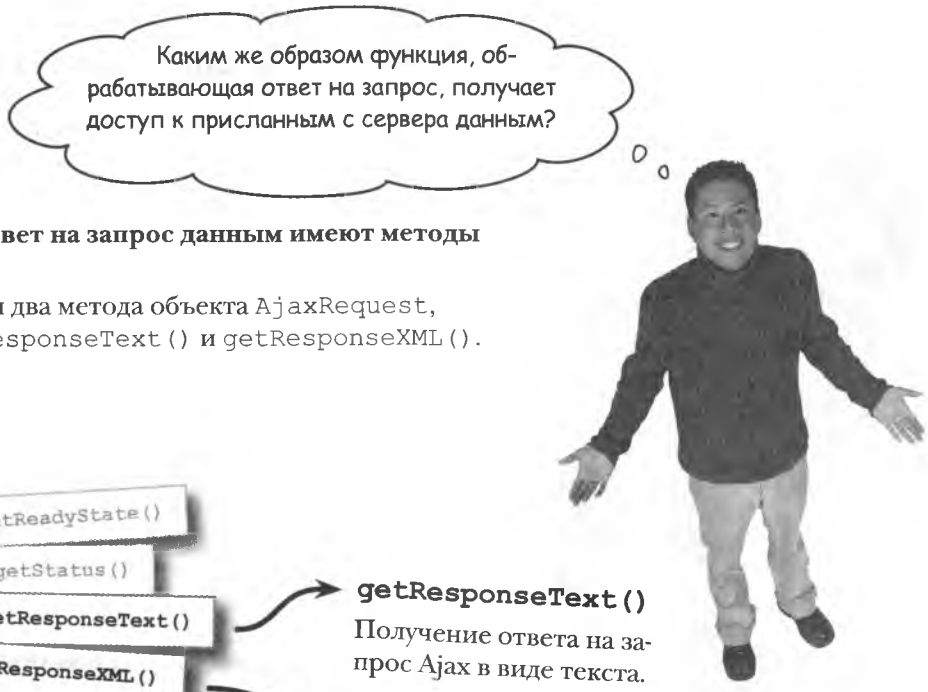


1. Создание объекта `AjaxRequest`.
2. ~~Запрос GET для получения с сервера файла blog.xml.~~
3. Обработка запроса.

В сценарии блога должна присутствовать функция обратного вызова handleRequest().

Обработка ответа

В нашем случае окончание обработки запроса должно приводить к вызову специальной функции `handleRequest()`, которая в зависимости от полученных с сервера данных предпринимает определенные действия.



Доступ к полученным в ответ на запрос данным имеют методы объекта AjaxRequest.

Для этой цели применяется два метода объекта `AjaxRequest`, которые называются `getResponseText()` и `getResponseXML()`.

`getResponseText()`
Получение ответа на запрос Ajax в виде текста.

`getResponseXML()`
Получение ответа на запрос Ajax в виде структурированного XML-кода.

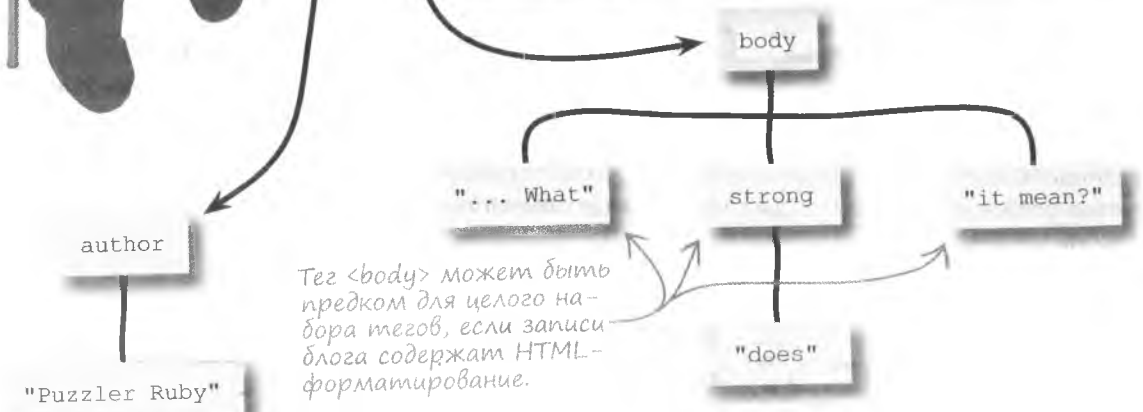
Выбор метода зависит от того, в каком формате вам нужно получить ответ. Для получения структурированного кода используйте метод `getResponseXML()`. Соответственно, метод `getResponseText()` даст ответ в виде обычного текста.

Код XML во многом напоминает код HTML. Как бы вы организовали доступ обработчика запросов к данным XML?

DOM как выход из положения

Любовь Руби к складыванию головоломок явно развивает логическое мышление, потому что она совершенно права, предлагая воспользоваться DOM для обработки данных в формате XML. Как вы помните, DOM управляет данными HTML, представляя их в виде дерева узлов. А значит, аналогично можно поступить с данными XML. Достаточно представить как дерево узлов блог YouCube.

```
<blog>
  <title>YouCube - The Blog for Cube Puzzlers</title>
  <author>Puzzler Ruby</author>
  <entries>
    <entry>
      <date>08/14/2008</date>
      <body>Got the new cube I ordered. It's a real pearl.</body>
    </entry>
    ...
    <entry>
      <date>09/26/2008</date>
      <body>These dreams just keep getting weirder... now I'm seeing
      a cube take itself apart. What <strong>does</strong> it
      mean?</body>
      <image>cubeapart.png</image>
    </entry>
</blog>
```



Руби нужно извлечь записи блога из набора узлов XML, что проще всего сделать при помощи функции. Нет смысла добавлять к блогу YouCube дублирующиеся фрагменты кода, если этого можно избежать.

О функции getText() подробно

Специальная функция `getText()` выполняет монотонную работу по извлечению содержимого узлов DOM.

```
function getText(elem) {
  var text = "";
  if (elem) {
    if (elem.childNodes) {
      for (var i = 0; i < elem.childNodes.length; i++) {
        var child = elem.childNodes[i];
        if (child.nodeValue)
          text += child.nodeValue;
        else {
          if (child.childNodes)
            if (child.childNodes[0].nodeValue)
              text += child.childNodes[0].nodeValue;
        }
      }
    }
  }
  return text;
}
```

Этот аргумент указывает на элемент, содержимое которого требуется извлечь.

Циклический просмотр всех дочерних узлов элемента.

Добавление содержимого дочерних узлов в переменную text.

Если дочерние узлы, в свою очередь, являются предками для других узлов, берем текстовое содержимое первого из них и двигаемся дальше.

Возвращаем переменную text, в которой теперь находится все содержимое дочерних узлов.

Возьми в руку карандаш

Предположим, что полученные в ответ на запрос данные в формате XML уже сохранены в переменную `xmlData`. Напишите код, задающий сигнатуру блога YouCube как содержимое XML-тега `<author>`.

.....

Вот как выглядит код, задающий сигнатуру блога YouCube как содержимое XML-тега <author>.

```
Blog.prototype.signature = "by " + getText(xmlData.getElementsByTagName("author")[0]);
```

Так как сигнатура является свойством уровня класса, ее задание должно осуществляться при помощи прототипа Blog.

Вспомогательная функция getText() извлекает содержимое тега <author>.

В XML-данных может быть только один тег <author>, поэтому просто берем первый тег.

ОБРАБОТКА ОТВЕТОВ НА ЗАПРОСЫ

Интервью недели: Исповедь функции handleRequest()

Head First: Мы слышали, что вы достигли совершенства в ответах на запросы Ajax. Расскажите, что для этого нужно?

handleRequest(): Когда приходит ответ на запрос, я его обрабатываю. Сначала нужно убедиться, что ответ корректен, затем я перехожу к полученным данным и при необходимости интегрирую их в веб-страницу.

Head First: То есть вас вызывают сразу после завершения запроса?

handleRequest(): Да. Хотя на самом деле меня несколько раз вызывают и в процессе обработки запроса, но по большей части в моих услугах люди заинтересованы только в самом конце.

Head First: И как же вы узнаете о том, что наступил тот самый момент?

handleRequest(): Ну, объект AjaxRequest имеет пару методов, при помощи которых я могу проверить состояние запроса и убедиться в том, что его обработка без проблем завершена.

Head First: А откуда вы узнаете, что нужно делать в этот момент?

handleRequest(): Это решаю не я. Я же специальная функция и меняюсь от приложения к приложению.

Head First: Почему?

handleRequest(): Потому что разные приложения используют полученные в ответ на запрос данные по-разному.

Head First: Вы хотите сказать, что для каждого приложения вас нужно писать заново?

handleRequest(): Именно так. И это имеет смысл, ведь, к примеру, приложение для покупок в Интернете обрабатывает Ajax-запросы совсем не так, как блог. Ajax гарантирует мой вызов после завершения обработки запроса, а дальше все зависит от приложения.

Head First: То есть создание управляемой Ajax страницы означает создание специального обработчика запросов?

handleRequest(): Да, вы все правильно поняли.

Head First: Спасибо за поучительную беседу.

handleRequest(): Всегда счастлива поговорить.

ВАША НОВАЯ РАБОТА

Попробуйте себя в роли составителя примечаний и объясните, как работает функция handleRequest() function. 7 – магическое число. Найдете ли вы 7 особенностей, ведущих к успешному выполнению запроса?

```
function handleRequest() {
  if (ajaxReq.getReadyState() == 4 && ajaxReq.getStatus() == 200) {
    // Сохраняем полученные данные в формате XML
    var xmlData = ajaxReq.getResponseXML().getElementsByTagName("blog")[0];

    // Задаем сигнатуру блога
    Blog.prototype.signature = "by " + getText(xmlData.getElementsByTagName("author")[0]);

    // Создаем массив объектов Blog, содержащий отдельные записи
    var entries = xmlData.getElementsByTagName("entry");
    for (var i = 0; i < entries.length; i++) {
      // Создаем запись
      blog.push(new Blog(getText(entries[i].getElementsByTagName("body")[0]),
        new Date(getText(entries[i].getElementsByTagName("date")[0])),
        getText(entries[i].getElementsByTagName("image")[0])));
    }

    // Отображаем блог
    showBlog(5);
  }
}
```

Ответ на ЗАДАЧУ

Вот какими комментариями следовало снабдить специальную функцию handleRequest().

Присваиваем сигнатуре блога содержимое тега <author>.

Убеждаемся в успешном выполнении запроса Ajax, проверяя его состояние.

XML-данные содержат всего один тег <blog>, поэтому берем первый элемент массива, возвращенного функцией getElementsByTagName().

```
function handleRequest() {
  if (ajaxReq.getReadyState() == 4 && ajaxReq.getStatus() == 200) {
    // Сохраняем полученные данные в формате XML
    var xmlData = ajaxReq.getResponseXML().getElementsByTagName("blog")[0];

    // Задаем сигнатуру блога
    Blog.prototype.signature = "by " + getText(xmlData.getElementsByTagName("author")[0]);

    // Создаем массив объектов Blog, содержащий отдельные записи
    var entries = xmlData.getElementsByTagName("entry");
    for (var i = 0; i < entries.length; i++) {
      // Создаем запись
      blog.push(new Blog(getText(entries[i].getElementsByTagName("body")[0]),
        new Date(getText(entries[i].getElementsByTagName("date")[0])),
        getText(entries[i].getElementsByTagName("image")[0])));
    }

    // Отображаем блог
    showBlog(5);
  }
}
```

Получаем все элементы, содержащие записи блога.

Создаем новый объект blog для очередной записи и добавляем туда последний элемент массива при помощи метода push() объекта Array.

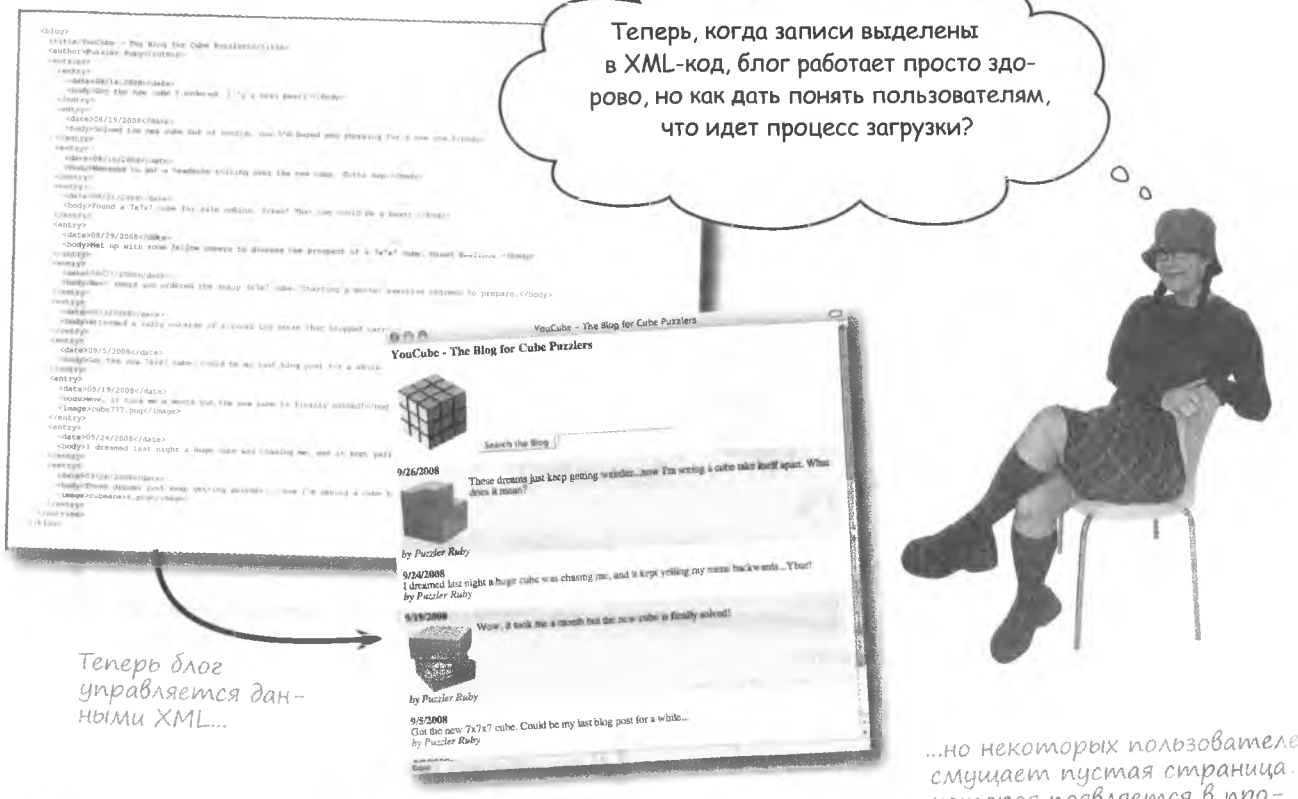
Вызываем функцию showBlog() для отображения на странице пяти последних записей.

1. ~~Создание объекта AjaxRequest.~~
2. ~~Запрос GET для получения с сервера файла blog.xml.~~
3. ~~Обработка запроса.~~ Сделано!

YouCube, управляемый данными

Руби восхищена новым обликом, который ее блог приобрел благодаря Ajax (это сэкономило ей много времени), но она обеспокоена тем, что происходит на странице в процессе загрузки данных.

Последняя версия файлов YouCube доступна для скачивания по адресу http://www.headfirstlabs.com/books/hfjs/.



Возьми в руку карандаш

Впишите в функцию `loadBlog()` отсутствующую строчку кода, которая отвечает за отображение картинки wait.gif, демонстрируемой в процессе загрузки записей.

Подсказка: используйте основной тег `div` блога с ID `"blog"`.

```
function loadBlog() {

  .....

  ajaxReq.send("GET", "blog.xml", handleRequest);

}
```

Возьми в руку карандаш

Решение

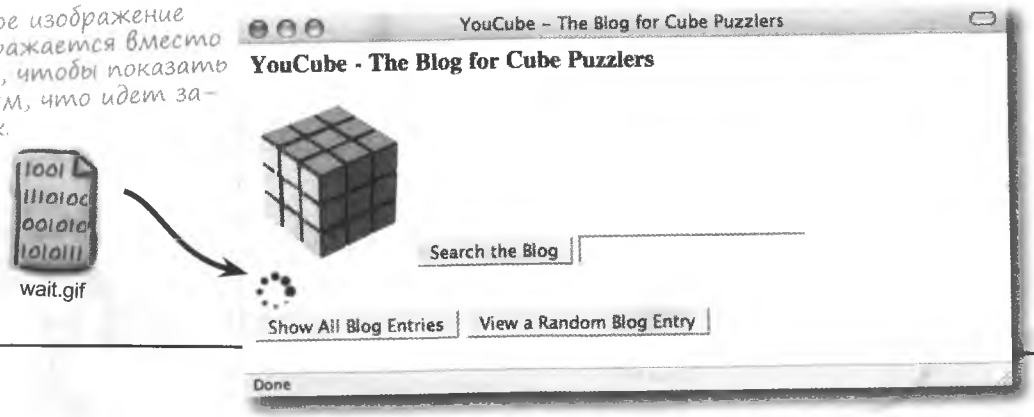
Вот как выглядит код функции `loadBlog()`.

```
function loadBlog() {
  document.getElementBuld("blog").innerHTML = "<img src='wait.gif' alt='Loading... ' />";
  ajaxReq.send("GET", "blog.xml", handleRequest);
}
```

Изображение демонстрируется в процессе загрузки записей блога.

В данном случае проще воспользоваться свойством innerHTML, чем DOM, потому что тег image добавляется к паре атрибутов.

Анимированное изображение wait.gif отображается вместо записей блога, чтобы показать пользователям, что идет загрузка данных.



Часто Задаваемые Вопросы

В: Последняя запись блога YouCube содержала тег HTML `<strong>`. Допустимо ли это в коде XML?

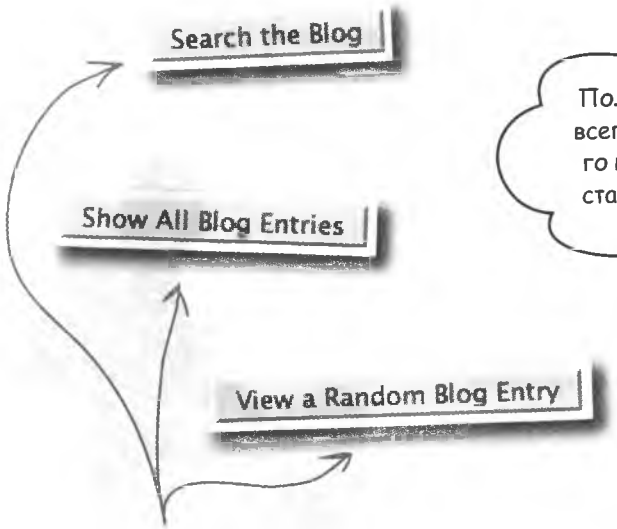
О: Если помните, XML-код применяется для представления данных любого типа. В данном случае, учитывая, что тело записи будет вставлено в веб-страницу, технически возможно включить HTML-теги, влияющие на вид записи после вставки. Другими словами, содержимое каждой записи блога может включать в себя HTML-теги, передаваемые вместе с XML-кодом в виде специальных узлов, ответственных за форматирование. Впрочем, это достаточно сложная задача, ведь нам нужно реконструировать узлы, которые использовались для форматирования HTML-кода, при вставке в страницу XML-данных. В случае блога YouCube мы решили отделить текст от HTML-тегов, оставив форматирование за кадром. Но вполне возможно, что в следующей версии YouCube этот аспект будет учтен.

В: Как работают свойства `readyState` и `status`?

О: Оба этих свойства принадлежат объекту `XMLHttpRequest` и предназначены для отслеживания **состояния** запроса, например (0) означает, что запрос не инициализирован, а (4) — что данные загружены, а также его **статуса**, например 404 означает (not found), а 200 означает (OK). Подробное рассмотрение этих свойств в данном случае не требуется. Вам достаточно знать, что запрос выполнен, если состояние равно 4 (loaded), а статус имеет значение 200 (OK). Только при этих условиях вызывается функция `handleRequest()`.

Неработающие кнопки

Хотя внесенные при помощи Ajax изменения не оказали особого влияния на внешний вид блога YouCube, кое в чем интерфейс претерпел изменения. Кажется, кнопки на нашей странице перестали работать нужным нам образом.



По непонятным причинам кнопки иногда перестают работать, а блог при этом становится невидимым.

Пользователи сообщают, что кнопки не всегда срабатывают. После щелчка ничего не происходит. Более того, сам блог становится невидимым. В чем же дело?

Неработающие кнопки = Разочарованные пользователи

Руби не волнуется, но очень хотела бы устранить эти неполадки.

МОЗГОВОЙ ШТУРМ

По каким причинам могут не работать кнопки? На какой стадии загрузки страницы может возникать эта проблема?

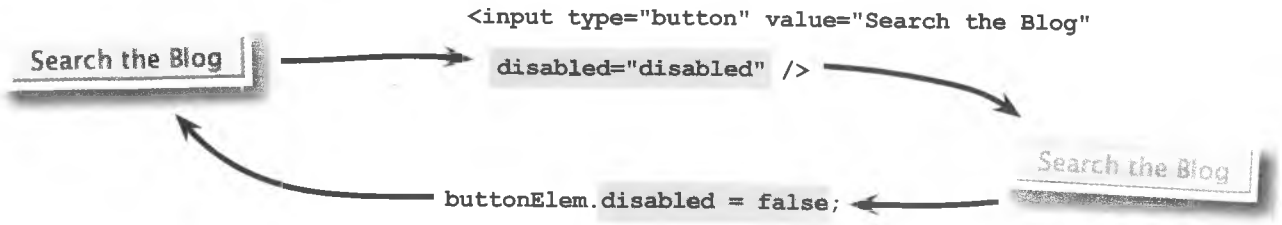
Кнопкам нужны данные

Дело в том, что кнопки в нашем блоге работают только при наличии доступа к данным. А так как данные теперь загружаются из внешнего XML-файла, некоторое время страница существует без них. В этот период существование кнопок лишено смысла и только запутывает пользователя.

Отключение кнопок — прекрасное решение.

Отключив кнопки на время загрузки данных, мы простым и элегантным способом решим проблему. Ajax отправляет запрос данных при первой загрузке страницы, а значит, кнопки можно изначально отключить, предоставив затем к ним доступ при помощи функции `handleRequest()`, которая запускается после завершения обработки запроса.

Отключение реализуется при помощи атрибута `disabled` тега `<input>`. Этому тегу присваивается значение `"disabled"` в HTML-коде. И наоборот, ему можно присвоить значение `false` в коде JavaScript, включив тем самым кнопку.



Магниты JavaScript

Воспользуйтесь магнитами для заполнения пробелов в коде страницы YouCube. Сделайте так, чтобы кнопки были недоступны до завершения загрузки записей. Некоторые магниты можно использовать несколько раз.

```
<html>
  <head>
    <title>YouCube - The Blog for Cube Puzzlers</title>

    <script type="text/javascript" src="ajax.js"> </script>
    <script type="text/javascript" src="date.js"> </script>

    <script type="text/javascript">
      ...
      function handleRequest() {
        if (ajaxReq.getReadyState() == 4 && ajaxReq.getStatus() == 200) {
          ...
          // Включение кнопок

          document.getElementById(.....).....= .....;

          document.getElementById(.....).....= .....;

          document.getElementById(.....).....= .....;

          ...
        }
      }
      ...
    </script>
  </head>

  <body onload="loadBlog();">
    <h3>YouCube - The Blog for Cube Puzzlers</h3>
    <img src="cube.png" alt="YouCube" />
    <input type="button" id="search" value="Search the Blog"

      .....= .....  onclick="searchBlog();" />

    <input type="text" id="searchtext" name="searchtext" value="" />
    <div id="blog"></div>
    <input type="button" id="showall" value="Show All Blog Entries"

      .....= .....  onclick="showBlog();" />

    <input type="button" id="viewrandom" value="View a Random Blog Entry"

      .....= .....  onclick="randomBlog();" />

  </body>
</html>
```

true

"search"

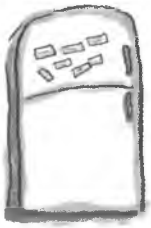
"viewrandom"

disabled

false

"showall"

"disabled"



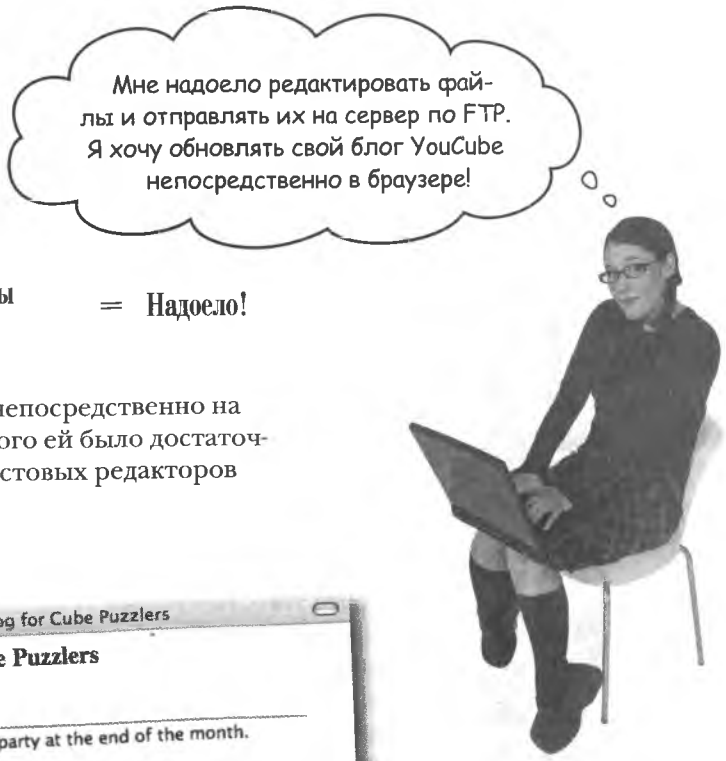
Решение задачи с магнитами

Вот как должен выглядеть код блога YouCube, в котором доступ к кнопкам предоставляется только после окончания загрузки записей.

```
<html>
  <head>
    <title>YouCube - The Blog for Cube Puzzlers</title>

    <script type="text/javascript" src="ajax.js"> </script>
    <script type="text/javascript" src="date.js"> </script>

    <script type="text/javascript">
      ...
      function handleRequest() {
        if (ajaxReq.getReadyState() == 4 && ajaxReq.getStatus() == 200) {
          ...
          // Включение кнопок
          document.getElementById( "search" ).disabled = false ;
          document.getElementById( "showall" ).disabled = false ;
          document.getElementById( "viewrandom" ).disabled = false ;
          ...
        }
      }
      ...
    </script>
  </head>

  <body onload="loadBlog();">
    <h3>YouCube - The Blog for Cube Puzzlers</h3>
    <img src="cube.png" alt="YouCube" />
    <input type="button" id="search" value="Search the Blog"
        disabled = "disabled"  onclick="searchBlog();" />
    <input type="text" id="searchtext" name="searchtext" value="" />
    <div id="blog"></div>
    <input type="button" id="showall" value="Show All Blog Entries"
        disabled = "disabled"  onclick="showBlog();" />
    <input type="button" id="viewrandom" value="View a Random Blog Entry"
        disabled = "disabled"  onclick="randomBlog();" />
  </body>
</html>
```

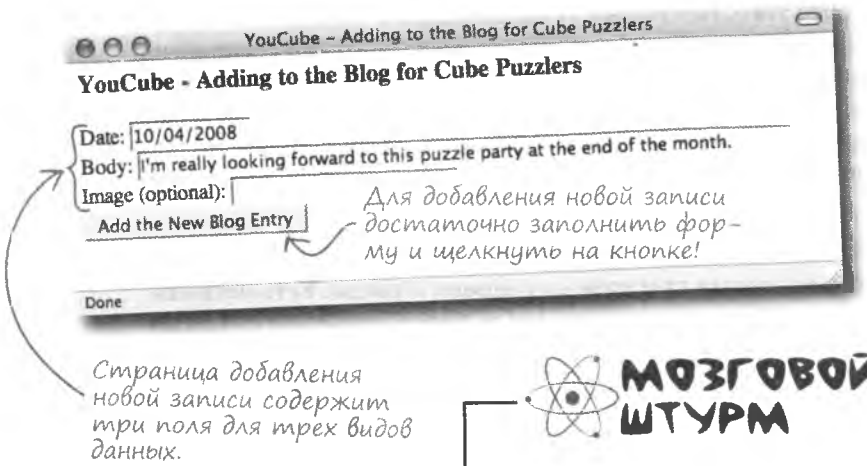
Функция, экономящая время

На данный момент блог YouCube уже управляется динамическими данными, но Руби пока еще рано почивать на лаврах. На странице блога отсутствует интерфейс для добавления записей. А Руби хотела бы не редактировать каждый раз XML-файл, а писать непосредственно в блог и сохранять результаты на сервер.



Редактировать код + **Отправлять файлы на сервер** = **Надоело!**

Руби хотела бы вводить записи в форму непосредственно на странице блога. Хотела бы, чтобы для этого ей было достаточно браузера и чтобы больше никаких текстовых редакторов и FTP-клиентов.

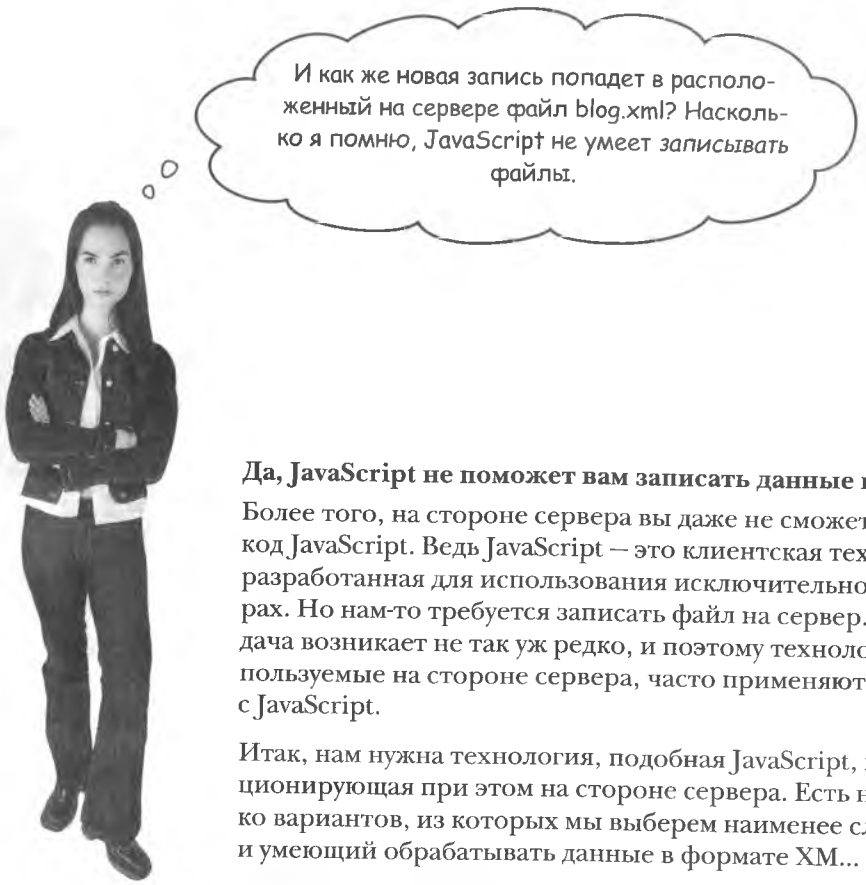


МОЗГОВОЙ ШТУРМ

Каким образом Ajax позволяет добавлять записи в формате XML посредством пользовательского интерфейса на веб-странице?

Запись данных в блог

Размышляя о записи данных в блог в терминах Ajax, можно представить запрос типа `POST`, отправляющий их на сервер, где они записываются в файл `blog.xml`. Ответ сервера в данном случае не требуется.



Да, JavaScript не поможет вам записать данные на сервер.

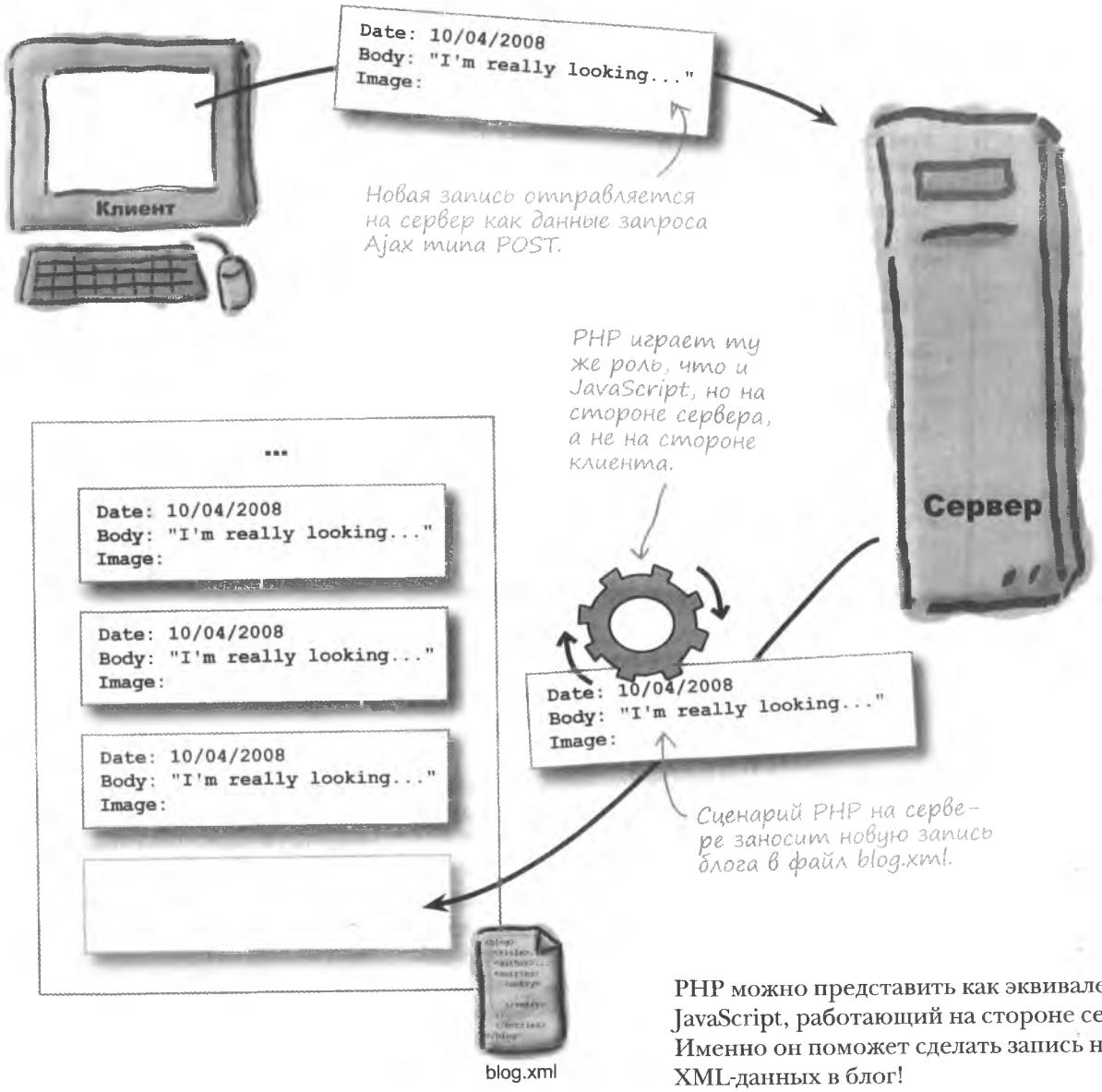
Более того, на стороне сервера вы даже не сможете запустить код JavaScript. Ведь JavaScript — это клиентская технология, разработанная для использования исключительно в браузерах. Но нам-то требуется записать файл на сервер. Эта задача возникает не так уж редко, и поэтому технологии, используемые на стороне сервера, часто применяются в паре с JavaScript.

Итак, нам нужна технология, подобная JavaScript, но функционирующая при этом на стороне сервера. Есть несколько вариантов, из которых мы выберем наименее сложный и умеющий обрабатывать данные в формате XM...

На помощь приходит PHP

Язык написания сценариев PHP — это то, что нам нужно для записи данных в XML-файл на сервере. Задача включает в себя чтение XML-файла, добавление новой записи к уже существующим и сохранение новой информации. Впрочем, в результате мы все равно возвращаемся к получению записей с сервера при помощи Ajax-запроса со стороны браузера.

PHP — это язык написания сценариев, работающий на стороне сервера.



PHP можно представить как эквивалент JavaScript, работающий на стороне сервера... Именно он поможет сделать запись новых XML-данных в блог!

Готовый код PHP

Сценарий PHP на стороне сервера добавляет новую запись в файл blog.xml.

Загрузка строчных данных XML в переменную $rawBlog.

Проверяем, существует ли нужный файл.

Если файл не существует, создаем пустой документ XML.

Преобразуем строчные данные записи в формат XML, который во многом напоминает дерево DOM в JavaScript.

Новая запись становится дочерним узлом в структуре данных XML.

Сохраняем файл блога после добавления новой записи.

```
<?php
$filename = "blog.xml";

if (file_exists($filename)) {
  // Загрузка записей блога из файла XML
  $rawBlog = file_get_contents($filename);
}
else {
  // Создание пустого XML-документа
  $rawBlog = "<?xml version=\"1.0\" encoding=\"utf-8\" ?>";
  $rawBlog .= "<blog><title>YouCube - The Blog for Cube Puzzlers</
title>";
  $rawBlog .= "<author>Puzzler Ruby</author><entries></entries></blog>";
}
$xml = new SimpleXmlElement($rawBlog);

// Добавляем новую запись блога как дочерний узел
$entry = $xml->entries->addChild("entry");
$entry->addChild("date", $_REQUEST["date"]);
$entry->addChild("body", stripslashes($_REQUEST["body"]));
if ($_REQUEST["image"] != "")
  $entry->addChild("image", $_REQUEST["image"]);

// Записываем блог в файл
$file = fopen($filename, 'w');
fwrite($file, $xml->asXML());
fclose($file);
?>
```

addblogentry.php

Сценарий PHP хранится в файле addblogentry.php.

Часто Задаваемые Вопросы

В: Обязательно ли использовать PHP для записи файлов на сервер?

О: Не обязательно. Существуют и другие технологии написания сценариев, работающих на стороне сервера. Например, Perl (CGI), который умеет делать ровно то же самое, что и PHP. Вы можете по своему желанию выбрать технологию для создания работающего на стороне сервера компонента Ajax-приложения.

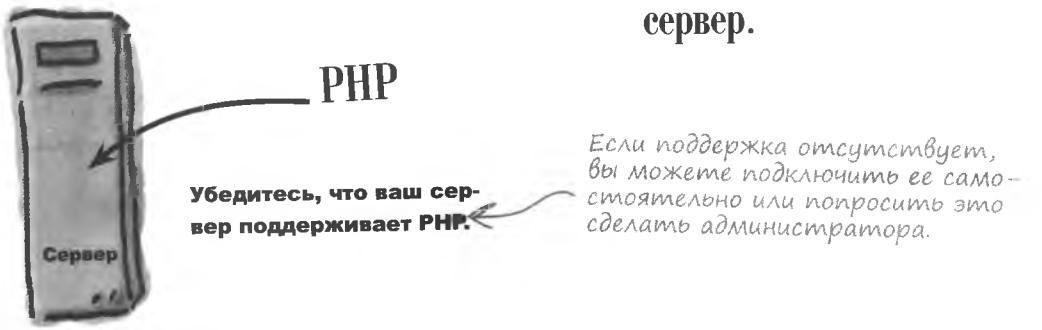
В: Можно ли применять Ajax без необходимости использовать программы, работающие на стороне сервера?

О: В некоторых случаях да. Помните, что все запросы Ajax, кроме самых простых, сводятся к получению сервером данных от клиента и последующей обработке этих данных, например поиску информации в базе или записи в файл. Хорошим примером запроса Ajax, не требующим выполнения сценариев на стороне сервера, является главная страница блога YouCube. Но большинство Ajax-приложений далеко не так просты, и без сценариев, работающих на стороне сервера, уже не обойтись. Вопрос в том, отправляется ли вам в качестве ответа сервера целый файл, как в случае с файлом blog.xml, или же данные требуют более сложной обработки. Впрочем, большинство предназначенных для этого сценариев крайне просты и не требуют особых знаний по программированию на стороне сервера.

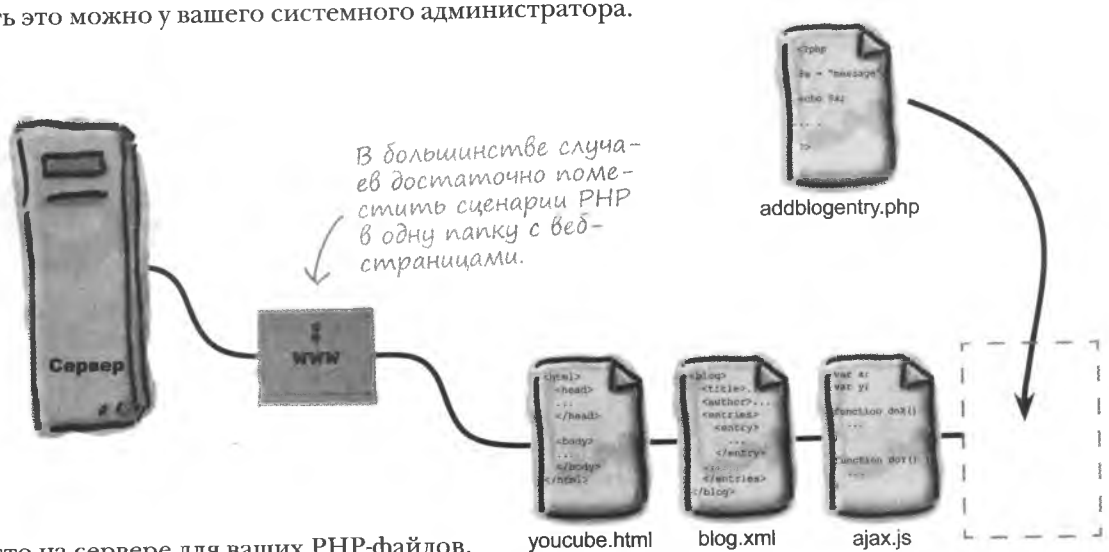
Требования PHP

В отличие от языка JavaScript, по умолчанию поддерживаемого современными браузерами, далеко не на всех серверах поддерживается PHP. Поэтому перед отправкой на сервер PHP-файлов нужно узнать у системного администратора, насколько это допустимо. Возможно вам потребуется провести некоторые операции по настройке или поискать другой сервер. Ведь без поддержки PHP сценарий блога YouCube просто не будет работать.

Для запуска PHP-сценариев может потребоваться предварительно настроить сервер.



Затем нужно выбрать место на сервере, где будут храниться ваши PHP-файлы. В большинстве случаев их можно поместить в ту же папку, в которой хранятся HTML-страницы и внешние файлы JavaScript. Но иногда установки PHP требуют сохранения сценариев в отдельную папку. Узнать это можно у вашего системного администратора.



Выбрав место на сервере для ваших PHP-файлов, продолжим работу над усовершенствованием нашего блога YouCube.

Данные для PHP-сценария

Давайте посмотрим, каким образом PHP-сценарий записывает данные в расположенный на сервере XML-файл. Именно это позволит нам составить запрос Ajax таким образом, чтобы выполнить поставленную задачу.

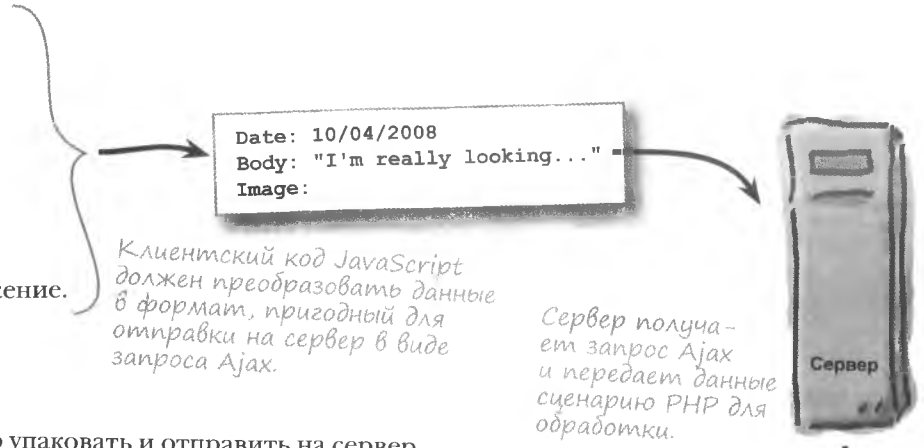
Сценарию PHP требуется информация о новой записи в блог, состоящая из двух, а может быть, даже трех фрагментов.

Данные передаются PHP-сценарию через запрос Ajax.

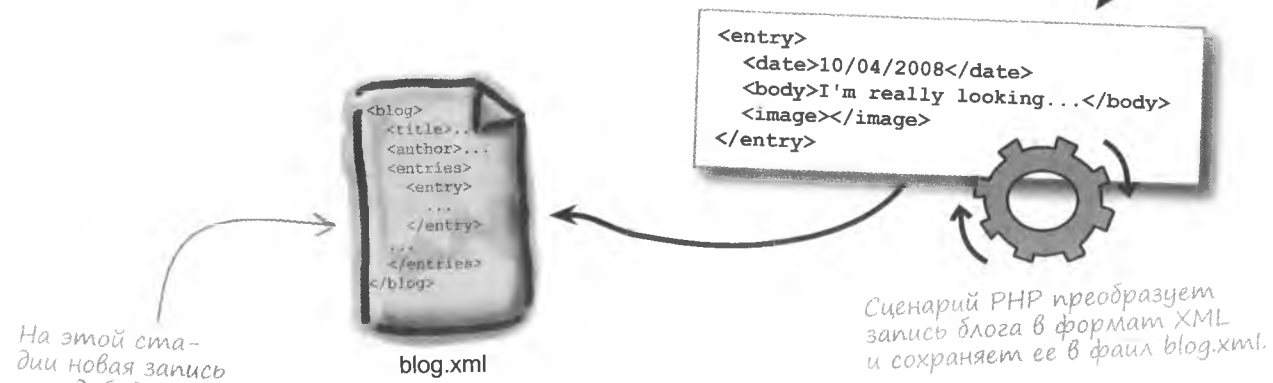
Date
Дата записи.

Body
Тело записи.

Image
Присоединенное изображение.



Всю эту информацию нужно упаковать и отправить на сервер в виде запроса Ajax. Там он будет обработан и сохранен в файл blog.xml.



Теперь нужно понять, как должна выглядеть веб-страница, интерфейс которой позволяет выполнять ввод новых записей, а потом собирает информацию и передает ее на сервер при помощи запроса Ajax. К счастью, нам практически ничего не потребуется делать в ответ на запрос, кроме разве что подтверждения об успешном сохранении новой записи.

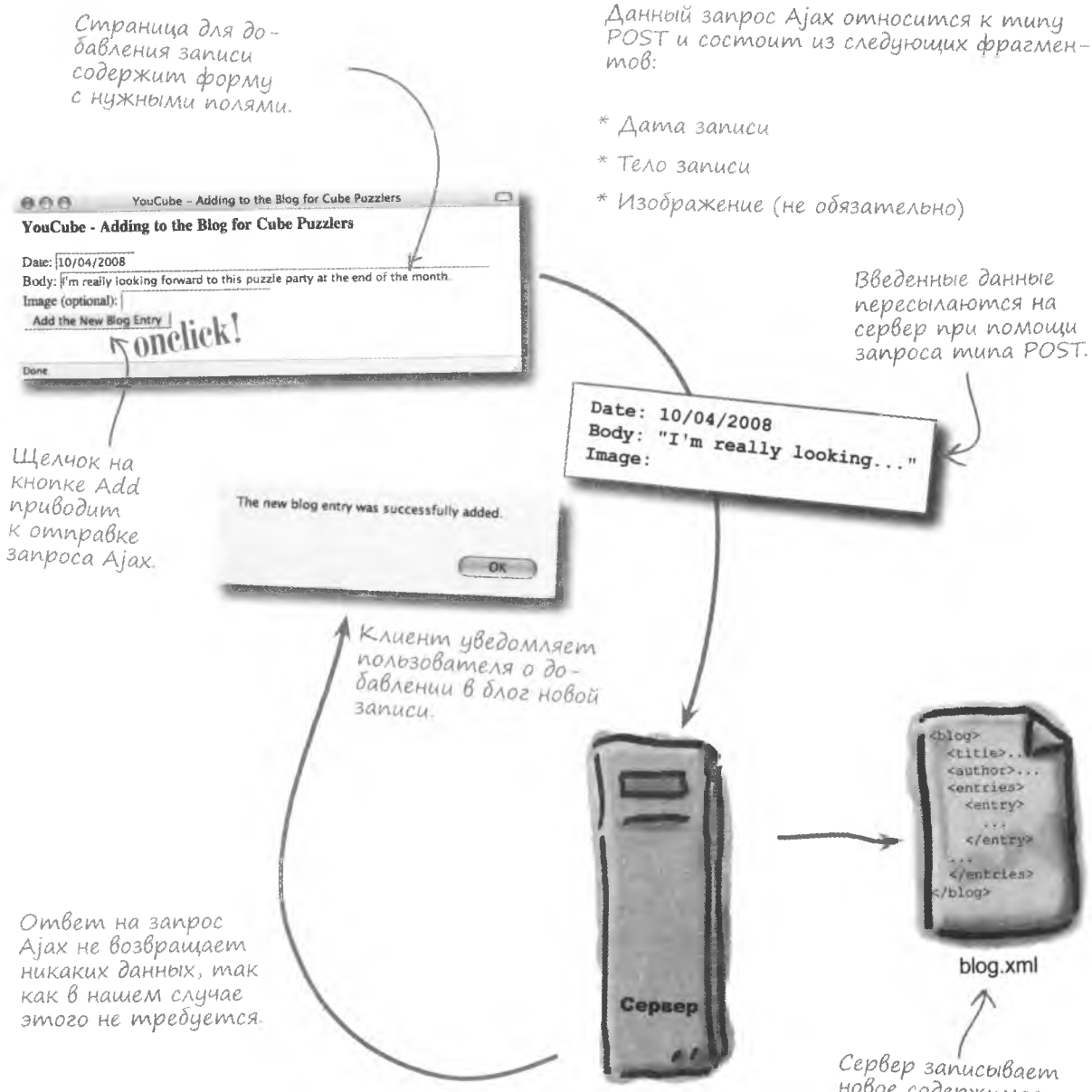
Возьми в руку карандаш

Набросайте конструкцию веб-страницы блога YouCube, предназначенную для добавления новых записей. Не забудьте показать, каким именно образом запрос Ajax и ответ на него влияют на передачу данных.

Возьми в руку карандаш

Решение

Вот как выглядит веб-страница блога YouCube, предназначенная для добавления новых записей.



Отправка данных на сервер

Запрос POST сложнее запроса GET, так как связан с отправкой данных на сервер. Он поддерживает различные способы упаковки информации, но для нас вполне подойдет стандартное **кодирование URL**. Именно при помощи этой техники браузеры передают на сервер поля данных в адресе URL веб-страницы. Ее можно отличить по символу (&), используемому для разделения фрагментов данных.

```
Date: 10/04/2008
Body: "I'm really looking... "
Image:
```

```
"date=10/04/2008&body=I'm really looking forward... &image="
```

Отдельный фрагмент данных состоит из имени и значения.

Фрагменты данных отделяются друг от друга символом &.

В этом формате данных все фрагменты состоят из имени и значения, разделенных знаком (=), а каждая пара имя/значение отделена знаком (&). Формат называется закодированным URL и имеет свой собственный тип данных, который указывается в запросе Ajax POST.

Это официальный тип данных закодированного URL, который указывается при формировании запроса POST.

```
"application/x-www-form-urlencoded; charset=UTF-8"
```

Итак, все готово для написания кода запроса и его отправки на сервер, где данные будут сохранены в файл blog.xml.

Упражнение

Приведите показанные ниже фрагменты данных в формат закодированного URL, подходящий для создания запроса POST.

```
releaseDate: 01/13/1989

title: Gleaming the Cube

director: Graeme Clifford
```

.....

Упражнение
Решение

Вот как будут выглядеть фрагменты данных в формате закодированного URL, подходящего для создания запроса POST.

```
releaseDate: 01/13/1989
title: Gleaming the Cube
director: Graeme Clifford
```

```
"title=Gleaming the Cube&releaseDate=01/13/1989&director=Graeme Clifford"
```

Часто Задаваемые Вопросы

В: Если сценарий добавления записи в блог не требует данных с сервера в ответ на запос Ajax, зачем нам обрабатывать этот запрос?

О: Нам важно узнать, что запрос был успешно завершен. Ведь именно эта информация сигнализирует сценарию, что можно отобразить всплывающее окно с подтверждением добавления новой записи в блог.

В: Можно ли в сценарии добавления записи использовать еще и запрос GET?

О: Технически это возможно. Допустимо отправлять данные на сервер вместе с запросом GET, но нужно точно указывать URL этого запроса. Впрочем, это не проблема — проблема в том, что запрос GET не предназначен для ситуаций, когда меняется состояние сервера. А в случае добавления записи в файл blog.xml определенно можно говорить об изменении состояния. Именно поэтому нужно использовать запрос POST, недвусмысленно указывающий на намерение взаимодействовать с сервером.

В: Так как обработка запроса Ajax и сохранение записи занимают некоторое время, что будет при щелчке на кнопке Add до завершения запроса?

О: Каждый щелчок на кнопке Add отменяет текущий запрос и отправляет новый. Хотя можно представить и преднамеренный двойной щелчок, в интерфейсе имело бы смысл предусмотреть, чтобы эта кнопка становилось недоступной до завершения запроса. То есть код добавления новой записи должен отключать кнопку Add на время обработки запроса, а затем активировать ее снова. Подобные усовершенствования интерфейса приложений JavaScript сделают работу более интуитивной и простой и, как результат, осчастливят пользователей.

В: Что происходит с пробелами в данных, форматируемых в закодированный URL?

О: Пробелы в данном случае не являются проблемой, так как Ajax обрабатывает данные автоматически и гарантирует корректность формата с точки зрения сервера.

В: Всегда ли при передаче данных на сервер к ним нужно добавлять изображение?

О: Нет, этого можно не делать. Вполне допустимо отправлять пустые фрагменты данных, у которых после знака равенства в закодированном URL ничего не стоит:

```
"date=...&body=...&image="
```

В данном примере содержимое поля с изображением отправляется на сервер, хотя и не содержит данных. Далее начинается работа PHP-сценария на стороне сервера, который достаточно интеллектуален, чтобы увидеть, что в поле ничего не было введено, соответственно, новая запись в блог не сопровождается картинкой.

Возьми в руку карандаш

Закончите написание кода функций `addBlogEntry()` и `handleRequest()` в сценарии добавления записей в блог YouCube.

```
function addBlogEntry() {
  // Отключение кнопки Add и присвоение статусу значения busy
  ..........
  ..........

  // Пересылка новой записи блога в виде запроса Ajax
  ajaxReq.send("POST", "addblogentry.php", handleRequest,
    "application/x-www-form-urlencoded; charset=UTF-8",
    ..........
    ..........
    ..........);
}

function handleRequest() {
  if (ajaxReq.getReadyState() == 4 && ajaxReq.getStatus() == 200) {
    // Включение кнопки Add и очистка статуса
    ..........
    ..........

    // Подтверждение добавления записи в блог
    alert("The new blog entry was successfully added.");
  }
}
```

Возьми в руку карандаш

Решение

Вот как выглядят функции `addBlogEntry()` и `handleRequest()` из сценария добавления в блог YouCube записей.

В процессе сохранения новой записи кнопка Add отключается.

В строке состояния отображается сообщение «busy», чтобы пользователь знал, что идет загрузка.

```
function addBlogEntry() {

  // Отключение кнопки Add и присвоение статусу значения busy
  document.getElementById("add").disabled = true;
  document.getElementById("status").innerHTML = "Adding... ";

  // Пересылка новой записи блога в виде запроса Ajax
  ajaxReq.send("POST", "addblogentry.php", handleRequest,
    "application/x-www-form-urlencoded; charset=UTF-8",
    "date=" + document.getElementById("date").value +
    "&body=" + document.getElementById("body").value +
    "&image=" + document.getElementById("image").value );
}

function handleRequest() {
  if (ajaxReq.getReadyState() == 4 && ajaxReq.getStatus() == 200) {
    // Включение кнопки Add и очистка статуса
    document.getElementById("add").disabled = false;
    document.getElementById("status").innerHTML = "";

    // Подтверждение добавления записи в блог
    alert("The new blog entry was successfully added.");
  }
}
```

Это запрос POST.

Сценарий PHP используется для сохранения записи блога в файле на сервере.

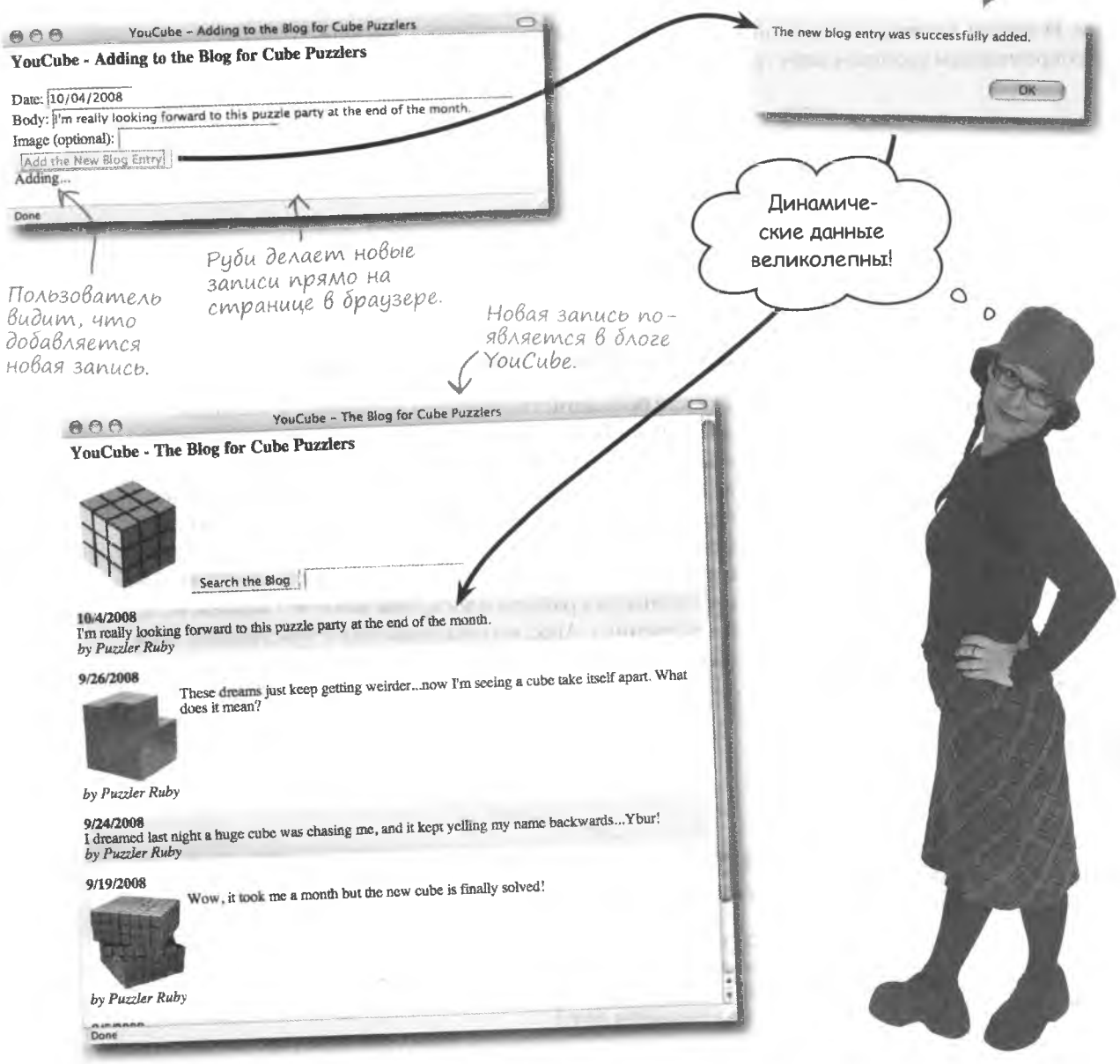
Сборка данных запроса POST из полей формы date, body и image.

Проверка результатов выполнения запроса на сохранение записи.

Включение кнопки Add и очистка строки состояния указывает на завершение процедуры сохранения.

Вести блог легко

Руби не может поверить, насколько проще стало обновлять блог. Ведь ей больше не нужно открывать файл, редактировать код и загружать новую версию файла на сервер. Блог теперь не только управляется данными, но дарит вдохновение для новых записей!



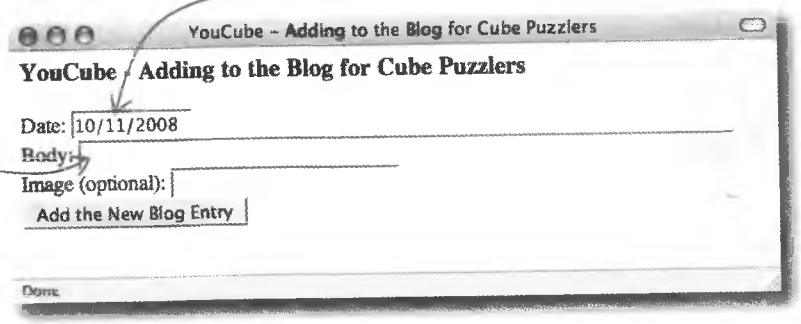
Делаем работу с блогом еще удобнее

Невозможно достичь подлинного мастерства в сборке головоломок без скрупулезного внимания к деталям. Не удивительно, что Руби хочет довести страницу добавления записей до совершенства. Она узнала, что приложения Ajax известны внимательным отношением к мелочам, влияющим на удобство работы. И теперь хочет сделать свой блог конкурентоспособным по отношению к современным удобным веб-страницам.

Дорабатывает ввод данных в блог

Так как большинство записей датируются текущим днем, Руби решила сэкономить время, включив автозаполнение первого поля. А раз дата в результате будет вставляться автоматически, фокус ввода имеет смысл переместить на поле для сообщений. Это даст возможность приступить к набору новой записи сразу же после загрузки страницы. Разумеется, эти изменения не критичны для работы блога, они даже не связаны непосредственно с Ajax, но они позволяют чувствовать себя на странице более комфортно, что идеально отвечает духу Ajax.

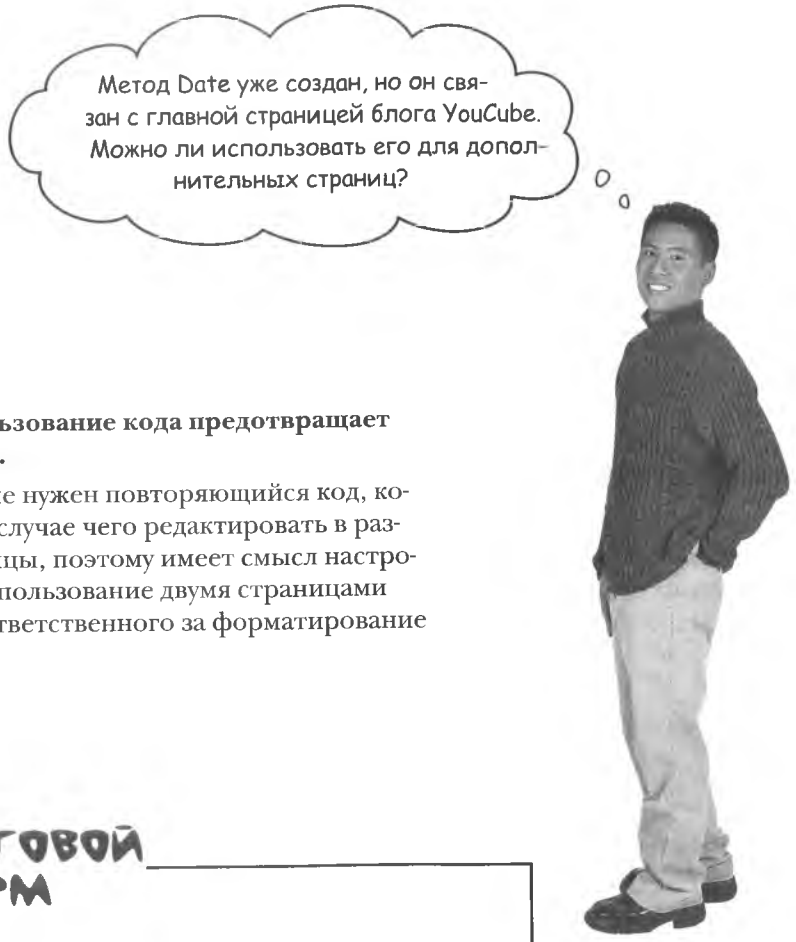
Автоматическая подстановка в поле текущей даты.



Фокус ввода установим на это поле, чтобы Руби сразу могла приступить к созданию новой записи.

Автозаполнение полей

Если помните, в блоге YouCube используется формат данных ММ/ДД/ГГГГ, значит, именно его нужно использовать для автоматического заполнения поля даты.



Повторное использование кода предотвращает его дублирование.

Нам совершенно не нужен повторяющийся код, который придется в случае чего редактировать в разных местах страницы, поэтому имеет смысл настроить совместное использование двумя страницами фрагмента кода, ответственного за форматирование даты.

МОЗГОВОЙ ШТУРМ

Каким образом организовать совместное использование метода `shortFormat()` разными страницами блога?

Повторяющаяся задача?

Совместное использование кода JavaScript различными страницами вынуждает нас поместить такой код в отдельный файл или модуль, который затем будет импортироваться на каждую страницу. Вы уже видели, как это делается, на примере объекта `AjaxRequest`, хранящегося в файле ajax.js. Вот как он импортируется:

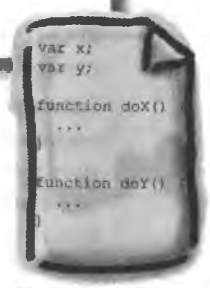
Имя внешнего файла JavaScript присвоено атрибуту src тега <script>.

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

Для импорта кода JavaScript из внешнего файла воспользуемся уже знакомым нам тегом <script>.

Метод `shortFormat()` объекта `Date` для этой цели поместим в файл date.js, который затем импортируем в каждую страницу блога YouCube.

```
Date.prototype.shortFormat = function() {

  return (this.getMonth() + 1) + "/" + this.getDate() + "/" + this.getFullYear();

}
```



date.js

Тег `<script>`, который мы использовали для кода Ajax, применим для импорта в страницы блога сценария из файла date.js.

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

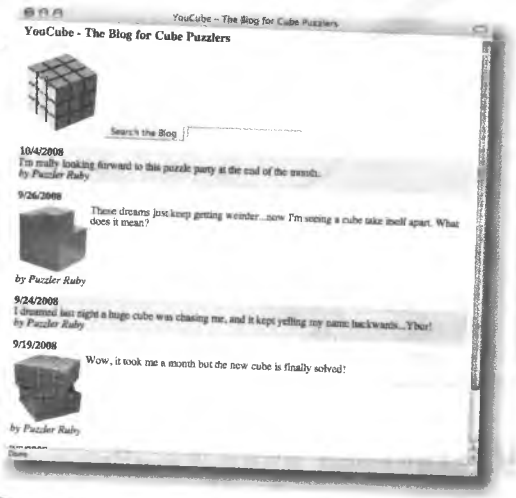
Содержимое файла date.js импортируется одним тегом <script>.

Сохранив код JavaScript во внешнем файле, мы получаем возможность использовать его на разных страницах.

Всегда имеет смысл помещать код многократного использования во внешний файл, импортируя его затем по мере надобности.

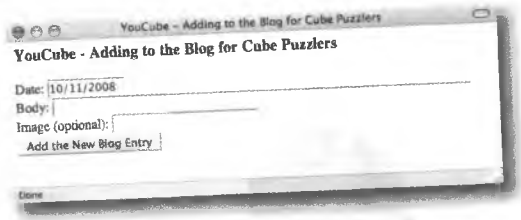
```
<div id="blog"></div>
```

Метод shortFormat() форматирует даты на главной странице блога YouCube...




```
<input type="text" id="date" name="date" value="" size="10" />
```

...а также на странице добавления новых записей.



Возьми в руку карандаш

Напишите код функции `initForm()`, вызываемой обработчиком события `onload` в сценарии добавления записей в блог. Функция должна вставлять в первое поле текущую дату и устанавливать фокус ввода на втором поле.

.....

.....

.....

.....

Возьми в руку карандаш

Решение

Вот как выглядит функция `initForm()`, осуществляющая автозаполнение поля date и установку фокуса ввода на поле body.

В поле date появляется текущая дата.

```
function initForm() {
  document.getElementById("date").value = (new Date()).shortFormat();
  document.getElementById("body").focus();
}
```

Фокус ввода устанавливается на поле body.

Увеличение продуктивности

Наконец-то Руби довольна тем, как работает ее блог YouCube. Благодаря Ajax он управляется данными, имеет дружелюбный интерфейс и внимателен к деталям, которые может оценить только настоящий фанат головоломок.

Сразу после загрузки страницы фокус ввода устанавливается на поле body.

После открытия страницы в поле date автоматически появляется текущая дата.