ВЫРАЗИТЕЛЬНЫЙ JAVASCRIPT

ВТОРОЕ ИЗДАНИЕ

Современное введение в программирование

Марейн Хавербек

no starch press

Содержание

Introduction	1.1
О программировании	1.2
Величины, типы и операторы	1.3
Структура программ	1.4
Функции	1.5
Структуры данных: объекты и массивы	1.6
Функции высшего порядка	1.7
Тайная жизнь объектов	1.8
Проект: электронная жизнь	1.9
Поиск и обработка ошибок	1.10
Регулярные выражения	1.11
Модули	1.12
Проект: язык программирования	1.13
JavaScript и браузер	1.14
Document Object Model	1.15
Обработка событий	1.16
Проект: игра-платформер	1.17
Рисование на холсте	1.18
HTTP	1.19
Формы и поля форм	1.20

Проект: Paint 1.21
Node.js 1.22
Проект: веб-сайт по обмену опытом 1.23

Выразительный Javascript

2-е издание

Автор: Marijn Haverbeke

Перевод: Вячеслав Голованов

Распространяется под лицензией Creative Commons Attribution-Noncommercial.

Исходный код в книге распростроняется под лицензией MIT.

Сборка в Gitbook: Антон Кармазин

Введение

Это книга рассказывает, как заставить компьютеры делать то, что вам от них нужно. Компьютеры сегодня так же распространены, как отвёртки – но содержат гораздо больше скрытых сложностей, и поэтому их сложнее понять и с ними сложнее работать. Для многих они остаются чуждыми, слегка угрожающими штуками.

Мы обнаружили два эффективных способа уменьшить коммуникационный разрыв между нами – водянистыми биологическими организмами, у которых есть талант к социальным связям и пространным рассуждениям, и компьютерами – бесчувственными манипуляторами, работающими с бессмысленными данными. Первый – обратиться к нашему ощущению физического мира, и строить интерфейсы, имитирующие его, чтобы мы могли при помощи пальцев манипулировать формами на экране. Для простого взаимодействия с компьютером это неплохо подходит.

Но мы не нашли хороший способ передавать компьютеру при помощи перемещений и нажатий мышью те вещи, которые дизайнер интерфейса не предусмотрел. Для того, чтобы взаимодействовать с компьютером на более сложных уровнях, например задавать ему произвольные задачи на выполнение, лучше подходит наш талант к общению: мы обучаем компьютер языку.

Человеческие языки позволяют комбинировать слова великим множеством способов, так, что мы можем сказать очень много разных вещей. Компьютерные языки устроены примерно так же, хотя и менее гибки грамматически.

За последние 20 лет работа с компьютером стала очень распространённым явлением, и интерфейсы, построенные на языке (а когда-то это был единственный способ общения с компьютером) почти вытеснены графическими. Но они всё ещё есть – если вы знаете, где их искать. Один из таких языков, JavaScript, встроен почти в любой веб-браузер, и потому доступен почти на каждом вычислительном устройстве.

Эта книга ставит целью познакомить вас с этим языком достаточно для того, чтобы вы могли заставить компьютер делать то, что вам нужно.

О программировании

Я не просвещаю тех, кто не жаждет учиться, и не побуждаю тех, кто не хочет искать ответы самостоятельно. Если я покажу один угол квадрата, и они не приходят ко мне с остальными тремя – мне не нужно давать повторных объяснений.

Конфуций

Кроме объяснения JavaScript я также хочу объяснить основные принципы программирования. Как выясняется, программировать тяжело. Обычно базовые принципы просты и понятны. Но программы, построенные на этих принципах, становятся сложными настолько, что вводят свои собственные правила и уровни сложности. Вы строите свой собственный лабиринт, и можете в нём потеряться.

Возможно, временами чтение будет разочаровывать вас. Если вы новичок в программировании, вам нужно будет много чего переварить. Много материала будет скомбинировано таким образом, что вам нужно будет установить новые связи между его частями.

Вы сами должны обосновать необходимость этих усилий. Если вам тяжело продираться через книгу, не нужно думать о себе плохо. С вами всё в порядке – вам нужно просто продолжать движение. Сделайте перерыв, вернитесь назад – и всегда удостоверяйтесь, что вы прочли и поняли примеры программ. Обучение – это сложная работа, но раз вы что-то выучили, оно уже принадлежит вам, и облегчает дальнейшие шаги.

Программист создаёт вселенные, за которые он один в ответе. В компьютерных программах могут быть созданы вселенные практически неограниченной сложности.

Джозеф Вайзенбаум, «Сила компьютеров и Разум людей»

Программа – сложное понятие. Это кусок текста, набранный программистом, это направляющая сила, заставляющая компьютер что-то делать, это данные в памяти компьютера, и при этом она контролирует работу с этой же самой памятью. Аналогии, которые пытаются сравнивать программы со знакомыми нам объектами обычно не справляются с этим. Одна более-менее подходящая – аналогия с машиной. Множество отдельных частей составляют одно целое, и чтобы

заставить её работать, нам нужно представлять себе способы, которыми эти части взаимодействуют и что они привносят в работу целой машины.

Компьютер – это машина, которая устроена так, чтобы содержать в себе эти нематериальные машинки. Компьютеры сами по себе могут выполнять только простые действия. Польза их в том, что они могут делать это очень быстро. Программа может очень хитрым образом комбинировать эти действия так, чтобы в результате выполнялись очень сложные действия.

Для некоторых из нас программирование – это увлекательная игра. Программа – это мысленная конструкция. Ничего не стоит её построить, она ничего не весит, и она легко вырастает под нашими пальцами.

Если не быть осторожным, размер и сложность выходят из-под контроля, запутывая даже того, кто её пишет. Это основная проблема программирования: сохранять контроль над программами. Когда программа работает – это прекрасно. Искусство программирования – это умение контролировать сложность. Большая программа находится под контролем, и выполнена просто в своей сложности.

Многие программисты верят, что этой сложностью лучше всего управлять, используя в программах небольшой набор хорошо известных техник. Они описали строгие правила («наилучшие практики») того, какую форму программы должны иметь. И самые ревностные среди них считают тех, кто отклоняется от этих практик, плохими программистами.

Что за враждебность по отношению к богатству программирования – попытки принизить его до чего-то прямолинейного и предсказуемого, наложить табу на всякие странные и прекрасные программы! Ландшафт техник программирования огромен, увлекателен своим разнообразием, и до сих пор изучен мало. Это опасное путешествие, заманивающее и запутывающее неопытного программиста, но это всего лишь означает, что вы должны следовать этим путём осторожно и думать головой. По мере обучения вам всегда будут встречаться новые задачи и новые неизведанные территории. Программисты, не изучающие новое, стагнируют, забывают свою радость, их работа наскучивает им.

Почему язык имеет значение

В начале, при зарождении компьютерных дисциплин, не было языков программирования. Программы выглядели так:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Это программа, складывающая числа от 1 до 10, и выводящая результат (1 + 2 +… + 10 = 55). Она может выполняться на очень простой гипотетической машине. Для программирования первых компьютеров было необходимо устанавливать большие массивы переключателей в нужные позиции, или пробивать дырки в перфокартах и скармливать их компьютеру. Можете представить, какая это была утомительная, подверженная ошибкам процедура. Написание даже простых программ требовало большого ума и дисциплины. Сложные программы были практически немыслимы.

Конечно, ручной ввод этих мистических диаграмм бит (нулей и единиц) давал программисту возможность ощутить себя волшебником. И это чего-то стоило в смысле удовлетворения работой.

Каждая строка указанной программы содержит одну инструкцию. На обычном языке их можно описать так:

```
1. записать 0 в ячейку памяти 0
2. записать 1 в ячейку памяти 1
3. записать значение ячейки 1 в ячейку 2
4. вычесть 11 из значения ячейки 2
5. если у ячейки 2 значение 0, тогда продолжить с пункта
 9.
6. добавить значение ячейки 1 к ячейке 0
7. добавить 1 к ячейке 1
8. продолжить с пункта 3.
9. вывести значение ячейки 0
```

Этот вариант легче прочесть, чем кучу бит, но он всё равно не очень удобен. Использование имён вместо номеров инструкций и ячеек памяти может улучшить понимание.

```
  установить ‘total’ в 0
  установить ‘count’ в 1
[loop]
  установить ‘compare’ в ‘count’
  вычесть 11 из ‘compare’
  если ‘compare’ равно нулю, перейти на [end]
  добавить ‘count’ к ‘total’
  добавить 1 к ‘count’
  перейти на [loop]
[end]
  вывести ‘total’
```

Вот теперь уже не так сложно понять, как работает программа. Справитесь? Первые две строки назначают двум областям памяти начальные значения. total будет использоваться для подсчёта результата вычисления, а count будет следить за числом, с которым мы работаем в данный момент. Строчки, использующие ‘compare’, наверно, самые странные. Программе нужно понять, не равно ли count 11, чтобы прекратить подсчёт. Так как наша воображаемая машина довольно примитивна, она может только выполнить проверку на равенство переменной нулю, и принять решение о том, надо ли перепрыгнуть на другую строку. Поэтому она использует область памяти под названием ‘compare’, чтобы подсчитать значение count – 11 и принять решение на основании этого значения. Следующие две строки

добавляют значение count в счетчик результата и увеличивают count на 1 каждый раз, когда программа решает, что ещё не достигла значения 11.

Вот та же программа на JavaScript:

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

Еще несколько улучшений. Главное – нет необходимости вручную обозначать переходы между строками. Конструкция языка while делает это сама. Она продолжает вычислять блок, заключённый в фигурные скобки, пока условие выполняется (count <=10), то есть значение count меньше или равно 10. Уже не нужно создавать временное значение и сравнивать его с нулём. Это было скучно, и сила языков программирования в том, что они помогают избавиться от скучных деталей.

В конце программы по завершению while к результату применяется операция console.log с целью вывода.

И наконец, вот так могла бы выглядеть программа, если б у нас были удобные операции range и sum, которые, соответственно, создавали бы набор номеров в заданном промежутке и подсчитывали сумму набора:

```
console.log(sum(range(1, 10)));
// → 55
```

Мораль сей басни – одна и та же программа может быть написана и долго, и коротко, читаемо и нечитаемо. Первая версия программы была совершенно смутной, а последняя – почти настоящий язык – записать сумму диапазона номеров от 1 до 10. В следующих главах мы рассмотрим, как делать такие вещи.

Хороший язык программирования помогает программисту сообщать компьютеру о необходимых операциях на высоком уровне. Позволяет опускать скучные детали, даёт удобные строительные блоки (while и console.log), позволяет создавать свои собственные блоки (sum и range), и делает простым комбинирование блоков.

Что такое JavaScript?

JavaScript был представлен в 1995 году как способ добавлять программы на веб-страницы в браузере Netscape Navigator. С тех пор язык прижился во всех основных графических браузерах. Он дал возможность появиться современным веб-приложениям – браузерные е-мейл-клиенты, карты, социальные сети. А ещё он используется на более традиционных сайтах для обеспечения интерактивности и всяких наворотов.

Важно отметить, что JavaScript практически не имеет отношения к другому языку под названием Java. Похожее имя было выбрано из маркетинговых соображений. Когда появился JavaScript, язык Java широко рекламировался и набирал популярность. Кое-кто решил, что неплохо бы прицепиться к этому паровозу. А теперь мы уже никуда не денемся от этого имени.

После того, как язык вышел за пределы Netscape, был составлен документ, описывающий работу языка, чтобы разные программы, заявляющие о его поддержке, работали одинаково. Он называется стандарт ECMAScript по имени организации ECMA. На практике можно говорить о ECMAScript и JavaScript как об одном и том же.

Многие ругают JavaScript и говорят о нём много плохого. И многое из этого – правда. Когда мне первый раз пришлось писать программу на JavaScript, я быстро

почувствовал отвращение – язык принимал практически всё, что я писал, при этом интерпретировал это вовсе не так, как я подразумевал. В основном это было из-за того, что я не имел понятия о том, что делаю, но тут есть и проблема: JavaScript слишком либерален. Задумывалось это как облегчение программирования для начинающих. В реальности, это затрудняет розыск проблем в программе, потому что система о них не сообщает.

Гибкость имеет свои преимущества. Она оставляет место для разных техник, невозможных в более строгих языках. Иногда, как мы увидим в главе «модули», её можно использовать для преодоления некоторых недостатков языка. После того, как я по-настоящему изучил и поработал с ним, я научился любить JavaScript.

Вышло уже несколько версий языка JavaScript. ECMAScript 3 была доминирующей, распространённой версией во время подъёма языка, примерно с 2000 до 2010. В это время готовилась амбициозная 4-я версия, в которой было запланировано несколько радикальных улучшений и расширений языка. Однако политические причины сделали изменение живого популярного языка очень сложным, и работа над 4-й версией была прекращена в 2008. Вместо неё вышла менее

амбициозная 5-я версия в 2009. Сейчас большинство браузеров поддерживает 5-ю версию, которую мы и будем использовать в книге.

JavaScript поддерживают не только браузеры. Базы данных типа MongoDB and CouchDB используют его в качестве скриптового языка и языка запросов. Есть несколько платформ для декстопов и серверов, наиболее известная из которых Node.js, предоставляющие мощное окружение для программирования вне браузера.

Код, и что с ним делать

Код – это текст, из которого состоят программы. В большинстве глав книги есть код. Чтение и написание кода – это неотъемлемая часть обучения программированию. Старайтесь не просто пробегать глазами примеры – читайте их внимательно и разбирайтесь. Сначала это будет происходить медленно и непонятно, но вы быстро овладеете навыками. То же – насчёт упражнений. Не подразумевайте, что вы в них разобрались, пока не напишете работающий вариант.

Рекомендую пробовать ваши решения в настоящем интерпретаторе языка, чтобы сразу получать отзыв, и, надеюсь, подвергаться искушению поэкспериментировать далее.

Вы можете установить Node.js и выполнять программы с его помощью. Также вы можете делать это в консоли браузера. В 12 главе будет объяснено, как встраивать программы в HTML-страницы. Также есть сайты типа jsbin.com, позволяющие просто запускать программы в браузере. На сайте книги есть песочница для кода.

Величины, типы и операторы

Под поверхностью машины движется программа. Без усилий, она расширяется и сжимается. Находясь в великой гармонии, электроны рассеиваются и собираются. Формы на мониторе – лишь рябь на воде. Суть остаётся скрытой внутри…

Мастер Юан-Ма, Книга программирования

В компьютерном мире есть только данные. Можно читать данные, изменять данные, создавать новые – но кроме данных ничего нет. Все данные хранятся как длинные последовательности бит, этим они сходны между собой.

Биты – это сущности с двумя состояниями, обычно описываемые как нули и единицы. В компьютере они живут в виде высоких и низких электрических зарядов, сильного или слабого сигнала, или блестящего и матового участка на поверхности CD. Каждая часть информации может быть представлена в виде последовательности нулей и единиц, то есть бит.

К примеру, номер 13. Вместо десятичной системы, состоящей из 10 цифр, у вас есть двоичная система с двумя цифрами. Значение каждой позиции числа удваивается при движении справа налево. Биты, составляющие число 13, вместе с их весами:

```
0   0   0   0 1 1 0 1
128 64 32  16 8 4 2 1
```

Получается двоичное число 00001101, или 8 + 4 + 1, что равно 13.

Величины

Представьте океан бит. Типичный современный компьютер хранит более 30 миллиардов бит в оперативной памяти. Постоянная память (жёсткий диск) обычно ещё на пару порядков объёмнее.

Чтобы работать с ними и не заблудиться, вы можете делить их на куски, представляющие единицы информации. В JavaScript эти куски называются величинами. Все они состоят из бит, но играют разные роли. У каждой величины есть тип, определяющий её роль. Всего есть шесть основных типов: числа, строки, булевые величины, объекты, функции и неопределённые величины.

Для создания величины вам нужно указать её имя. Это удобно. Вам не надо собирать стройматериалы или платить за них. Нужно просто позвать – и оп-па, готово. Они не создаются из воздуха – каждая величина где-то хранится, и если вы хотите использовать огромное их количество, у вас могут закончиться биты. К счастью, это только если они все нужны вам одновременно. Когда величина вам станет не нужна, она растворяется, и использованные ею биты поступают в переработку как стройматериал для новых величин.

В этой главе мы знакомимся с атомами программ JavaScript – простые типы величин и операторы, которые к ним применимы.

Числа

Величины числовых типов, это – сюрприз – числа. В программе JavaScript они записываются как

```
13
```

Используйте эту запись в программе, и она вызовет к жизни в компьютерной памяти цепочку бит, представляющую число 13.

JavaScript использует фиксированное число бит (64) для хранения численных величин. Число величин, которые можно выразить при помощи 64 бит, ограничено – то есть и сами числа тоже ограничены. Для N десятичных цифр количество чисел, которые ими можно записать, равно 10 в степени N. Аналогично, 64 битами можно выразить 2 в 64 степени чисел. Это довольно много.

Раньше у компьютеров памяти было меньше, и тогда для хранения чисел использовали группы из 8 или 16 бит. Было легко случайно превысить максимальное число для таких небольших чисел – то есть, использовать число, которое не помещалось в этот набор бит. Сегодня у компьютеров памяти много, можно использовать куски по 64 бит, и значит, вам надо беспокоиться об этом только, если вы работаете с астрономическими числами.

Правда, не все числа меньше 2^64 помещаются в число JavaScript. В этих битах также хранятся отрицательные числа – поэтому, один бит хранит знак числа. Кроме того, нам нужно иметь возможность хранить дроби. Для этого часть бит используется для хранения позиции десятичной точки. Реальный максимум для чисел – примерно 10^15, что в общем всё равно довольно много.

Дроби записываются с помощью точки.

```
9.81
```

Очень большие или маленькие числа записываются научной записью с буквой “e” (exponent), за которой следует степень:

```
2.998e8
```

Это 2.998 × 10^8 = 299800000.

Вычисления с целыми числами (которые также называются integer), меньшими, чем 10^15, гарантированно будут точными. Вычисления с дробями обычно нет. Так же, как число π (пи) нельзя представить точно при помощи конечного числа цифр, так и многие дроби нельзя представить в случае, когда у нас есть

только 64 бита. Плохо, но это мешает в очень специфических случаях. Важно помнить об этом и относиться к дробям как к приближённым значениям.

Арифметика

Главное, что можно делать с числами – это арифметические вычисления. Сложения и умножения используют два числа и выдают третье. Как это записывается в JavaScript:

```
100 + 4 * 11
```

Символы + и * называются операторами. Первый – сложение, второй – умножение. Помещаем оператор между двумя величинами и получаем значение выражения.

А в примере получается «сложить 4 и 100 и затем умножить результат на 11», или умножение выполняется сначала? Как вы могли догадаться, умножение выполняется первым. Но как и в математике, это можно изменить при помощи скобок:

```
(100 + 4) * 11
```

Для вычитания используется оператор -, а для деления - /

Когда операторы используются без скобок, порядок их выполнения определяется их приоритетом. У операторов * и / приоритет одинаковый, выше, чем у + и -, которые между собой равны по приоритету. При вычислении операторов с равным приоритетом они вычисляются слева направо:

```
1 - 2 + 1
```

вычисляется как (1 - 2) + 1

Пока беспокоиться о приоритетах не надо. Если сомневаетесь – используйте скобки.

Есть ещё один оператор, который вы не сразу узнаете. Символ % используется для получения остатка. X % Y – остаток от деления X на Y. 314 % 100 даёт 14, и 144 % 12 даёт 0. Приоритет у оператора такой же, как у умножения и деления. Математики для операции *нахождения остатка от деления* % могут использовать термин *сравнение по модулю*.

Специальные числа

В JavaScript есть три специальных значения, которые считаются числами, но ведут себя не как обычные числа.

Это `Infinity` и `-Infinity`, которые представляют положительную и отрицательную бесконечности. Infinity - 1 = Infinity, и так далее. Не надейтесь сильно на вычисления с бесконечностями, они не слишком строгие.

Третье число: `NaN`. Обозначает «not a number» (не число), хотя это величина числового типа. Вы можете получить её после вычислений типа 0 / 0, Infinity – Infinity, или других операций, которые не ведут к точным осмысленным результатам.

Строки

Следующий базовый тип данных – строки. Они используются для хранения текста. Записываются они в кавычках:

```
"Что посеешь, то из пруда"
'Баба с возу, потехе час'
```

Можно использовать как двойные, так и одинарные кавычки – главное использовать их вместе. Почти всё можно заключить в кавычки и сделать из этого строку. Но некоторые символы вызывают проблемы. Например,

сложно заключить кавычки в кавычки. Перевод строки тоже нельзя просто так заключить в них – строка должна идти одной строкой.

Для заключения специальных символов используется обратный слеш \. Он обозначает, что символ, идущий за ним, имеет специальное значение – это называется «экранирование символов» (escape character). \” можно заключать в двойные кавычки. \n обозначает перевод строки, \t – табуляцию.

Строка “Между первой и второй\nсимвол будет небольшой” на самом деле будет выглядеть так:

```
Между первой и второй
символ будет небольшой
```

Если вам нужно включить в строку обратный слеш, его тоже нужно экранировать: \\. Инструкцию “Символ новой строки — это “\n”” нужно будет написать так:

```
"Символ новой строки – это \"\\n\""
```

Строки нельзя делить, умножать и складывать. Однако с ними можно использовать оператор +, который будет соединять их друг с другом. Следующее выражение выдаст слово «соединение»:

```
"сое" + "ди" + "н" + "ение"
```

Есть много способов манипуляций со строками, которые мы обсудим в главе 4.

Унарные операторы

Не все операторы записываются символами – некоторые словами. Один из таких операторов – `typeof`, который выдаёт название типа величины, к которой он применяется.

```
console.log(typeof 4.5)
// → number

console.log(typeof "x")
// → string
```

Будем использовать вызов `console.log` в примерах, когда захотим увидеть результат на экране. Как именно будет выдан результат – зависит от окружения, в котором вы запускаете скрипт.

Предыдущие операторы работали с двумя величинами, однако typeof использует только одну. Операторы, работающие с двумя величинами, называются

бинарными, а с одной – унарными. Минус (вычитание) можно использовать и как унарный, и как бинарный.

```
console.log(- (10 - 2))
// → -8
```

Булевские величины

Часто вам нужна величина, которая просто показывает одну из двух возможностей – типа «да» и «нет», или «вкл» и «выкл». Для этого в JavaScript есть тип Boolean, у которого есть всего два значения – `true` и `false` (правда и ложь).

Сравнения

Один из способов получить булевские величины:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

Знаки < и > традиционно обозначают «меньше» и «больше». Это бинарные операторы. В результате их использования мы получаем булевскую величину, которая

показывает, является ли неравенство верным.

Строки можно сравнивать так же:

```
console.log("Арбуз" <  "Яблоко")
// → true
```

Строки сравниваются по алфавиту: буквы в верхнем регистре всегда «меньше» букв в нижнем регистре. Сравнение основано на стандарте Unicode. Этот стандарт присваивает номер практически любому символу из любого языка. Во время сравнения строк JavaScript проходит по их символам слева направо, сравнивая номерные коды этих символов.

Другие сходные операторы – это >= (больше или равно), <= (меньше или равно), == (равно), != (не равно).

```
console.log("Хочется" != "Колется")
// → true
```

В JavaScript есть только одна величина, которая не равна самой себе – `NaN` («не число»).

```
console.log(NaN == NaN)
// → false
```

NaN – это результат любого бессмысленного вычисления, поэтому он не равен результату какого-то другого бессмысленного вычисления.

Есть операции, которые можно совершать и с самими булевскими значениями. JavaScript поддерживает три логических оператора: и, или, нет.

Оператор && — логическое «и». Он бинарный, и его результат – правда, только если обе величины, к которым он применяется, тоже правда.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

Оператор || — логическое «или». Выдаёт true, если одна из величин true.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

«Нет» записывается при помощи восклицательного знака “!”. Это унарный оператор, который обращает данную величину на обратную. !true получается false, !false получается true.

При использовании логических и арифметических операторов не всегда ясно, когда нужны скобки. На практике вы можете справиться с этим, зная, что у || приоритет ниже всех, потом идёт &&, потом операторы сравнения, потом все остальные. Такой порядок был выбран для того, чтобы в выражениях типа следующего можно было использовать минимальное количество скобок:

```
1 + 1 == 2 && 10 * 10 > 50
```

Последний логический оператор не унарный и не бинарный – он тройной. Записывается при помощи вопросительного знака и двоеточия:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

Это условный оператор, у которого величина слева от вопросительного знака выбирает одну из двух величин, разделённых двоеточием. Когда величина слева true, выбираем первое значение. Когда false, второе.

Неопределённые значения

Существуют два специальных значения, null и undefined, которые используются для обозначения отсутствия осмысленного значения. Сами по себе они никакой информации не несут.

Много операторов, которые не выдают значения, возвращают undefined просто для того, чтобы что-то вернуть. Разница между undefined и null появилась в языке случайно, и обычно не имеет значения.

Автоматическое преобразование типов

Ранее я упоминал, что JavaScript позволяет выполнять любые, подчас очень странные программы. К примеру:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("пять" * 2)
// → NaN
console.log(false == 0)
// → true
```

Когда оператор применяется «не к тому» типу величин, JavaScript втихую преобразовывает величину к нужному типу, используя набор правил, которые не всегда соответствуют вашим ожиданиям. Это называется приведением типов (coercion). В первом выражении null превращается в 0, а “5” становится 5 (из строки – в число). Однако в третьем выражении + выполняет конкатенацию (объединение) строк, из-за чего 1 преобразовывается в “1” (из числа в строку).

Когда что-то неочевидное превращается в число (к примеру, “пять” или undefined), возвращается значение NaN. Последующие арифметические операции с NaN опять получают NaN. Если вы получили такое значение, поищите, где произошло случайное преобразование типов.

При сравнении величин одного типа через ==, легко предсказать, что вы должны получить true, если они одинаковые (исключая случай с NaN). Но когда типы различаются, JavaScript использует сложный и запутанный набор правил для сравнений. Обычно он пытается преобразовать тип одной из величин в тип другой. Когда с одной из сторон оператора возникает null или undefined, он выдаёт true только если обе стороны имеют значение null или undefined.

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

Последний пример демонстрирует полезный приём. Когда вам надо проверить, имеет ли величина реальное значение вместо null или undefined, вы просто сравниваете её с null при помощи == или !=.

Но что, если вам надо сравнить нечто с точной величиной? Правила преобразования типов в булевские значения говорят, что 0, NaN и пустая строка “” считаются false, а все остальные – true. Поэтому 0 == false и “” == false. В случаях, когда вам не нужно автоматическое преобразование типов, можно использовать ещё два оператора: === и !==. Первый проверяет, что две величины абсолютно идентичны, второй – наоборот. И тогда сравнение “” === false возвращает false.

Рекомендую использовать трёхсимвольные операторы сравнения для защиты от неожиданных преобразований типов, которые могут привести к непредсказуемым последствиям. Если вы уверены, что типы сравниваемых величин будут совпадать, можно спокойно использовать короткие операторы.

Короткое вычисление логических операторов

Логические операторы && и || работают с величинами разных типов очень странным образом. Они преобразуют величину с левой стороны оператора в булевскую, чтобы понять, что делать дальше, но в зависимости от оператора и от результата этого преобразования, возвращают оригинальное значение либо левой, либо правой части.

К примеру, || вернёт значение с левой части, когда его можно преобразовать в true – а иначе вернёт правую часть.

```
console.log(null || "user")
// → user
console.log("Karl" || "user")
// → Karl
```

Такая работа оператора || позволяет использовать его как откат к значению по умолчанию. Если вы дадите ему выражение, которое может вернуть пустое значение слева, то значение справа будет служить заменой на этот случай.

Оператор && работает сходным образом, но наоборот. Если величина слева преобразовывается в false, он возвращает эту величину, а иначе – величину справа.

Ещё одно важное их свойство – выражение в правой части вычисляется только при необходимости. В случае true || X неважно, чему равно X. Даже если это какое-то ужасное выражение. Результат всегда true и X не вычисляется. Так же работает false && X – X просто игнорируется. Это называется коротким вычислением.

Оператор условия работает так же. Первое выражение всегда вычисляется, а из второго и третьего значения – только то, которое оказывается выбранным в результате.

Итог

Мы рассмотрели четыре типа величин JavaScript: числа, строки, булевские и неопределённые.

Эти величины получаются, когда мы пишем их имена (true, null) или значения (13, “ёпрст”). Их можно комбинировать и изменять при помощи операторов. Для арифметики есть бинарные операторы (+, -, *, / и %), объединение строк (+), сравнение (==, !=, ===, !==, <, >, <=, >=) и логические операторы (&&, ||), а также несколько

унарных операторов (- для отрицательного значения, ! для логического отрицания и typeof для определения типа величины).

Эти знания позволяют использовать JavaScript в качестве калькулятора, но и только. В следующей главе мы будем связывать эти простые значения вместе, чтобы составлять простые программы.

Структура программ

Сердце моё сияет ярко-красным светом под моей тонкой, прозрачной кожей, и им приходится вколоть мне десять кубиков JavaScript, чтобы вернуть меня к жизни (я хорошо реагирую на токсины в крови). От этой фигни у вас враз жабры побледнеют!

_why, Why's (Poignant) Guide to Ruby

В этой главе мы начнём заниматься тем, что уже можно назвать программированием. Мы расширим использование языка JavaScript за пределы существительных и фрагментов предложений к более-менее осмысленной прозе.

Выражения и инструкции

В первой главе мы создавали величины и применяли к ним операторы, получая новые величины. Это важная часть каждой программы, но только лишь часть.

Фрагмент кода, результатом работы которого является некая величина, называется выражением. Каждая величина, записанная буквально (например, 22 или

“психоанализ”) тоже является выражением. Выражение, записанное в скобках, также является выражением, как и бинарный оператор, применяемый к двум выражениям или унарный – к одному.

Это часть красоты языкового интерфейса. Выражения могут включать другие выражения так же, как сложноподчинённое предложение состоит из простых. Это позволяет нам комбинировать выражения для создания вычислений любой сложности.

Если выражение – это фрагмент предложения, то инструкция – это предложение полностью. Программа – это просто список инструкций.

Простейшая инструкция – это выражение с точкой с запятой после него. Это — программа:

```
1;
!false;
```

Правда, это бесполезная программа. Выражение можно использовать только для получения величины, которая может быть использована в другом выражении, охватывающем это. Инструкция стоит сама по себе и её применение изменяет что-то в мире программы. Она может выводить что-то на экран (изменение в мире), или менять внутреннее состояние машины таким образом,

что это повлияет на следующие за ним инструкции. Эти изменения называются побочными эффектами. Инструкции в предыдущем примере просто выдают величины 1 и true, и сразу их выбрасывают. Они не оказывают никакого влияния на мир программы. При выполнении программы ничего заметного не происходит.

В некоторых случаях JavaScript позволяет опускать точку с запятой в конце инструкции. В других случаях она обязательна, или следующая строка будет расцениваться как часть той же инструкции. Правила, согласно которым можно или нельзя опускать точку с запятой, довольно сложны и увеличивают вероятность ошибиться. В этой книге мы не будем опускать точку с запятой, и я рекомендую делать так же в своих программах, пока вы не накопите опыт.

Переменные

Как же программа хранит внутреннее состояние? Как она его запоминает? Мы получали новые величины из старых, но старые величины это не меняло, а новые нужно было использовать сразу, или же они исчезали. Чтобы захватить и хранить их, JavaScript предлагает нечто под названием «переменная».

```
var caught = 5 * 5;
```

И это даёт нам второй вид инструкций. Специальное ключевое слово (keyword) `var` показывает, что в этой инструкции мы объявляем переменную. За ним идёт имя переменной, и, если мы сразу хотим назначить ей значение – оператор = и выражение.

Пример создаёт переменную под именем caught и использует её для захвата числа, которое получается в результате перемножения 5 и 5.

После определения переменной её имя можно использовать в выражениях. Величина переменной будет такой, какое значение в ней сейчас содержится. Пример:

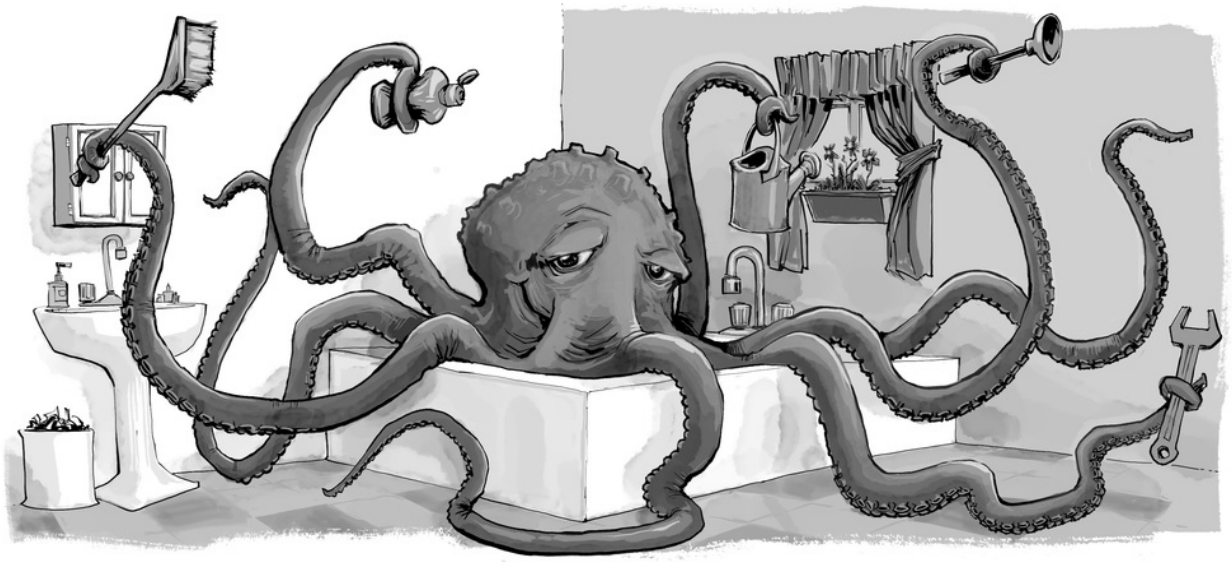
```
var ten = 10;
console.log(ten * ten);
// → 100
```

Переменные можно называть любым словом, которое не является ключевым (типа var). Нельзя использовать пробелы. Цифры тоже можно использовать, но не первым символом в названии. Нельзя использовать знаки пунктуации, кроме символов $ и _.

Переменной присваивают значение не навсегда. Оператор = можно использовать на существующих переменных в любое время, чтобы присвоить им новое значение.

```
var mood = "лёгкое";
console.log(mood);
// → лёгкое
mood = "тяжёлое";
console.log(mood);
// → тяжёлое
```

Представляйте себе переменные не в виде коробочек, а в виде щупалец. Они не содержат значения – они хватают их. Две переменные могут ссылаться на одно значение. Программа имеет доступ только к значениям, которые они содержат. Когда вам нужно что-то запомнить, вы отращиваете щупальце и держитесь за это, или вы используете существующее щупальце, чтобы удержать это.



Переменные как щупальца

Пример. Для запоминания количества денег, которые вам должен Василий, вы создаёте переменную. Затем, когда он выплачивает часть долга, вы даёте ей новое значение.

```
var vasyaDebt = 140;
vasyaDebt = vasyaDebt  - 35;
console.log(vasyaDebt);
// → 105
```

Когда вы определяете переменную без присваивания ей значения, щупальцу не за что держаться, оно висит в воздухе. Если вы запросите значение пустой переменной, вы получите undefined.

Одна инструкция var может содержать несколько переменных. Определения нужно разделять запятыми.

```
var one = 1, two = 2;
console.log(one + two);
// → 3
```

Ключевые и зарезервированные слова

Слова со специальным смыслом, типа var – ключевые. Их нельзя использовать как имена переменных. Также есть несколько слов, «зарезервированных для использования» в будуших версиях JavaScript. Их тоже нельзя использовать, хотя в некоторых средах исполнения это возможно. Полный их список достаточно большой.

```
break case catch continue debugger default delete do els
e false finally
for function if implements in instanceof interface let n
ew null package
private protected public return static switch throw true
 try typeof var
void while with yield this
```

Не нужно их запоминать, но имейте в виду, что ошибка может крыться здесь, если ваши определения переменных не работают, как надо.

Окружение

Коллекция переменных и их значений, которая существует в определённый момент, называется окружением. Когда программа запускается, окружение не пустое. Там всегда есть переменные, являющиеся частью программного стандарта, и большую часть времени там есть переменные, помогающие взаимодействовать с окружающей системой. К примеру, в браузере есть переменные и функции для изучения состояния загруженной веб-страницы и влияния на неё, для чтения ввода с мыши и клавиатуры.

Функции

Многие величины из стандартного окружения имеют тип `function` (функция). Функция – отдельный кусочек программы, который можно использовать вместе с другими величинами. К примеру, в браузере переменная alert содержит функцию, которая показывает небольшое окно с сообщением. Используют его так:

```
alert("С добрым утром!");
```

Диалог alert

Выполнение функции называют вызовом. Вы можете вызвать функцию, записав скобки после выражения, которое возвращает значение функции. Обычно вы напрямую используете имя функции в качестве выражения. Величины, которые можно написать внутри скобок, передаются программному коду внутри функции. В примере, функция alert использует данную ей строку для показа в диалоговом окне. Величины, передаваемые функциям, называются аргументами функций. Функция alert требует один аргумент, но другие могут требовать разное количество аргументов разных типов.

Функция console.log

Функция alert может использоваться как средство вывода при экспериментах, но закрывать каждый раз это окно вам скоро надоест. В прошлых примерах мы использовали функцию console.log для вывода значений. Большинство систем JavaScript (включая все современные браузеры и Node.js) предоставляют

функцию console.log, которая выводит величины на какое-либо устройство вывода. В браузерах это консоль JavaScript. Эта часть браузера обычно скрыта – большинство браузеров показывают её по нажатию F12, или Command-Option-I на Маке. Если это не сработало, поищите в меню “web console” или “developer tools”.

В примерах этой книги результаты вывода показаны в комментариях:

```
var x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

Хотя в именах переменных нельзя использовать точку – она, очевидно, содержится в названии console.log. Это оттого, что console.log – не простая переменная. Это выражение, возвращающее свойство log переменной console. Мы поговорим об этом в главе 4.

Возвращаемые значения

Показ диалогового окна или вывод текста на экран – это побочный эффект. Множество функций полезны оттого, что они производят эти эффекты. Функции также могут производить значения, и в этом случае им не нужен

побочный эффект для того, чтобы быть полезной. К примеру, функция Math.max принимает любое количество переменных и возвращает значение самой большой:

```
console.log(Math.max(2, 4));
// → 4
```

Когда функция производит значение, говорят, что она возвращает значение. Всё, что производит значение – это выражение, то есть вызовы функций можно использовать внутри сложных выражений. К примеру, возвращаемое функцией Math.min (противоположность Math.max) значение используется как один из аргументов оператора сложения:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

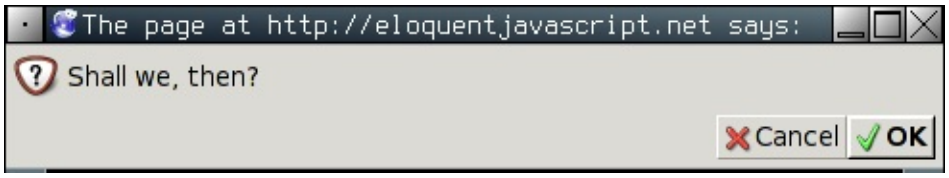
В следующей главе описано, как писать собственные функции.

prompt и confirm

Окружение браузера содержит другие функции, кроме alert, которые показывают всплывающие окна. Можно вызвать окно с вопросом и кнопками OK/Cancel при

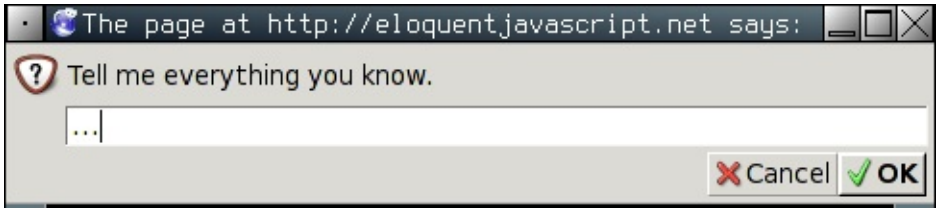
помощи функции `confirm`. Она возвращает булевское значение – true, если нажато OK, и false, если нажато Cancel.

```
confirm("Ну что, поехали?");
```



Функцию `prompt` можно использовать, чтобы задать открытый вопрос. Первый аргумент – вопрос, второй – текст, с которого пользователь начинает. В диалоговое окно можно вписать строку текста, и функция вернёт его в виде строки.

```
prompt("Расскажи мне всё, что знаешь.", "...");
```



Эти функции нечасто используют, потому что нельзя изменять внешний вид этих окон — но они могут пригодиться для экспериментальных программ.

Управление порядком выполнения программы

Когда в программе больше одной инструкции, они выполняются сверху вниз. В этом примере у программы две инструкции. Первая спрашивает число, вторая, выполняемая следом, показывает его квадрат.

```
var theNumber = Number(prompt("Выбери число", ""));
alert("Твоё число – квадратный корень из " +  theNumber
* theNumber);
```

Функция Number преобразовывает величину в число. Нам это нужно, потому что prompt возвращает строку. Есть сходные функции String и Boolean, преобразующие величины в соответствующие типы.

Простая схема прямого порядка исполнения программы:

Условное выполнение

Выполнять инструкции по порядку – не единственная возможность. В качестве альтернативы существует условное выполнение, где мы выбираем из двух возможных путей, основываясь на булевской величине:

Условное выполнение записывается при помощи ключевого слова if. В простом случае нам нужно, чтобы некий код был выполнен, только если выполняется некое условие. К примеру, в предыдущей программе мы можем считать квадрат, только если было введено именно число.

```
var theNumber = prompt("Выбери число ", "");
if (!isNaN(theNumber))
  alert("Твоё число – квадратный корень из " +  theNumbe
r * theNumber);
```

Теперь, введя «сыр», вы не получите вывод.

Ключевое слово if выполняет или пропускает инструкцию, в зависимости от значения булевого выражения. Это выражение записывается после if в скобках, и за ним идёт нужная инструкция.

Функция `isNaN` – стандартная функция JavaScript, которая возвращает true, только если её аргумент – NaN (не число). Функция Number возвращает NaN, если задать ей строку, которая не представляет собой

допустимое число. В результате, условие звучит так: «выполнить, если только theNumber не является не-числом».

Часто нужно написать код не только для случая, когда выражение истинно, но и для случая, когда оно ложно. Путь с вариантами – это вторая стрелочка диаграммы. Ключевое слово else используется вместе с if для создания двух раздельных путей выполнения.

```
var theNumber = Number(prompt("Выбери число", ""));
if (!isNaN(theNumber))
  alert("Твоё число – квадратный корень из " + theNumber
 * theNumber);
else
  alert("Ну ты что число-то не ввёл?");
```

Если вам нужно больше разных путей, можно использовать несколько пар if/else по цепочке.

```
var num = Number(prompt("Выбери число", "0"));

if (num < 10)
  alert("Маловато");
else if (num < 100)
  alert("Нормально");
else
  alert("Многовато");
```

Программа проверяет, действительно ли num меньше 10. Если да – выбирает эту ветку, и показывает «Маловато». Если нет, выбирает другую – на которой ещё один if. Если следующее условие выполняется, значит номер будет между 10 и 100, и выводится «Нормально». Если нет – значит, выполняется последняя ветка.

Последовательность выполнения примерно такая:

Циклы while и do

Представьте программу, выводящую все чётные числа от 0 до 12. Можно записать её так:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

Это работает – но смысл программирования в том, чтобы работать меньше, чем компьютер, а не наоборот. Если б нам понадобились все числа до 1000, это решение было бы неприемлемым. Нам нужна возможность повторения. Этот вид контроля над порядком выполнения называется циклом.

Зацикливание даёт возможность вернуться назад к какой-то инструкции и повторить всё заново с новым состоянием программы. Если скомбинировать это с переменной для подсчёта, можно сделать следующее:

```
var number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … и т.д.
```

Инструкция, начинающаяся с ключевого слова `while` – это цикл. За while следует выражение в скобках, и затем инструкция (тело цикла) – так же, как у if. Цикл выполняет инструкцию, пока выражение выдаёт истинный результат.

В цикле нам нужно выводить значение и прибавлять к нему. Если нам нужно выполнять в цикле несколько инструкций, мы заключаем его в фигурные скобки { }. Фигурные скобки для инструкций – как круглые скобки для выражений. Они группируют их и превращают в единое. Последовательность инструкций, заключённая в фигурные скобки, называется блоком.

Много программистов заключают любое тело цикла в скобки. Они делают это для единообразия, и для того, чтобы не нужно было добавлять и убирать скобки, если приходится изменять количество инструкций в цикле. В книге я не буду писать скобки вокруг единичных инструкций в цикле, так как люблю краткость. Вы можете делать, как угодно.

Переменная number показывает, как переменная может отслеживать прогресс программы. При каждом повторении цикла number увеличивается на 2. Перед каждым повторением оно сравнивается с 12, чтобы понять, сделала ли программа всё, что требовалось.

Для примера более полезной работы мы можем написать программу вычисления 2 в 10 степени. Мы используем две переменные: одну для слежения за результатом, а вторую – для подсчёта количества умножений. Цикл проверяет, достигла ли вторая переменная 10, и затем обновляет обе.

```
var result = 1;
var counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

Можно начинать counter с 1 и проверять его на <=10, но по причинам, которые станут ясны далее, всегда лучше начинать счётчики с 0.

Цикл do похож на цикл while. Отличается только в одном: цикл do всегда выполняет тело хотя бы один раз, а проверяет условие после первого выполнения. Поэтому и тестируемое выражение записывают после тела цикла:

```
do {
  var name = prompt("Who are you?");
} while (!name);
console.log(name);
```

Эта программа заставляет ввести имя. Она спрашивает его снова и снова, пока не получит что-то кроме пустой строки. Добавление "!" превращает значение в булевское и затем применяет логическое отрицание, а все строки, кроме пустой, преобразуются в булевское true.

Вы, наверно, заметили пробелы перед некоторыми инструкциями. В JavaScript это не обязательно – программа отработает и без них. Даже переводы строк не обязательно делать. Можно написать программу в одну строку. Роль пробелов в блоках – отделять их от остальной программы. В сложном коде, где в блоках встречаются другие блоки, может быть сложно разглядеть, где кончается один и начинается другой. Правильно отделяя их пробелами вы приводите в соответствие внешний вид кода и его блоки. Я люблю отделять каждый блок двумя пробелами, но вкусы различаются – некоторые используют четыре, некоторые – табуляцию. Чем больше пробелов использовать, тем заметнее отступ, но тем быстрее вложенные блоки убегают за правый край экрана.

Циклы for

Много циклов строятся по такому шаблону, как в примере. Создаётся переменная-счётчик, потом идёт цикл while, где проверочное выражение обычно проверяет, не достигли ли мы какой-нибудь границы. В конце тела цикла счётчик обновляется.

Поскольку это такой частый случай, в JavaScript есть вариант покороче, цикл `for`.

```
for (var number = 0; number <= 12; number = number + 2)
  console.log(number);
// → 0
// → 2
//   … и т.д.
```

Эта программа эквивалентна предыдущей. Только теперь все инструкции, относящиеся к отслеживанию состояния цикла, сгруппированы.

Скобки после for содержат две точки с запятой, разделяя инструкцию на три части. Первая инициализирует цикл, обычно задавая начальное значение переменной. Вторая – выражение проверки необходимости продолжения цикла. Третья – обновляет состояние после каждого прохода. В большинстве случаев такая запись более короткая и понятная, чем while.

Вычисляем 2^10 при помощи for:

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1
)
  result = result * 2;
console.log(result);
// → 1024
```

Хотя я не писал фигурных скобок, я отделяю тело цикла пробелами.

Выход из цикла

Дождаться, пока условие цикла не станет ложным – не единственный способ закончить цикл. Специальная инструкция `break` приводит к немедленному выходу из цикла.

В следующем примере мы покидаем цикл, когда находим число, большее 20 и делящееся на 7 без остатка.

```
for (var current = 20; ; current++) {
  if (current % 7 == 0)
    break;
}
console.log(current);
// → 21
```

Конструкция for не имеет проверочной части – поэтому цикл не остановится, пока не сработает инструкция break.

Если вы не укажете эту инструкцию, или случайно напишете условие, которое всегда выполняется, программа зависнет в бесконечном цикле и никогда не закончит работу – обычно это плохо.

Если вы сделаете бесконечный цикл, обычно через несколько секунд среда исполнения предложит вам прервать его. Если нет, вам придётся закрыть закладку, или даже весь браузер.

Ключевое слово `continue` также влияет на исполнение цикла. Когда это слово встречается в цикле, он немедленно переходит на следующую итерацию.

Короткое обновление переменных

Особенно часто в циклах программе нужно обновить переменную, основываясь на её предыдущем состоянии.

```
counter = counter + 1;
```

В JavaScript есть для этого короткая запись:

```
counter += 1;
```

Подобные записи работают для многих других операторов, к примеру result *= 2 для удвоения, или counter -= 1 для обратного отсчёта.

Это позволяет нам сократить программу вывода чётных чисел:

```
for (var number = 0; number <= 12; number += 2)
  console.log(number);
```

Для counter += 1 и counter -= 1 есть ещё более короткие записи: counter++ и counter--.

Работаем с переменными при помощи switch

Часто код выглядит так:

```
if (variable == "value1") action1();
else if (variable == "value2") action2();
else if (variable == "value3") action3();
else defaultAction();
```

Существует конструкция под названием `switch`, которая упрощает подобную запись. К сожалению, синтаксис JavaScript в этом случае довольно странный – часто цепочка if/else выглядит лучше. Пример:

```
switch (prompt("Как погодка?")) {
  case "дождь":
    console.log("Не забудь зонт.");
    break;
  case "снег":
    console.log("Блин, мы в России!");
    break;
  case "солнечно":
    console.log("Оденься полегче.");
  case "облачно":
    console.log("Иди гуляй.");
    break;
  default:
    console.log("Непонятная погода!");
    break;
}
```

В блок switch можно поместить любое количество меток `case`. Программа перепрыгивает на метку, соответствующую значению переменной в switch, или на метку `default`, если подходящих меток не найдено. После этого инструкции исполняются до первой инструкции `break` – даже если мы уже прошли другую метку. Иногда это можно использовать для исполнения одного и того же кода в разных случаях (в обоих случаях «солнечно» и «облачно» программа порекомендует пойти погулять). Однако, очень легко забыть запись break, что приведёт к выполнению нежелательного участка кода.

Регистр имён

Имена переменных не могут содержать пробелы, однако часто удобно использовать несколько слов для понятного описания переменной. Вы можете выбирать из нескольких вариантов:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

Первый довольно сложно читать. Мне нравятся подчёркивания, хотя их не очень удобно печатать. Стандартные функции JavaScript и большинство программистов используют последний вариант – каждое слово с большой буквы, кроме первого.

В некоторых случаях, например в случае функции Number, первую букву тоже пишут большой – когда нужно выделить функцию как конструктор. О конструкторах мы поговорим в главе 6. Сейчас просто не обращайте на это внимания.

Комментарии

Часто код не содержит всю информацию, которую хотелось бы передать читателям-людям, или доносит её в непонятном виде. Иногда вы чувствуете поэтическое вдохновение, или просто хотите поделиться мыслями в своей программе. Для этого служат комментарии.

Комментарий – это текст, который записан в программе, но игнорируется компьютером. В JavaScript комментарии можно писать двумя способами. Для однострочного комментария можно использовать два слеша:

```
var accountBalance = calculateBalance(account);
// Издалека долго
accountBalance.adjust();
// Течёт река Волга
var report = new Report();
// Течёт река Волга
addToReport(accountBalance, report);
// Конца и края нет
```

Комментарий продолжается только до конца строки. Код между символами /* и */ будет игнорироваться вместе с возможными переводами строки. Это подходит для включения целых информационных блоков в программу:

```
/*
Этот город – самый лучший
Город на Земле.
Он как будто нарисован
Мелом на стене.
*/
var myCity = 'Челябинск';
```

Итог

Теперь вы знаете, что программа состоит из инструкций, которые сами могут содержать инструкции. В инструкциях содержатся выражения, которые могут состоять из выражений.

Записывая инструкции подряд, мы получаем программу, которая выполняется сверху вниз. Вы можете изменять этот поток выполнения, используя условные (if, else и switch) операторы и операторы цикла (while, do и for).

Переменные можно использовать для хранения кусочков данных под определённым названием и для отслеживания состояния программы. Окружение – набор определённых переменных. Системы, исполняющие JavaScript, всегда добавляют несколько стандартных переменных в ваше окружение.

Функции – особые переменные, включающие части программы. Их можно вызвать командой functionName(argument1, argument2). Такой вызов – это выражение и может выдавать значение.

Упражнения

Каждое упражнение начинается с описания задачи. Прочтите и постарайтесь выполнить. В сложных ситуациях обращайтесь к подсказкам. Готовые решения задач можно найти на сайте книги eloquentjavascript.net/code/. Чтобы обучение было эффективным, не заглядывайте в ответы, пока не решите задачу сами, или хотя бы не попытаетесь её решить достаточно долго для того, чтобы у вас слегка заболела голова. Там же можно писать код прямо в браузере и выполнять его.

Треугольник в цикле

Напишите цикл, который за 7 вызовов console.log выводит такой треугольник:

```
#
##
###
####
#####
######
#######
```

Будет полезно знать, что длину строки можно узнать, приписав к переменной .length.

```
var abc = "abc";
console.log(abc.length);
// → 3
```

FizzBuzz

Напишите программу, которая выводит через console.log все числа от 1 до 100, с двумя исключениями. Для чисел, нацело делящихся на 3, она должна выводить ‘Fizz’, а для чисел, делящихся на 5 (но не на 3) – ‘Buzz’.

Когда сумеете – исправьте её так, чтобы она выводила «FizzBuzz» для всех чисел, которые делятся и на 3, и на 5.

(На самом деле, этот вопрос подходит для собеседований, и, говорят, он позволяет отсеивать довольно большое число кандидатов. Поэтому, когда вы

решите эту задачу, можете себя похвалить.)

Шахматная доска

Напишите программу, создающую строку, содержащую решётку 8x8, в которой линии разделяются символами новой строки. На каждой позиции либо пробел, либо #. В результате должна получиться шахматная доска.

```
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
```

Когда справитесь, сделайте размер доски переменным, чтобы можно было создавать доски любого размера.

Функции

Люди считают, что компьютерные науки – это искусство для гениев. В реальности всё наоборот – просто множество людей делают вещи, которые стоят друг на друге, будто составляя стену из маленьких камушков.

Дональд Кнут

Вы уже видели вызовы функций, таких как `alert`. Функции – это хлеб с маслом программирования на JavaScript. Идея оборачивания куска программы и вызова её как переменной очень востребована. Это инструмент для структурирования больших программ, уменьшения повторений, назначения имён подпрограммам, и изолирование подпрограмм друг от друга.

Самое очевидное использование функций – создание нового словаря. Придумывать слова для обычной человеческой прозы – дурной тон. В языке программирования это необходимо.

Средний взрослый русскоговорящий человек знает примерно 10000 слов. Редкий язык программирования содержит 10000 встроенных команд. И словарь языка

программирования определён чётче, поэтому он менее гибок, чем человеческий. Поэтому нам обычно приходится добавлять в него свои слова, чтобы избежать излишних повторений.

Определение функции

Определение функции – обычное определение переменной, где значение, которое получает переменная, является функцией. Например, следующий код определяет переменную square, которая ссылается на функцию, подсчитывающую квадрат заданного числа:

```
var square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

Функция создаётся выражением, начинающимся с ключевого слова `function`. У функций есть набор параметров (в данном случае, только x), и тело, содержащее инструкции, которые необходимо выполнить при вызове функции. Тело функции всегда заключают в фигурные скобки, даже если оно состоит из одной инструкции.

У функции может быть несколько параметров, или вообще их не быть. В следующем примере makeNoise не имеет списка параметров, а у power их целых два:

```
var makeNoise = function() {
  console.log("Хрясь!");
};

makeNoise();
// → Хрясь!

var power = function(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
};

console.log(power(2, 10));
// → 1024
```

Некоторые функции возвращают значение, как power и square, другие не возвращают, как makeNoise, которая производит только побочный эффект. Инструкция `return` определяет значение, возвращаемое функцией. Когда обработка программы доходит до этой инструкции, она сразу же выходит из функции, и возвращает это значение в то место кода, откуда была вызвана функция. return без выражения возвращает значение `undefined`.

Параметры и область видимости

Параметры функции – такие же переменные, но их начальные значения задаются при вызове функции, а не в её коде.

Важное свойство функций в том, что переменные, созданные внутри функции (включая параметры), локальны внутри этой функции. Это означает, что в примере с power переменная result будет создаваться каждый раз при вызове функции, и эти отдельные её инкарнации никак друг с другом не связаны.

Эта локальность переменных применяется только к параметрам и созданным внутри функций переменным. Переменные, заданные снаружи какой бы то ни было функции, называются глобальными, поскольку они видны на протяжении всей программы. Получить доступ к таким переменным можно и внутри функции, если только вы не объявили локальную переменную с тем же именем.

Следующий код иллюстрирует это. Он определяет и вызывает две функции, которые присваивают значение переменной x. Первая объявляет её как локальную, тем

самым меняя только локальную переменную. Вторая не объявляет, поэтому работа с x внутри функции относится к глобальной переменной x, заданной в начале примера.

```
var x = "outside";

var f1 = function() {
  var x = "inside f1";
};
f1();
console.log(x);
// → outside

var f2 = function() {
  x = "inside f2";
};
f2();
console.log(x);
// → inside f2
```

Такое поведение помогает предотвратить случайное взаимодействие между функциями. Если бы все переменные использовались в любом месте программы, было бы очень трудно убедиться, что одна переменная не используется по разным назначениям. А если бы вы использовали переменную повторно, вы бы столкнулись со странными эффектами, когда сторонний код портит значения вашей переменной. Относясь к локальным для функций переменным так, что они существуют только

внутри функции, язык делает возможной работу с функциями будто с отдельными маленькими вселенными, что позволяет не волноваться про весь код целиком.

Вложенные области видимости

JavaScript различает не только глобальные и локальные переменные. Функции можно задавать внутри функций, что приводит к нескольким уровням локальности.

К примеру, следующая довольно бессмысленная функция содержит внутри ещё две:

```
var landscape = function() {
  var result = "";
  var flat = function(size) {
    for (var count = 0; count < size; count++)
      result += "_";
  };
  var mountain = function(size) {
    result += "/";
    for (var count = 0; count < size; count++)
      result += "'";
    result += "\\";
  };

  flat(3);
  mountain(4);
  flat(6);
  mountain(1);
  flat(1);
  return result;
};

console.log(landscape());
// → ___/''''\______/'\_
```

Функции flat и mountain видят переменную result, потому что они находятся внутри функции, в которой она определена. Но они не могут видеть переменные count друг друга, потому что переменные одной функции находятся вне области видимости другой. А окружение снаружи функции landscape не видит ни одной из переменных, определённых внутри этой функции.

Короче говоря, в каждой локальной области видимости можно увидеть все области, которые её содержат. Набор переменных, доступных внутри функции, определяется местом, где эта функция описана в программе. Все переменные из блоков, окружающих определение функции, видны – включая и те, что определены на верхнем уровне в основной программе. Этот подход к областям видимости называется лексическим.

Люди, изучавшие другие языки программирования, могут подумать, что любой блок, заключённый в фигурные скобки, создаёт своё локальное окружение. Но в JavaScript область видимости создают только функции. Вы можете использовать отдельно стоящие блоки:

```
var something = 1;
{
  var something = 2;
  // Делаем что-либо с переменной something...
}
// Вышли из блока...
```

Но something внутри блока – это та же переменная, что и снаружи. Хотя такие блоки и разрешены, имеет смысл использовать их только для команды if и циклов.

Если это кажется вам странным – так кажется не только вам. В версии JavaScript 1.7 появилось ключевое слово let, которое работает как var, но создаёт переменные, локальные для любого данного блока, а не только для функции.

Функции как значения

Имена функций обычно используют как имя для кусочка программы. Такая переменная однажды задаётся и не меняется. Так что легко перепутать функцию и её имя.

Но это – две разные вещи. Вызов функции можно использовать, как простую переменную – например, использовать их в любых выражениях. Возможно хранить вызов функции в новой переменной, передавать её как параметр другой функции, и так далее. Также переменная, хранящая вызов функции, остаётся обычной переменной и её значение можно поменять:

```
var launchMissiles = function(value) {
  missileSystem.launch("пли!");
};
if (safeMode)
  launchMissiles = function(value) {/* отбой */};
```

В главе 5 мы обсудим чудесные вещи, которые возможно сделать, передавая вызовы функций другим функциям.

Объявление функций

Есть более короткая версия выражения “var square = function…”. Ключевое слово function можно использовать в начале инструкции:

```
function square(x) {
  return x * x;
}
```

Это объявление функции. Инструкция определяет переменную square и присваивает ей заданную функцию. Пока всё ок. Есть только один подводный камень в таком определении.

```
console.log("The future says:", future());

function future() {
  return "We STILL have no flying cars.";
}
```

Такой код работает, хотя функция объявляется ниже того кода, который её использует. Это происходит оттого, что объявления функций не являются частью обычного

исполнения программ сверху вниз. Они «перемещаются» наверх их области видимости и могут быть вызваны в любом коде в этой области. Иногда это удобно, потому что вы можете писать код в таком порядке, который выглядит наиболее осмысленно, не беспокоясь по поводу необходимости определять все функции выше того места, где они используются.

А что будет, если мы поместим объявление функции внутрь условного блока или цикла? Не надо так делать. Исторически разные платформы для запуска JavaScript обрабатывали такие случаи по разному, а текущий стандарт языка запрещает так делать. Если вы хотите, чтобы ваши программы работали последовательно, используйте объявления функций только внутри других функций или основной программы.

```
function example() {
  function a() {} // Нормуль
  if (something) {
    function b() {} // Ай-яй-яй!
  }
}
```

Стек вызовов

Полезным будет присмотреться к тому, как порядок выполнения работает с функциями. Вот простая программа с несколькими вызовами функций:

```
function greet(who) {
  console.log("Привет, " + who);
}
greet("Семён");
console.log("Покеда");
```

Обрабатывается она примерно так: вызов greet заставляет проход прыгнуть на начало функции. Он вызывает встроенную функцию console.log, которая перехватывает контроль, делает своё дело и возвращает контроль. Потом он доходит до конца greet, и возвращается к месту, откуда его вызвали. Следующая строчка опять вызывает console.log.

Схематично это можно показать так:

```
top
   greet
        console.log
   greet
top
   console.log
top
```

Поскольку функция должна вернуться на то место, откуда её вызвали, компьютер должен запомнить контекст, из которого была вызвана функция. В одном случае, console.log должна вернуться обратно в greet. В другом, она возвращается в конец программы.

Место, где компьютер запоминает контекст, называется стеком. Каждый раз при вызове функции, текущий контекст помещается наверх стека. Когда функция возвращается, она забирает верхний контекст из стека и использует его для продолжения работы.

Хранение стека требует места в памяти. Когда стек слишком сильно разрастается, компьютер прекращает выполнение и выдаёт что-то вроде “stack overflow” или “too much recursion”. Следующий код это демонстрирует – он задаёт компьютеру очень сложный вопрос, который приводит к бесконечным прыжкам между двумя функциями. Точнее, это были бы бесконечные прыжки, если бы у компьютера был бесконечный стек. В реальности стек переполняется.

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

Необязательные аргументы

Следующий код вполне разрешён и выполняется без проблем:

```
alert("Здрасьте", "Добрый вечер", "Всем привет!");
```

Официально функция принимает один аргумент. Однако, при таком вызове она не жалуется. Она игнорирует остальные аргументы и показывает «Здрасьте».

JavaScript очень лоялен по поводу количества аргументов, передаваемых функции. Если вы передадите слишком много, лишние будут проигнорированы. Слишком мало – отсутствующим будет назначено значение undefined.

Минус этого подхода в том, что возможно – и даже вероятно – передать функции неправильное количество аргументов, и вам никто на это не пожалуется.

Плюс в том, что вы можете создавать функции, принимающие необязательные аргументы. К примеру, в следующей версии функции power её можно вызывать как с двумя, так и с одним аргументом. В последнем случае экспонента будет равна двум, и функция работает как квадрат.

```
function power(base, exponent) {
  if (exponent == undefined)
    exponent = 2;
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}

console.log(power(4));
// → 16
console.log(power(4, 3));
// → 64
```

В следующей главе мы увидим, как в теле функции можно узнать точное число переданных ей аргументов. Это полезно, т. к. позволяет создавать функцию,

принимающую любое количество аргументов. К примеру, console.log использует это свойство, и выводит все переданные ей аргументы:

```
console.log("R", 2, "D", 2);
// → R 2 D 2
```

Замыкания

Возможность использовать вызовы функций как переменные вкупе с тем фактом, что локальные переменные каждый раз при вызове функции создаются заново, приводит нас к интересному вопросу. Что происходит с локальными переменными, когда функция перестаёт работать?

Следующий пример иллюстрирует этот вопрос. В нём объявляется функция wrapValue, которая создаёт локальную переменную. Затем она возвращает функцию, которая читает эту локальную переменную и возвращает её значение.

```
function wrapValue(n) {
  var localVariable = n;
  return function() { return localVariable; };
}

var wrap1 = wrapValue(1);
var wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

Это допустимо и работает так, как должно – доступ к переменной остаётся. Более того, в одно и то же время могут существовать несколько экземпляров одной и той же переменной, что ещё раз подтверждает тот факт, что с каждым вызовом функции локальные переменные пересоздаются.

Эта возможность работать со ссылкой на какой-то экземпляр локальной переменной называется замыканием. Функция, замыкающая локальные переменные, называется замыкающей. Она не только освобождает вас от забот, связанных с временем жизни переменных, но и позволяет творчески использовать функции.

С небольшим изменением мы превращаем наш пример в функцию, умножающую числа на любое заданное число.

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

var twice = multiplier(2);
console.log(twice(5));
// → 10
```

Отдельная переменная вроде localVariable из примера с wrapValue уже не нужна. Так как параметр – сам по себе локальная переменная.

Потребуется практика, чтобы начать мыслить подобным образом. Хороший вариант мысленной модели – представлять, что функция замораживает код в своём теле и обёртывает его в упаковку. Когда вы видите return function(...) {...}, представляйте, что это пульт управления куском кода, замороженным для употребления позже.

В нашем примере multiplier возвращает замороженный кусок кода, который мы сохраняем в переменной twice. Последняя строка вызывает функцию, заключённую в переменной, в связи с чем активируется сохранённый код (return number * factor;). У него всё ещё есть доступ к переменной factor, которая определялась при вызове

multiplier, к тому же у него есть доступ к аргументу, переданному во время разморозки (5) в качестве числового параметра.

Рекурсия

Функция вполне может вызывать сама себя, если она заботится о том, чтобы не переполнить стек. Такая функция называется рекурсивной. Вот пример альтернативной реализации возведения в степень:

```
function power(base, exponent) {
  if (exponent == 0)
    return 1;
  else
    return base * power(base, exponent - 1);
}

console.log(power(2, 3));
// → 8
```

Примерно так математики определяют возведение в степень, и, возможно, это описывает концепцию более элегантно, чем цикл. Функция вызывает себя много раз с разными аргументами для достижения многократного умножения.

Однако, у такой реализации есть проблема – в обычной среде JavaScript она раз в 10 медленнее, чем версия с циклом. Проход по циклу выходит дешевле, чем вызов функции.

Дилемма «скорость против элегантности» довольно интересна. Есть некий промежуток между удобством для человека и удобством для машины. Любую программу можно ускорить, сделав её больше и замысловатее. От программиста требуется находить подходящий баланс.

В случае с первым возведением в степень, неэлегантный цикл довольно прост и понятен. Не имеет смысла заменять его рекурсией. Часто, однако, программы работают с такими сложными концепциями, что хочется уменьшить эффективность путём повышения читаемости.

Основное правило, которое уже не раз повторяли, и с которым я полностью согласен – не беспокойтесь насчёт быстродействия, пока вы точно не уверены, что программа тормозит. Если так, найдите те части, которые работают дольше всех, и меняйте там элегантность на эффективность.

Конечно, мы не должны сразу же полностью игнорировать быстродействие. Во многих случаях, как с возведением в степень, особой простоты от элегантных решений мы не

получаем. Иногда опытный программист сразу видит, что простой подход никогда не будет достаточно быстрым.

Я заостряю на этом внимание оттого, что слишком много начинающих программистов хватаются за эффективность даже в мелочах. Результат получается больше, сложнее и часто не без ошибок. Такие программы дольше писать, а работают они часто не сильно быстрее.

Но рекурсия не всегда лишь менее эффективная альтернатива циклам. Некоторые задачи проще решить рекурсией. Чаще всего это обход нескольких веток дерева, каждая из которых может ветвиться.

Вот вам загадка: можно получить бесконечное количество чисел, начиная с числа 1, и потом либо добавляя 5, либо умножая на 3. Как нам написать функцию, которая, получив число, пытается найти последовательность таких сложений и умножений, которые приводят к заданному числу? К примеру, число 13 можно получить, сначала умножив 1 на 3, а затем добавив 5 два раза. А число 15 вообще нельзя так получить.

Рекурсивное решение:

```
function findSolution(target) {
  function find(start, history) {
    if (start == target)
      return history;
    else if (start > target)
      return null;
    else
      return find(start + 5, "(" + history + " + 5)") ||
             find(start * 3, "(" + history + " * 3)");
  }
  return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

Этот пример не обязательно находит самое короткое решение – он удовлетворяется любым. Не ожидаю, что вы сразу поймёте, как программа работает. Но давайте разбираться в этом отличном упражнении на рекурсивное мышление.

Внутренняя функция find занимается рекурсией. Она принимает два аргумента – текущее число и строку, которая содержит запись того, как мы пришли к этому номеру. И возвращает либо строчку, показывающую нашу последовательность шагов, либо null.

Для этого функция выполняет одно из трёх действий. Если заданное число равно цели, то текущая история как раз и является способом её достижения, поэтому она и возвращается. Если заданное число больше цели, продолжать умножения и сложения смысла нет, потому что так оно будет только увеличиваться. А если мы ещё не достигли цели, функция пробует оба возможных пути, начинающихся с заданного числа. Она дважды вызывает себя, один раз с каждым из способов. Если первый вызов возвращает не null, он возвращается. В другом случае возвращается второй.

Чтобы лучше понять, как функция достигает нужного эффекта, давайте просмотрим её вызовы, которые происходят в поисках решения для числа 13.

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        too big
      find(33, "(((1 + 5) + 5) * 3)")
        too big
    find(18, "((1 + 5) * 3)")
      too big
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        found!
```

Отступ показывает глубину стека вызовов. В первый раз функция find вызывает сама себя дважды, чтобы проверить решения, начинающиеся с (1 + 5) и (1 * 3). Первый вызов ищет решение, начинающееся с (1 + 5), и при помощи рекурсии проверяет все решения, выдающие число, меньшее или равное требуемому. Не находит, и возвращает null. Тогда-то оператор || и переходит к вызову функции, который исследует вариант (1 * 3). Здесь нас ждёт удача, потому что в третьем рекурсивном вызове мы получаем 13. Этот вызов возвращает строку, и каждый из операторов || по пути передаёт эту строку выше, в результате возвращая решение.

Выращиваем функции

Существует два более-менее естественных способа ввода функций в программу.

Первый – вы пишете схожий код несколько раз. Этого нужно избегать – больше кода означает больше места для ошибок и больше материала для чтения тех, кто пытается понять программу. Так что мы берём повторяющуюся функциональность, подбираем ей хорошее имя и помещаем её в функцию.

Второй способ – вы обнаруживаете потребность в некоей новой функциональности, которая достойна помещения в отдельную функцию. Вы начинаете с названия функции, и затем пишете её тело. Можно даже начинать с написания кода, использующего функцию, до того, как сама функция будет определена.

То, насколько сложно вам подобрать имя для функции, показывает, как хорошо вы представляете себе её функциональность. Возьмём пример. Нам нужно написать программу, выводящую два числа, количество коров и куриц на ферме, за которыми идут слова «коров» и «куриц». К числам нужно спереди добавить нули так, чтобы каждое занимало ровно три позиции.

```
007 Коров
011 Куриц
```

Очевидно, что нам понадобится функция с двумя аргументами. Начинаем кодить.

```
// вывестиИнвентаризациюФермы
function printFarmInventory(cows, chickens) {
  var cowString = String(cows);
  while (cowString.length < 3)
    cowString = "0" + cowString;
  console.log(cowString + " Коров");
  var chickenString = String(chickens);
  while (chickenString.length < 3)
    chickenString = "0" + chickenString;
  console.log(chickenString + " Куриц");
}
printFarmInventory(7, 11);
```

Если мы добавим к строке .length, мы получим её длину. Получается, что циклы while добавляют нули спереди к числам, пока не получат строчку в 3 символа.

Готово! Но только мы собрались отправить фермеру код (вместе с изрядным чеком, разумеется), он звонит и говорит нам, что у него в хозяйстве появились свиньи, и не могли бы мы добавить в программу вывод количества свиней?

Можно, конечно. Но когда мы начинаем копировать и вставлять код из этих четырёх строчек, мы понимаем, что надо остановиться и подумать. Должен быть способ лучше. Пытаемся улучшить программу:

```
// выводСДобавлениемНулейИМеткой
function printZeroPaddedWithLabel(number, label) {
  var numberString = String(number);
  while (numberString.length < 3)
    numberString = "0" + numberString;
  console.log(numberString + " " + label);
}

// вывестиИнвентаризацюФермы
function printFarmInventory(cows, chickens, pigs) {
  printZeroPaddedWithLabel(cows, "Коров");
  printZeroPaddedWithLabel(chickens, "Куриц");
  printZeroPaddedWithLabel(pigs, "Свиней");
}

printFarmInventory(7, 11, 3);
```

Работает! Но название printZeroPaddedWithLabel немного странное. Оно объединяет три вещи – вывод, добавление нулей и метку – в одну функцию. Вместо того, чтобы вставлять в функцию весь повторяющийся фрагмент, давайте выделим одну концепцию:

```
// добавитьНулей
function zeroPad(number, width) {
  var string = String(number);
  while (string.length < width)
    string = "0" + string;
  return string;
}

// вывестиИнвентаризацию Фермы
function printFarmInventory(cows, chickens, pigs) {
  console.log(zeroPad(cows, 3) + " Коров");
  console.log(zeroPad(chickens, 3) + " Куриц");
  console.log(zeroPad(pigs, 3) + " Свиней");
}

printFarmInventory(7, 16, 3);
```

Функция с хорошим, понятным именем zeroPad облегчает понимание кода. И её можно использовать во многих ситуациях, не только в нашем случае. К примеру, для вывода отформатированных таблиц с числами.

Насколько умными и универсальными должны быть функции? Мы можем написать как простейшую функцию, которая дополняет число нулями до трёх позиций, так и навороченную функцию общего назначения для форматирования номеров, поддерживающую дроби, отрицательные числа, выравнивание по точкам, дополнение разными символами, и т.п.

Хорошее правило – добавляйте только ту функциональность, которая вам точно пригодится. Иногда появляется искушение создавать фреймворки общего назначения для каждой небольшой потребности. Сопротивляйтесь ему. Вы никогда не закончите работу, а просто напишете кучу кода, который никто не будет использовать.

Функции и побочные эффекты

Функции можно грубо разделить на те, что вызываются из-за своих побочных эффектов, и те, что вызываются для получения некоторого значения. Конечно, возможно и объединение этих свойств в одной функции.

Первая вспомогательная функция в примере с фермой, printZeroPaddedWithLabel, вызывается из-за побочного эффекта: она выводит строку. Вторая, zeroPad, из-за возвращаемого значения. И это не совпадение, что вторая функция пригождается чаще первой. Функции, возвращающие значения, легче комбинировать друг с другом, чем функции, создающие побочные эффекты.

Чистая функция – особый вид функции, возвращающей значения, которая не только не имеет побочных эффектов, но и не зависит от побочных эффектов остального кода – к примеру, не работает с глобальными переменными, которые могут быть случайно изменены где-то ещё. Чистая функция, будучи вызванной с одними и теми же аргументами, возвращает один и тот же результат (и больше ничего не делает) – что довольно приятно. С ней просто работать. Вызов такой функции можно мысленно заменять результатом её работы, без изменения смысла кода. Когда вы хотите проверить такую функцию, вы можете просто вызвать её, и быть уверенным, что если она работает в данном контексте, она будет работать в любом. Не такие чистые функции могут возвращать разные результаты в зависимости от многих факторов, и иметь побочные эффекты, которые сложно проверять и учитывать.

Однако, не надо стесняться писать не совсем чистые функции, или начинать священную чистку кода от таких функций. Побочные эффекты часто полезны. Нет способа написать чистую версию функции console.log, и эта функция весьма полезна. Некоторые операции легче выразить, используя побочные эффекты.

Итог

Эта глава показала вам, как писать собственные функции. Когда ключевое слово function используется в виде выражения, возвращает указатель на вызов функции. Когда оно используется как инструкция, вы можете объявлять переменную, назначая ей вызов функции.

```
// Создаём f со ссылкой на функцию
var f = function(a) {
  console.log(a + 2);
};

// Объявляем функцию g
function g(a, b) {
  return a * b * 3.5;
}
```

Ключевой момент в понимании функций – локальные области видимости. Параметры и переменные, объявленные внутри функции, локальны для неё, пересоздаются каждый раз при её вызове, и не видны снаружи. Функции, объявленные внутри другой функции, имеют доступ к её области видимости.

Очень полезно разделять разные задачи, выполняемые программой, на функции. Вам не придётся повторяться, функции делают код более читаемым, разделяя его на смысловые части, так же, как главы и секции книги помогают в организации обычного текста.

Упражнения

Минимум

В предыдущей главе была упомянута функция Math.min, возвращающая самый маленький из аргументов. Теперь мы можем написать такую функцию сами. Напишите функцию min, принимающую два аргумента, и возвращающую минимальный из них.

```
console.log(min(0, 10));
// → 0
console.log(min(0, -10));
// → -10
```

Рекурсия

Мы видели, что оператор % (остаток от деления) может использоваться для определения того, чётное ли число (% 2). А вот ещё один способ определения:

- Ноль чётный.
- Единица нечётная.
- У любого числа *N* чётность такая же, как у *N* - 2.

Напишите рекурсивную функцию isEven согласно этим правилам. Она должна принимать число и возвращать булевское значение.

Потестируйте её на 50 и 75. Попробуйте задать ей -1. Почему она ведёт себя таким образом? Можно ли её как-то исправить?

```
console.log(isEven(50));
// → true
console.log(isEven(75));
// → false
console.log(isEven(-1));
// → ??
```

Считаем бобы

Символ номер N строки можно получить, добавив к ней .charAt(N) ("строчка".charAt(5)) – схожим образом с получением длины строки при помощи .length. Возвращаемое значение будет строковым, состоящим из одного символа (к примеру, "к"). У первого символа строки позиция 0, что означает, что у последнего символа позиция будет string.length - 1. Другими словами, у строки из двух символов длина 2, а позиции её символов будут 0 и 1.

Напишите функцию countBs, которая принимает строку в качестве аргумента, и возвращает количество символов “B”, содержащихся в строке.

Затем напишите функцию countChar, которая работает примерно как countBs, только принимает второй параметр — символ, который мы будем искать в строке (вместо того, чтобы просто считать количество символов “B”). Для этого переделайте функцию countBs.

Структуры данных: объекты и массивы

Два раза меня спрашивали: «Скажите, м-р Бэббидж, а если вы введёте в машину неправильные данные, получится ли правильный ответ?». Непостижима та путаница в головах, которая приводит к таким вопросам.

Чарльз Бэббидж, «Отрывки из жизни философа» (1864)

Числа, булевские значения и строки – кирпичики, из которых строятся структуры данных. Но нельзя сделать дом из одного кирпича. Объекты позволяют нам группировать значения (в том числе и другие объекты) вместе – и строить более сложные структуры.

Написание программ, которым мы до сего момента занимались, сильно затруднял тот факт, что они работали только с простыми данными. Эта глава добавит вам в инструментарий понимание структур данных. К её концу вы будете знать достаточно для того, чтобы начать писать полезные программы.

Глава пройдётся по более-менее реалистичному примеру программирования, вводя понятия по мере необходимости. Код примеров будет строиться из функций и переменных, которые мы определяли ранее.

Белка-оборотень

Иногда, обычно между восемью и десятью часами вечера, Жак против своей воли превращается в небольшого грызуна с пушистым хвостом.

С одной стороны, Жак рад, что он не превращается в классического волка. Превращение в белку влечёт меньше проблем. Вместо того, чтобы волноваться о том, не съешь ли ты соседа (это было бы неловко), он волнуется, как бы его не съел соседский кот. После того, как он дважды просыпался на очень тонкой ветке в кроне дуба, голый и дезориентированный, он приучился запирать окна и двери в своей комнате на ночь, и класть несколько орешков на пол, чтобы чем-то занять себя.

Так решаются проблемы с котом и дубом. Но Жак всё ещё страдает от своего заболевания. Нерегулярные обращения наводят его на мысль, что они должны быть чем-то вызваны. Сначала он думал, что это происходит только в те дни, когда он прикасался к деревьям. Он перестал это делать, и даже стал избегать подходить к ним. Но проблема не исчезла.

Перейдя к более научному подходу, Жак решил вести ежедневный дневник всего, чем он занимался, записывая туда, обращался ли он в белку. Так он надеется сузить круг вещей, приводящих к трансформации.

Сперва он решил разработать структуру данных для хранения этой информации.

Наборы данных

Для работы с куском данных нам вначале нужно найти способ представлять их в памяти машины. К примеру, нам нужно запомнить коллекцию чисел:

```
2, 3, 5, 7, 11
```

Можно поиграть со строками – строки могут быть любой длины, в них можно поместить много данных, и использовать для представления этого набора «2 3 5 7 11». Но это неудобно. Нам нужно будет как-то вынимать оттуда числа или вставлять новые в строку.

К счастью, JavaScript предлагает тип данных специально для хранения последовательностей чисел. Он называется массивом (array), и записывается, как список значений в квадратных скобках, разделённых запятыми:

```
var listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[1]);
// → 3
console.log(listOfNumbers[1 - 1]);
// → 2
```

Запись для получения элемента из массива тоже использует квадратные скобки. Пара скобок после выражения, содержащая внутри ещё одно выражение,

найдёт в массиве, который задан первым выражением, элемент, порядковый номер которого задан вторым выражением.

Номер первого элемента – ноль, а не один. Поэтому первый элемент можно получить так: listOfNumbers[0]. Если вы раньше не программировали, придётся привыкнуть к такой нумерации. Но она имеет давнюю традицию, и всё время, пока её последовательно соблюдают, она прекрасно работает.

Свойства

Мы видели много подозрительных выражений вроде myString.length (получение длины строки) и Math.max (получение максимума) в ранних примерах. Эти выражения используют свойства величин. В первом случае, мы получаем доступ к свойству length (длина) переменной myString. Во втором — доступ к свойству max объекта Math (который является набором функций и переменных, связанных с математикой).

Почти у всех переменных в JavaScript есть свойства. Исключения — null и undefined. Если вы попробуете получить доступ к несуществующим свойствам этих не-величин, получите ошибку:

```
null.length;
// → TypeError: Cannot read property 'length' of null
```

Два основных способа доступа к свойствам – точка и квадратные скобки. value.x и value[x] получают доступ к свойству value – но не обязательно к одному и тому же. Разница в том, как интерпретируется x. При использовании точки запись после точки должна быть именем существующей переменной, и она таким образом напрямую вызывает свойство по имени. При использовании квадратных скобок выражение в скобках вычисляется для получения имени свойства. value.x вызывает свойство под именем “x”, а value[x] вычисляет выражение x и использует результат в качестве имени свойства.

Если вы знаете, что интересующее вас свойство называется “length”, вы пишете value.length. Если вы хотите извлечь имя свойства из переменной i, вы пишете value[i]. А поскольку свойство может иметь любое имя, для доступа к свойству по имени “2” или “Jon Doe” вам придётся использовать квадратные скобки: value[2] или value["John Doe"]. Это необходимо даже когда вы знаете точное имя свойства, потому что “2” или «John Doe» не являются допустимыми именами переменных, поэтому к ним нельзя обратиться при помощи записи через точку.

Элементы массива хранятся в свойствах. Так как имена этих свойств – числа, и нам часто приходится получать их имена из значений переменных, нужно использовать квадратные скобки для доступа к ним. Свойство length массива говорит о том, сколько в нём элементов. Имя этого свойства – допустимое имя переменной, и мы его знаем заранее, поэтому обычно мы пишем array.length, потому, что это проще, чем писать array["length"].

Методы

Объекты string и array содержат, в дополнение к свойству length, несколько свойств, ссылающихся на функции.

```
var doh = "Дык";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → ДЫК
```

У каждой строки есть свойство toUpperCase. При вызове оно возвращает копию строки, в которой все буквы заменены на прописные. Есть также и toLowerCase – можете догадаться, что оно делает.

Что интересно, хотя вызов toUpperCase не передаёт никаких аргументов, функция каким-то образом получает доступ к строчке “Дык”, свойство которой мы вызывали.

Как это работает, описано в главе 6.

Свойства, содержащие функции, обычно называют методами той переменной, которой они принадлежат. То есть, toUpperCase – это метод строки.

В следующем примере демонстрируются некоторые методы, имеющиеся у массивов:

```
var mack = [];
mack.push("Трест,");
mack.push("который", "лопнул");
console.log(mack);
// → ["Трест,", "который", "лопнул"]
console.log(mack.join(" "));
// → Трест, который лопнул
console.log(mack.pop());
// → лопнул
console.log(mack);
// → ["Трест,", "который"]
```

Метод push используется для добавления значений в конец массива. pop делает обратное: удаляет значение из конца массива и возвращает его. Массив строк можно сплющить в одну строку при помощи метода join. В качестве аргумента join передают строку, которая будет вставлена между элементами массива.

Объекты

Вернёмся к нашей белке. Набор журнальных записей можно представить в виде массива. Но записи не состоят только лишь из номеров или строк – каждая должна хранить список того, что сделал наш герой, и булевское значение, показывающее, превратился ли Жак в белку. В идеале нам бы хотелось группировать каждую из записей в какую-то одну переменную, и потом добавлять их в массив.

Переменные типа object (объект) – коллекции произвольных свойств, и мы можем добавлять и удалять свойства объекта по желанию. Один из способов создать объект – использовать фигурные скобки:

```
var day1 = {
  squirrel: false,
  events: ["работа", "тронул дерево", "пицца", "пробежка"
, "телевизор"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

В скобках мы можем задать список свойств, разделённых запятыми. Записывается каждое свойство как имя, после которого идёт двоеточие, затем идёт выражение, которое и является значением свойства. Пробелы и переносы строк не учитываются. Разбивая запись свойств объекта на несколько строк, вы улучшаете читаемость кода. Если имя свойства не является допустимым именем переменной, его нужно заключать в кавычки:

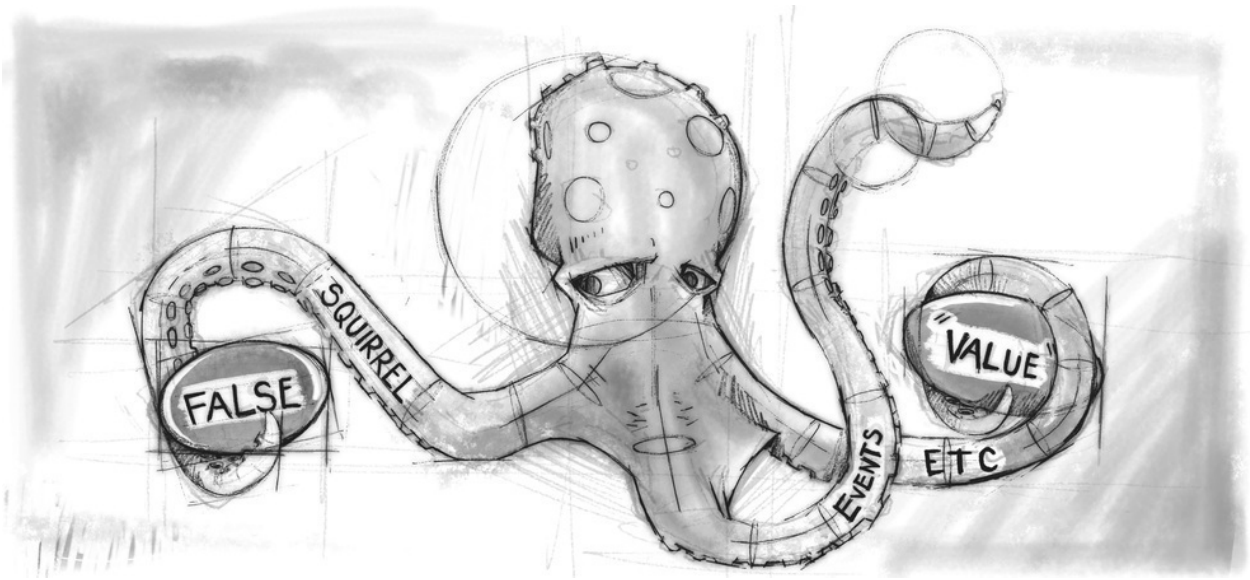
```
var descriptions = {
  work: "Пошёл на работу",
  "тронул дерево": "Дотронулся до дерева"
};
```

Получается, у фигурных скобок в JavaScript два значения. Употреблённые в начале инструкции, они начинают новый блок инструкций. В любом другом месте они описывают объект. Обычно нет смысла начинать инструкцию с описания объекта, и поэтому в программах обычно нет двусмысленностей по поводу этих двух применений фигурных скобок.

Если вы попытаетесь прочесть значение несуществующего свойства, вы получите undefined – как в примере, когда мы первый раз попробовали прочесть свойство wolf.

Свойству можно назначать значение через оператор =. Если у него ранее было значение, оно будет заменено. Если свойство отсутствовало, оно будет создано.

Возвращаясь к нашей модели со щупальцами и переменными, мы видим, что свойства тоже похожи на них. Они хватают значения, но на эти же значения могут ссылаться другие переменные и свойства. Объекты – это осьминоги с произвольным количеством щупалец, на каждом из которых написано имя свойства.

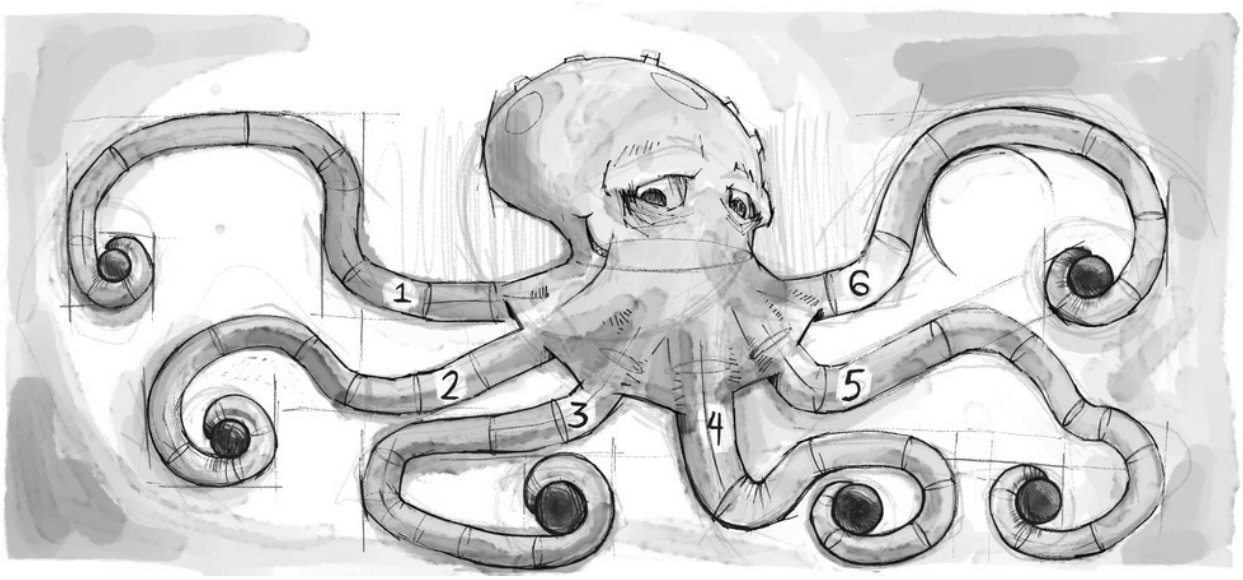


Оператор delete отрезает щупальце. Это унарный оператор, применяемый к выражению доступа к свойству. Это делается редко, но вполне возможно.

```
var anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

Бинарный оператор in принимает строку и имя объекта, и возвращает булевское значение, показывающее, есть ли у объекта свойство с таким именем. Есть разница между установкой значения свойства в undefined и удалением свойства. В первом случае свойство сохраняется у объекта, просто оно пустое. Во втором – свойства больше нет, и тогда in возвращает false.

Получается, что массивы – это разновидность объектов, которые специализируются на хранении последовательностей. Выражение typeof [1, 2] вернёт “object”. Их можно рассматривать как длинных плоских осьминогов, у которых все щупальца расположены ровным рядом и размечены номерами.



Поэтому журнал Жака можно представить в виде массива объектов:

```
var journal = [
  {events: ["работа", "тронул дерево", "пицца", "пробежка", "телевизор"],
   squirrel: false},
  {events: ["работа", "мороженое", "цветная капуста", "лазанья", "тронул дерево", "почистил зубы"],
   squirrel: false},
  {events: ["выходной", "велик", "перерыв", "арахис", "пивасик"],
   squirrel: true},
  /* и так далее... */
];
```

Изменчивость (Mutability)

Скоро мы уже и до программирования доберёмся. А пока нам нужно понять последнюю часть теории.

Мы увидели, что значения объекта можно менять. Типы значений, которые мы рассматривали ранее – числа, строки, булевские значения – неизменяемы. Нельзя поменять существующее значение заданного типа. Их можно комбинировать и выводить из них новые значения, но когда вы работаете с некоторым значением строки, это значение остаётся постоянным. Текст внутри строки нельзя поменять. Если у вас есть ссылка на строку "кошка", в коде нельзя поменять в ней символ, чтобы получилось "мошка".

А вот у объектов содержимое можно менять, изменяя значения их свойств.

Если у нас есть два числа, 120 и 120, мы можем рассматривать их как одно и то же, независимо от того, хранятся ли они в памяти в одном и том же месте. Но когда мы имеем дело с объектами, есть разница, есть ли у нас две ссылки на один объект или же у нас есть два разных объекта, содержащих одинаковые свойства. Рассмотрим пример:

```
var object1 = {value: 10};
var object2 = object1;
var object3 = {value: 10};

console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

Переменные object1 и object2 держатся за один и тот же объект, поэтому изменения object1 приводят к изменениям в object2. Переменная object3 показывает на другой объект, который изначально содержит те же свойства, что и object1, но живёт своей собственной жизнью.

Оператор == при сравнении объектов возвращает true только, если сравниваемые объекты – это одна и та же переменная. Сравнение разных объектов вернёт false, даже если у них идентичное содержимое. Оператора «глубокого» сравнения, который бы сравнивал содержимое объектов, в JavaScript не предусмотрено, но его возможно сделать самостоятельно (это будет одним из упражнений в конце главы).

Журнал оборотня

Итак, Жак запускает свой любимый интерпретатор JavaScript и создаёт окружение, необходимое для хранения журнала.

```
var journal = [];

function addEntry(events, didITurnIntoASquirrel) {
  journal.push({
    events: events,
    squirrel: didITurnIntoASquirrel
  });
}
```

Каждый вечер, часов в десять – а иногда и назавтра утром, спускаясь с верхней полки шкафа – он записывает свой день.

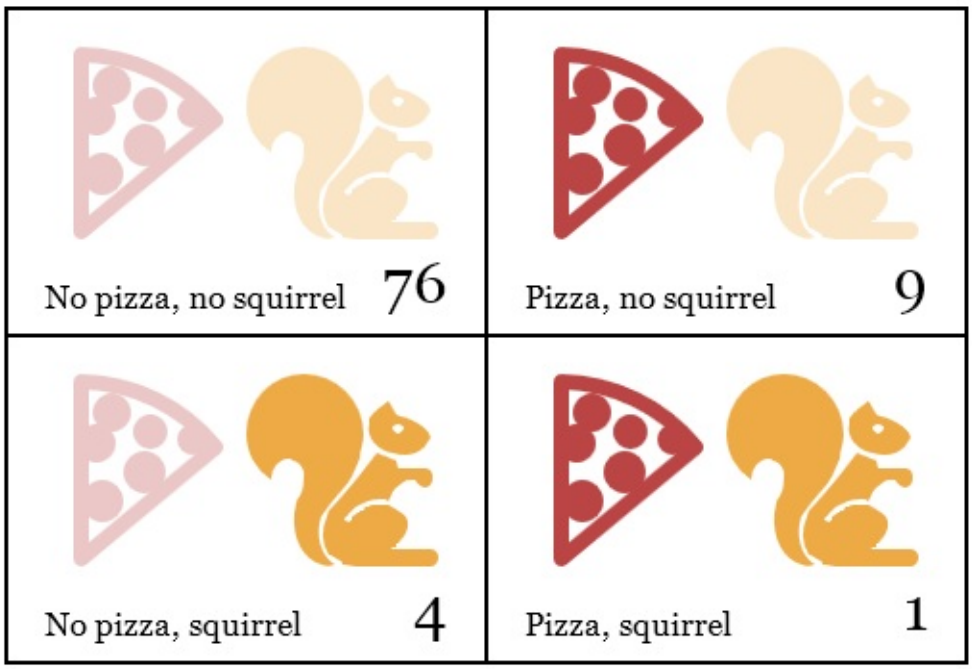
```
addEntry(["работа", "тронул дерево", "пицца", "пробежка"
, "телевизор"], false);
addEntry(["работа", "мороженое", "цветная капуста", "лаз
анья", "тронул дерево", "почистил зубы"], false);
addEntry(["выходной", "велик", "перерыв", "арахис", "пив
асик"], true);
```

Как только у него будет достаточно данных, он собирается вычислить корреляцию между его оборачиваниями и событиями каждого из дней, и в

идеале узнать из их корреляций что-то полезное.

Корреляция – это мера зависимости между переменными величинами (переменными в статистическом смысле, а не в смысле JavaScript). Она обычно выражается в виде коэффициента, принимающего значения от -1 до 1. Нулевая корреляция обозначает, что переменные вообще не связаны, а корреляция 1 означает, что они полностью связаны – если вы знаете одну, вы автоматически знаете другую. Минус один также означает прочную связь переменных, но и их противоположность – когда одна true, вторая всегда false.

Для измерения корреляции булевских переменных хорошо подходит коэффициент фи (ϕ), к тому же, его сравнительно легко подсчитать. Для этого нам нужна таблица, содержащая количество раз, когда наблюдались различные комбинации двух переменных. К примеру, мы можем взять события «поел пиццы» и «обращение» и представить их в следующей таблице:



ϕ можно вычислить по следующей формуле, где n относится к ячейкам таблицы:

$$\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}}$$

n01 обозначает количество измерений, когда первое событие (пицца) false (0), а второе событие (обращение) true (1). В нашем примере n01 = 4.

Запись n1• обозначает сумму всех измерений, где первое событие было true, что для нашего примера равно 10. Соответственно, n•0 – сумма всех измерений, где событие «обращение» было false.

Значит, для таблицы с пиццей числитель формулы будет 1×76 - 9×4 = 40, а знаменатель – корень из 10×80×5×85, или √340000. Получается, что ϕ ≈ 0.069, что довольно

мало. Непохоже, чтобы пицца влияла на обращения в белку.

Вычисляем корреляцию

Таблицу 2x2 можно представить массивом из четырёх элементов ([76, 9, 4, 1]), массивом из двух элементов, каждый из которых является также двухэлементным массивом ([[76, 9], [4, 1]]), или же объектом со свойствами под именами "11" или "01". Но для нас одномерный массив проще, и выражение для доступа к нему будет короче. Мы будем обрабатывать индексы массива как двузначные двоичные числа, где левый знак обозначает переменную оборачиваемости, а правый – события. К примеру, 10 обозначает случай, когда Жак обратился в белку, но событие (к примеру, «пицца») не имело места. Так случилось 4 раза. И поскольку двоичное 10 – это десятичное 2, мы будем хранить это в массиве по индексу 2.

Функция, вычисляющая коэффициент ϕ из такого массива:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

Это просто прямая реализация формулы ϕ на языке JavaScript. Math.sqrt – это функция извлечения квадратного корня объекта Math из стандартного окружения JavaScript. Нам нужно сложить два поля таблицы для получения полей типа n1•, потому что мы не храним в явном виде суммы столбцов или строк.

Жак вёл журнал три месяца. Результат доступен на сайте книги eloquentjavascript.net/code/jacques_journal.js

Чтобы извлечь переменную 2x2 для конкретного события, нам нужно в цикле пройтись по всем записям и посчитать, сколько раз оно случается по отношению к обращению в белку.

```
function hasEvent(event, entry) {
  return entry.events.indexOf(event) != -1;
}

function tableFor(event, journal) {
  var table = [0, 0, 0, 0];
  for (var i = 0; i < journal.length; i++) {
    var entry = journal[i], index = 0;
    if (hasEvent(event, entry)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
}

console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

Функция hasEvent проверяет, содержит ли запись нужный элемент. У массивов есть метод indexOf, который ищет заданное значение (в нашем случае – имя события) в массиве и возвращает индекс его положения в массиве (-1, если его в массиве нет). Значит, если вызов indexOf не вернул -1, то событие в записи есть.

Тело цикла в tableFor рассчитывает, в какую ячейку таблицы попадает каждая из журнальных записей. Она смотрит, содержит ли запись нужное событие, и связано

ли оно с обращением в белку. Затем цикл увеличивает на единицу элемент массива, соответствующий нужной ячейке.

Теперь у нас есть все инструменты для подсчёта корреляций. Осталось только подсчитать корреляции для каждого из событий, и посмотреть, не выдаётся ли что из списка. Но как хранить эти корреляции?

Объекты как карты (map)

Один из способов – хранить корреляции в массиве, используя объекты со свойствами name и value. Однако поиск корреляций в массиве будет довольно громоздким: нужно будет пройтись по всему массиву, чтобы найти объект с нужным именем. Можно было бы обернуть этот процесс в функцию, но код пришлось бы писать всё равно, и компьютер выполнял бы больше работы, чем необходимо.

Способ лучше – использовать свойства объектов с именами событий. Мы можем использовать квадратные скобки для создания и чтения свойств и оператор in для проверки существования свойства.

```
var map = {};
function storePhi(event, phi) {
  map[event] = phi;
}

storePhi("пицца", 0.069);
storePhi("тронул дерево", -0.081);
console.log("пицца" in map);
// → true
console.log(map["тронул дерево"]);
// → -0.081
```

Карта (map) – способ связать значения из одной области (в данном случае – названия событий) со значениями в другой (в нашем случае – коэффициенты ϕ).

С таким использованием объектов есть пара проблем – мы обсудим их в главе 6, но пока волноваться не будем.

Что, если нам надо собрать все события, для которых сохранены коэффициенты? Они не создают предсказуемую последовательность, как было бы в массиве, поэтому цикл for использовать не получится. JavaScript предлагает конструкцию цикла специально для обхода всех свойств объекта. Она похожа на цикл for, но использует команду in.

```
for (var event in map)
  console.log("Кореляция для '" + event +
              "' получается " + map[event]);
// → Кореляция для 'пицца' получается 0.069
// → Кореляция для 'тронул дерево' получается -0.081
```

Итоговый анализ

Чтобы найти все типы событий, представленных в наборе данных, мы обрабатываем каждое вхождение по очереди, и затем создаём цикл по всем событиям вхождения. Мы храним объект phis, в котором содержатся корреляционные коэффициенты для всех типов событий, которые мы уже нашли. Если мы встречаем новый тип, которого ещё не было в phis, мы подсчитываем его корреляцию и добавляем её в объект.

```
function gatherCorrelations(journal) {
  var phis = {};
  for (var entry = 0; entry < journal.length; entry++) {
    var events = journal[entry].events;
    for (var i = 0; i < events.length; i++) {
      var event = events[i];
      if (!(event in phis))
        phis[event] = phi(tableFor(event, journal));
    }
  }
  return phis;
}

var correlations = gatherCorrelations(JOURNAL);
console.log(correlations.пицца);
// → 0.068599434
```

Смотрим, что получилось:

```
for (var event in correlations)
  console.log(event + ": " + correlations[event]);
// → морковка:   0.0140970969
// → упражнения: 0.0685994341
// → выходной:   0.1371988681
// → хлеб:      -0.0757554019
// → пудинг:    -0.0648203724
// и так далее...
```

Большинство корреляций лежат близко к нулю. Морковки, хлеб и пудинг, очевидно, не связаны с обращением в белку. Но оно вроде бы более часто происходит на

выходных. Давайте отфильтруем результаты, чтобы выводить только корреляции больше 0.1 или меньше -0.1

```
for (var event in correlations) {
  var correlation = correlations[event];
  if (correlation > 0.1 || correlation < -0.1)
    console.log(event + ": " + correlation);
}
// → выходной:      0.1371988681
// → чистил зубы: -0.3805211953
// → конфета:       0.1296407447
// → работа:       -0.1371988681
// → спагетти:      0.2425356250
// → читал:         0.1106828054
// → арахис:        0.5902679812
```

Ага! У двух факторов корреляции заметно больше остальных. Арахис сильно влияет на вероятность превращения в белку, тогда как чистка зубов наоборот, препятствует этому.

Интересно. Попробуем вот что:

```
for (var i = 0; i < JOURNAL.length; i++) {
  var entry = JOURNAL[i];
  if (hasEvent("арахис", entry) &&
      !hasEvent("чистка зубов", entry))
    entry.events.push("арахис зубы");
}
console.log(phi(tableFor("арахис зубы", JOURNAL)));
// → 1
```

Ошибки быть не может! Феномен случается именно тогда, когда Жак ест арахис и не чистит зубы. Если б он только не был таким неряхой относительно оральной гигиены, он бы вообще не заметил своего несчастья.

Зная это, Жак просто перестаёт есть арахис и обнаруживает, что трансформации прекратились.

У Жака какое-то время всё хорошо. Но через несколько лет он теряет работу, и в конце концов ему приходится наняться в цирк, где он выступает как Удивительный Человек-белка, набирая полный рот арахисового масла перед шоу. Однажды, устав от столь жалкого существования, Жак не обращается обратно в человека, пробирается через дыру в цирковом тенте и исчезает в лесу. Больше его никто не видел.

Дальнейшая массивология

В конце главы хочу познакомить вас ещё с несколькими концепциями, относящимися к объектам. Начнём с полезных методов, имеющихся у массивов.

Мы видели методы push и pop, которые добавляют и отнимают элементы в конце массива. Соответствующие методы для начала массива называются unshift и shift

```
var todoList = [];
function rememberTo(task) {
  todoList.push(task);
}
function whatIsNext() {
  return todoList.shift();
}
function urgentlyRememberTo(task) {
  todoList.unshift(task);
}
```

Данная программа управляет списком дел. Вы добавляете дела в конец списка, вызывая rememberTo("поесть"), а когда вы готовы заняться чем-то, вызываете whatIsNext(), чтобы получить (и удалить) первый элемент списка. Функция urgentlyRememberTo тоже добавляет задачу, но только в начало списка.

У метода indexOf есть родственник по имени lastIndexOf, который начинает поиск элемента в массиве с конца:

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

Оба метода, indexOf и lastIndexOf, принимают необязательный второй аргумент, который задаёт начальную позицию поиска.

Ещё один важный метод – slice, который принимает номера начального (start) и конечного (end) элементов, и возвращает массив, состоящий только из элементов, попадающих в этот промежуток. Включая тот, что находится по индексу start, но исключая тот, что по индексу end.

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

Когда индекс end не задан, slice выбирает все элементы после индекса start. У строк есть схожий метод, который работает так же.

Метод concat используется для склейки массивов, примерно как оператор + склеивает строки. В примере показаны методы concat и slice в деле. Функция принимает массив array и индекс index, и возвращает новый массив, который является копией предыдущего, за исключением удалённого элемента, находившегося по индексу index.

```
function remove(array, index) {
  return array.slice(0, index).concat(array.slice(index
+ 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

Строки и их свойства

Мы можем получать значения свойств строк, например length и toUpperCase. Но попытка добавить новое свойство ни к чему не приведёт:

```
var myString = "Шарик";
myString.myProperty = "значение";
console.log(myString.myProperty);
// → undefined
```

Величины типа строка, число и булевские – не объекты, и хотя язык не жалуется на попытки назначить им новые свойства, он на самом деле их не сохраняет. Величины неизменяемы.

Но у них есть свои встроенные свойства. У каждой строки есть набор методов. Самые полезные, пожалуй – slice и indexOf, напоминающие те же методы у массивов.

```
console.log("кокосы".slice(3, 6));
// → осы
console.log("кокос".indexOf("с"));
// → 4
```

Разница в том, что у строки метод indexOf может принять строку, содержащую больше одного символа, а у массивов такой метод работает только с одним элементом.

```
console.log("раз два три".indexOf("ва"));
// → 5
```

Метод trim удаляет пробелы (а также переводы строк, табуляцию и прочие подобные символы) с обоих концов строки.

```
console.log("  ладно  \n ".trim());
// → ладно
```

Мы уже сталкивались со свойством строки length. Доступ к отдельным символам строчки можно получить через метод charAt, а также просто через нумерацию позиций, как в массиве:

```
var string = "abc";
console.log(string.length);
// → 3
console.log(string.charAt(0));
// → a
console.log(string[1]);
// → b
```

Объект arguments

Когда вызывается функция, к окружению исполняемого тела функции добавляется особая переменная под названием arguments. Она указывает на объект, содержащий все аргументы, переданные функции. Помните, что в JavaScript вы можете передавать функции больше или меньше аргументов, чем объявлено при помощи параметров.

```
function noArguments() {}
noArguments(1, 2, 3); // Пойдёт
function threeArguments(a, b, c) {}
threeArguments(); // И так можно
```

У объекта arguments есть свойство length, которое содержит реальное количество переданных функции аргументов. Также у него есть свойства для каждого аргумента под именами 0, 1, 2 и т.д.

Если вам кажется, что это очень похоже на массив – вы правы. Это очень похоже на массив. К сожалению, у этого объекта нет методов типа slice или indexOf, что делает доступ к нему труднее.

```
function argumentCounter() {
  console.log("Ты дал мне", arguments.length, "аргумента
.");
}
argumentCounter("Дядя", "Стёпа", "Милиционер");
// → Ты дал мне 3 аргумента.
```

Некоторые функции рассчитаны на любое количество аргументов, как console.log. Они обычно проходят циклом по свойствам объекта arguments. Это можно использовать для создания удобных интерфейсов. К примеру, вспомните, как мы создавали записи для журнала Жака:

```
addEntry(["работа", "тронул дерево", "пицца", "пробежка"
, "телевизор"], false);
```

Так как мы часто вызываем эту функцию, мы можем сделать альтернативу, которую проще вызывать:

```
function addEntry(squirrel) {
  var entry = {events: [], squirrel: squirrel};
  for (var i = 1; i < arguments.length; i++)
    entry.events.push(arguments[i]);
  journal.push(entry);
}
addEntry(true, "работа", "тронул дерево", "пицца", "проб
ежка", "телевизор");
```

Эта версия читает первый аргумент как обычно, а по остальным проходит в цикле (начиная с индекса 1, пропуская первый аргумент) и собирает их в массив.

Объект Math

Мы уже видели, что Math – набор инструментов для работы с числами, такими, как Math.max (максимум), Math.min (минимум), и Math.sqrt (квадратный корень).

Объект Math используется просто как контейнер для группировки связанных функций. Есть только один объект Math, и он почти не используется в виде значений. Он просто предоставляет пространство имён для всех этих функций и значений, чтоб не нужно было делать их глобальными.

Слишком большое число глобальных переменных «загрязняет» пространство имён. Чем больше имён занято, тем больше вероятность случайно использовать одно из них в качестве переменной. К примеру, весьма вероятно, что вы захотите использовать имя max для чего-то в своей программе. Поскольку встроенная в JavaScript функция max безопасно упакована в объект Math, нам не нужно волноваться по поводу того, что мы её перезапишем.

Многие языки остановят вас, или хотя бы предупредят, когда вы будете определять переменную с именем, которое уже занято. JavaScript не будет этого делать, поэтому будьте осторожны.

Возвращаясь к объекту Math, если вам нужна тригонометрия, он вам поможет. У него есть cos (косинус), sin (синус), и tan (тангенс), их обратные функции — acos, asin, и atan. Число π (pi) – или, по крайней мере, его близкая аппроксимация, помещающаяся в число JavaScript – также доступно как Math.PI. (Есть такая старая традиция в программировании — записывать имена констант в верхнем регистре.)

```
function randomPointOnCircle(radius) {
  var angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

Если вы незнакомы с синусами и косинусами – не отчаивайтесь. Мы их будем использовать в 13 главе, и тогда я их объясню.

В предыдущем примере используется Math.random. Это функция, возвращающая при каждом вызове новое псевдослучайное число между нулём и единицей (включая ноль).

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

Хотя компьютеры – машины детерминированные (они всегда реагируют одинаково на одни и те же входные данные), возможно заставить их выдавать кажущиеся случайными числа. Для этого машина хранит у себя во внутреннем состоянии несколько чисел. Каждый раз,

когда идёт запрос на случайное число, она выполняет разные сложные детерминированные вычисления и возвращает часть результата вычислений. Этот результат она использует для того, чтобы изменить своё внутреннее состояние, поэтому следующее «случайное» число получается другим.

Если вам нужно целое случайное число, а не дробь, вы можете использовать Math.floor (округляет число вниз до ближайшего целого) на результате Math.random.

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

Умножая случайное число на 10, получаем число от нуля до 10 (включая ноль). Так как Math.floor округляет вниз, мы получим число от 0 до 9 включительно.

Есть также функция Math.ceil (от «ceiling» – потолок, округляет вверх до ближайшего целого) и Math.round (округляет до ближайшего целого).

Объект global

К глобальной области видимости, где живут глобальные переменные, можно также получить доступ как к объекту. Каждая глобальная переменная является свойством

этого объекта. В браузерах глобальная область видимости хранится в переменной window.

```
var myVar = 10;
console.log("myVar" in window);
// → true
console.log(window.myVar);
// → 10
```

Итог

Объекты и массивы (которые представляют из себя подвид объектов) позволяют сгруппировать несколько величин в одну. В принципе, это позволяет нам засунуть несколько связанных между собой вещей в мешок и бегать с ним кругами, вместо того, чтобы пытаться сгребать все эти вещи руками и пытаться держать их каждую по отдельности.

У большинства величин в JavaScript есть свойства, исключение составляют null и undefined. Мы получаем доступ к ним через value.propName или value["propName"]. Объекты используют имена для хранения свойств и хранят более-менее фиксированное их количество. Массивы обычно содержат переменное количество сходных по типу величин и используют числа (начиная с нуля) в качестве имён этих величин.

Также в массивах есть именованные свойства, такие как length, и несколько методов. Методы – это функции, живущие среди свойств и (обычно) работающие над той величиной, чьим свойством они являются.

Объекты также могут работать как карты, ассоциируя значения с именами. Оператор in используется для выяснения того, содержит ли объект свойство с данным именем. Это же ключевое слово используется в цикле for (for (var name in object)) для перебора всех свойств объекта.

Упражнения

Сумма диапазона

Во введении был упомянут удобный способ подсчёта сумм диапазонов чисел:

```
console.log(sum(range(1, 10)));
```

Напишите функцию range, принимающую два аргумента – начало и конец диапазона – и возвращающую массив, который содержит все числа из него, включая начальное и конечное.

Затем напишите функцию sum, принимающую массив чисел и возвращающую их сумму. Запустите указанную выше инструкцию и убедитесь, что она возвращает 55.

В качестве бонуса дополните функцию range, чтобы она могла принимать необязательный третий аргумент – шаг для построения массива. Если он не задан, шаг равен единице. Вызов функции range(1, 10, 2) должен будет вернуть [1, 3, 5, 7, 9]. Убедитесь, что она работает с отрицательным шагом так, что вызов range(5, 2, -1) возвращает [5, 4, 3, 2].

```
console.log(sum(range(1, 10)));
// → 55
console.log(range(5, 2, -1));
// → [5, 4, 3, 2]
```

Обращаем вспять массив

У массивов есть метод reverse, меняющий порядок элементов в массиве на обратный. В качестве упражнения напишите две функции, reverseArray и reverseArrayInPlace. Первая получает массив как аргумент и выдаёт новый массив – с обратным порядком элементов. Вторая работает как оригинальный метод

reverse – она меняет порядок элементов на обратный в том массиве, который был ей передан в качестве аргумента. Не используйте стандартный метод reverse.

Если иметь в виду побочные эффекты и чистые функции из предыдущей главы, какой из вариантов вам кажется более полезным? Какой более эффективным?

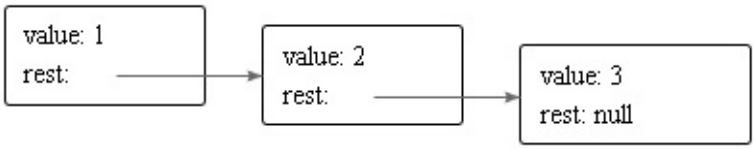
```
console.log(reverseArray(["A", "B", "C"]));
// → ["C", "B", "A"];
var arrayValue = [1, 2, 3, 4, 5];
reverseArrayInPlace(arrayValue);
console.log(arrayValue);
// → [5, 4, 3, 2, 1]
```

Список

Объекты могут быть использованы для построения различных структур данных. Часто встречающаяся структура – список (не путайте с массивом). Список – связанный набор объектов, где первый объект содержит ссылку на второй, второй – на третий, и т.п.

```
var list = {
  value: 1,
  rest: {
    value: 2,
    rest: {
      value: 3,
      rest: null
    }
  }
};
```

В результате объекты формируют цепочку:



Списки удобны тем, что они могут делиться частью своей структуры. Например, можно сделать два списка, {value: 0, rest: list} и {value: -1, rest: list}, где list – это ссылка на ранее объявленную переменную. Это два независимых списка, при этом у них есть общая структура list, которая включает три последних элемента каждого из них. Кроме того, оригинальный список также сохраняет свои свойства как отдельный список из трёх элементов.

Напишите функцию arrayToList, которая строит такую структуру, получая в качестве аргумента [1, 2, 3], а также функцию listToArray, которая создаёт массив из списка. Также напишите вспомогательную функцию prepend,

которая получает элемент и создаёт новый список, где этот элемент добавлен спереди к первоначальному списку, и функцию nth, которая в качестве аргументов принимает список и число, а возвращает элемент на заданной позиции в списке или же undefined в случае отсутствия такого элемента.

Если ваша версия nth не рекурсивна, тогда напишите её рекурсивную версию.

```
console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

Глубокое сравнение

Оператор == сравнивает переменные объектов, проверяя, ссылаются ли они на один объект. Но иногда полезно было бы сравнить объекты по содержимому.

Напишите функцию deepEqual, которая принимает два значения и возвращает true, только если это два одинаковых значения или это объекты, свойства которых

имеют одинаковые значения, если их сравнивать рекурсивным вызовом deepEqual.

Чтобы узнать, когда сравнивать величины через ===, а когда – объекты по содержимому, используйте оператор typeof. Если он выдаёт "object" для обеих величин, значит нужно делать глубокое сравнение. Примите во внимание одно дурацкое исключение, существующее по историческим причинам: typeof null тоже возвращает "object".

```
var obj = {here: {is: "an"}, object: 2};
console.log(deepEqual(obj, obj));
// → true
console.log(deepEqual(obj, {here: 1, object: 2}));
// → false
console.log(deepEqual(obj, {here: {is: "an"}, object: 2}
));
// → true
```

Функции высшего порядка

Цу-ли и Цу-су похвалялись размерами своих новых программ. «Двести тысяч строк»,- сказал Цу-ли,- «не считая комментариев!» Цу-су ответил: «Пф-ф, моя – почти миллион строк». Мастер Юнь-Ма сказал: «Моя лучшая программа занимает пятьсот строк». Услышав это, Цу-ли и Цу-су испытали просветление.

Мастер Юнь-Ма, Книга программирования

Есть два способа построения программ: сделать их настолько простыми, что там очевидно не будет ошибок, или же настолько сложными, что там не будет очевидных ошибок.

Энтони Хоар, 1980 лекция на вручении премии Тьюринга

Большая программа – затратная программа, и не только из-за времени её написания. Большой размер обычно означает сложность, а сложность сбивает с толку программистов. Сбитые с толку программисты делают ошибки в программах. Большая программа означает, что багам есть где спрятаться, и их получается труднее отыскать.

Вернёмся ненадолго к двум примерам из введения. Первый самодостаточен и занимает шесть строк.

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

Второй основан на двух внешних функциях и занимает одну строку.

```
console.log(sum(range(1, 10)));
```

В каком из них скорее встретится ошибка?

Если мы добавим размер определений sum и range, вторая программа тоже получится большой – больше первой. Но я всё равно утверждаю, что она скорее всего будет правильной.

Это будет потому, что выражение решения непосредственно относится к решаемой задаче. Суммирование числового промежутка – это не циклы и счётчики. Это суммы и промежутки.

Определения этого словаря (функции sum и range) будут включать циклы, счётчики и другие случайные детали. Но потому, что они выражают более простые концепции, чем вся программа, их проще сделать правильно.

Абстракции

В программном контексте эти «словарные» определения часто называют абстракциями. Абстракции прячут детали и дают нам возможность разговаривать о задачах на высшем, или более абстрактном, уровне.

Сравните два рецепта горохового супа:

Добавьте в ёмкость по одной чашке сухого гороха на порцию. Добавьте воды так, чтобы она покрыла горох. Оставьте его так минимум на 12 часов. Выньте горох из воды и поместите его в кастрюлю. Добавьте 4 чашки воды на порцию. Закройте кастрюлю и тушите горох два часа. Возьмите по половине луковицы на порцию. Порежьте на куски ножом, добавьте к гороху. Возьмите по одному стеблю сельдерея на порцию. Порежьте на куски ножом, добавьте к гороху. Возьмите по морковке на порцию. Порежьте на куски ножом, добавьте к гороху. Готовьте ещё 10 минут.

Второй рецепт:

На порцию: 1 чашка сухого гороха, половина луковицы, стебель сельдерея, морковка. Вымачивайте горох 12 часов. Тушите 2 часа в 4 чашках воды на порцию. Порежьте и добавьте овощи. Готовьте ещё 10 минут.

Второй – короче и проще. Но вам нужно знать чуть больше понятий, связанных с готовкой – вымачивание, тушение, резка (и овощи).

Программируя, мы не можем рассчитывать на то, что все необходимые слова будут в нашем словаре. Из-за этого вы можете скатиться до шаблона первого рецепта: диктовать компьютеру все мелкие шажки друг за другом, не замечая понятий более высокого уровня, которые они выражают.

Второй натурой программиста должно стать умение замечать, когда понятие умоляет придумать для него новое слово и вынести в абстракцию.

Абстрагируем обход массива

Простые функции, которые мы использовали раньше, хороши для построения абстракций. Но иногда их бывает недостаточно.

В предыдущей главе мы несколько раз встречали такой цикл:

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
  var current = array[i];
  console.log(current);
}
```

Код пытается сказать: «для каждого элемента в массиве – вывести его в консоль». Но он использует обходной путь – с переменной для подсчёта i, проверкой длины массива, и объявлением дополнительной переменной current. Мало того, что он не очень красив, он ещё и является почвой для потенциальных ошибок. Мы можем случайно повторно использовать переменную i, вместо length написать lenght, перепутать переменные i и current, и т.п.

Давайте абстрагируем его в функцию. Можете придумать способ сделать это?

Довольно просто написать функцию, обходящую массив и вызывающую для каждого элемента console.log

```
function logEach(array) {
  for (var i = 0; i < array.length; i++)
    console.log(array[i]);
}
```

Но что, если нам надо делать что-то другое, нежели выводить элементы в консоль? Поскольку «делать что-то» можно представить как функцию, а функции – это просто переменные, мы можем передать это действие как аргумент:

```
function forEach(array, action) {
  for (var i = 0; i < array.length; i++)
    action(array[i]);
}

forEach(["Тили", "Мили", "Трямдия"], console.log);
// → Тили
// → Мили
// → Трямдия
```

Часто можно не передавать заранее определённую функцию в forEach, а создавать функцию прямо на месте.

```
var numbers = [1, 2, 3, 4, 5], sum = 0;
forEach(numbers, function(number) {
  sum += number;
});
console.log(sum);
// → 15
```

Выглядит похоже на классический цикл for, с телом цикла, записанным в блоке. Однако, теперь тело находится внутри функции, и также внутри скобок вызова forEach.

Поэтому его нужно закрыть как фигурной, так и круглой скобкой.

Используя этот шаблон, мы можем задать имя переменной для текущего элемента массива (number), без необходимости выбирать его из массива вручную.

Вообще, нам даже не нужно писать самим forEach. Это стандартный метод массивов. Так как массив уже передан в качестве переменной, над которой мы работаем, forEach принимает только один аргумент – функцию, которую нужно выполнить для каждого элемента.

Для демонстрации удобства этого подхода вернёмся к функции из предыдущей главы. Она содержит два цикла, проходящих по массивам:

```
function gatherCorrelations(journal) {
  var phis = {};
  for (var entry = 0; entry < journal.length; entry++) {
    var events = journal[entry].events;
    for (var i = 0; i < events.length; i++) {
      var event = events[i];
      if (!(event in phis))
        phis[event] = phi(tableFor(event, journal));
    }
  }
  return phis;
}
```

Используя forEach мы делаем запись чуть короче и гораздо чище.

```
function gatherCorrelations(journal) {
  var phis = {};
  journal.forEach(function(entry) {
    entry.events.forEach(function(event) {
      if (!(event in phis))
        phis[event] = phi(tableFor(event, journal));
    });
  });
  return phis;
}
```

Функции высшего порядка

Функции, оперирующие другими функциями – либо принимая их в качестве аргументов, либо возвращая их, называются функциями высшего порядка. Если вы уже поняли, что функции – это всего лишь переменные, ничего особенного в существовании таких функций нет. Термин происходит из математики, где различия между функциями и другими значениями воспринимаются более строго.

Функции высшего порядка позволяют нам абстрагировать действия, а не только значения. Они бывают разными. Например, можно сделать функцию, создающую новые

функции.

```
function greaterThan(n) {
  return function(m) { return m > n; };
}
var greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

Можно сделать функцию, меняющую другие функции.

```
function noisy(f) {
  return function(arg) {
    console.log("calling with", arg);
    var val = f(arg);
    console.log("called with", arg, "- got", val);
    return val;
  };
}
noisy(Boolean)(0);
// → calling with 0
// → called with 0 - got false
```

Можно даже делать функции, создающие новые типы управления потоком выполнения программы.

```
function unless(test, then) {
  if (!test) then();
}
function repeat(times, body) {
  for (var i = 0; i < times; i++) body(i);
}

repeat(3, function(n) {
  unless(n % 2, function() {
    console.log(n, "is even");
  });
});
// → 0 is even
// → 2 is even
```

Правила лексических областей видимости, которые мы обсуждали в главе 3, работают нам на пользу в таких случаях. В последнем примере переменная n – это аргумент внешней функции. Поскольку внутренняя функция живёт в окружении внешней, она может использовать n. Тела таких внутренних функций имеют доступ к переменным, окружающим их. Они могут играть роль блоков {}, используемых в обычных циклах и условных выражениях. Важное отличие в том, что переменные, объявленные внутри внутренних функций, не попадают в окружение внешней. И обычно это только к лучшему.

Передача аргументов

Функция noisy, объявленная ранее, которая передаёт свой аргумент в другую функцию, не совсем удобна.

```
function noisy(f) {
  return function(arg) {
    console.log("calling with", arg);
    var val = f(arg);
    console.log("called with", arg, "- got", val);
    return val;
  };
}
```

Если f принимает больше одного параметра, она получит только первый. Можно было бы добавить кучу аргументов к внутренней функции (arg1, arg2 и т.д.) и передать все их в f, но ведь неизвестно, какого количества нам хватит. Кроме того, функция f не могла бы корректно работать с arguments.length. Так как мы всё время передавали бы одинаковое число аргументов, было бы неизвестно, сколько аргументов нам было задано изначально.

Для таких случаев у функций в JavaScript есть метод apply. Ему передают массив (или объект в виде массива) из аргументов, а он вызывает функцию с этими аргументами.

```
function transparentWrapping(f) {
  return function() {
    return f.apply(null, arguments);
  };
}
```

Данная функция бесполезна, но она демонстрирует интересующий нас шаблон – возвращаемая ею функция передаёт в f все полученные ею аргументы, но не более того. Происходит это при помощи передачи её собственных аргументов, хранящихся в объекте arguments, в метод apply. Первый аргумент метода apply, которому мы в данном случае присваиваем null, можно использовать для эмуляции вызова метода. Мы вернёмся к этому вопросу в следующей главе.

JSON

Функции высшего порядка, которые каким-то образом применяют функцию к элементам массива, широко распространены в JavaScript. Метод forEach – одна из самых примитивных подобных функций. В качестве методов массивов нам доступно много других вариантов функций. Для знакомства с ними давайте поиграем с ещё одним набором данных.

Несколько лет назад кто-то обследовал много архивов и сделал целую книгу по истории моей фамилии. Я открыл её, надеясь найти там рыцарей, пиратов и алхимиков… Но оказалось, что она заполнена в основном фламандскими фермерами. Для развлечения я извлёк информацию по моим непосредственным предкам и перевёл в формат, пригодный для чтения компьютером.

Файл выглядит примерно так:

```
[
  {"name": "Emma de Milliano", "sex": "f",
   "born": 1876, "died": 1956,
   "father": "Petrus de Milliano",
   "mother": "Sophia van Damme"},
  {"name": "Carolus Haverbeke", "sex": "m",
   "born": 1832, "died": 1905,
   "father": "Carel Haverbeke",
   "mother": "Maria van Brussel"},
  … и так далее
]
```

Этот формат называется JSON, что означает JavaScript Object Notation (разметка объектов JavaScript). Он широко используется в хранении данных и сетевых коммуникациях.

JSON похож на JavaScript по способу записи массивов и объектов – с некоторыми ограничениями. Все имена свойств должны быть заключены в двойные кавычки, а

также допускаются только простые величины – никаких вызовов функций, переменных, ничего что включало бы вычисления. Также не допускаются комментарии.

JavaScript предоставляет функции JSON.stringify и JSON.parse, которые преобразовывают данные из этого формата и в этот формат. Первая принимает значение и возвращает строчку с JSON. Вторая принимает такую строчку и возвращает значение.

```
var string = JSON.stringify({name: "X", born: 1980});
console.log(string);
// → {"name":"X","born":1980}
console.log(JSON.parse(string).born);
// → 1980
```

Переменная ANCESTRY_FILE, доступная здесь, содержит JSON файл в виде строки. Давайте её раскодируем и посчитаем количество упомянутых людей.

```
var ancestry = JSON.parse(ANCESTRY_FILE);
console.log(ancestry.length);
// → 39
```

Фильтруем массив

Чтобы найти людей, которые были молоды в 1924 году, может пригодиться следующая функция. Она отфильтровывает элементы массива, которые не проходят проверку.

```
function filter(array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i]))
      passed.push(array[i]);
  }
  return passed;
}

console.log(filter(ancestry, function(person) {
  return person.born > 1900 && person.born < 1925;
}));
// → [{name: "Philibert Haverbeke", …}, …]
```

Используется аргумент с именем test – это функция, которая производит вычисления проверки. Она вызывается для каждого элемента, а возвращаемое ею значение определяет, попадает ли этот элемент в возвращаемый массив.

В файле оказалось три человека, которые были молоды в 1924 – дедушка, бабушка и двоюродная бабушка.

Обратите внимание, функция filter не удаляет элементы из существующего массива, а строит новый, содержащий только прошедшие проверку элементы. Это чистая функция, потому что она не портит переданный ей массив.

Как и forEach, filter – это один из стандартных методов массива. В примере мы описали такую функцию, только чтобы показать, что она делает внутри. Отныне мы будем использовать её просто:

```
console.log(ancestry.filter(function(person) {
  return person.father == "Carel Haverbeke";
}));
// → [{name: "Carolus Haverbeke", …}]
```

Преобразования при помощи map

Допустим, есть у нас массив объектов, представляющих людей, который был получен фильтрацией массива предков. Но нам нужен массив имён, который было бы проще прочесть.

Метод map преобразовывает массив, применяя функцию ко всем его элементам и строя новый массив из возвращаемых значений. У нового массива будет та же

длина, что у входного, но его содержимое будет преобразовано в новый формат.

```
function map(array, transform) {
  var mapped = [];
  for (var i = 0; i < array.length; i++)
    mapped.push(transform(array[i]));
  return mapped;
}

var overNinety = ancestry.filter(function(person) {
  return person.died - person.born > 90;
});
console.log(map(overNinety, function(person) {
  return person.name;
}));
// → ["Clara Aernoudts", "Emile Haverbeke",
//    "Maria Haverbeke"]
```

Что интересно, люди, которые прожили хотя бы до 90 лет – это те самые, что мы видели ранее, которые были молоды в 1920-х годах. Это как раз самое новое поколение в моих записях. Видимо, медицина серьёзно улучшилась.

Как и forEach и filter, map также является стандартным методом у массивов.

Суммирование при помощи reduce

Другой популярный пример работы с массивами – получение одиночного значения на основе данных в массиве. Один пример – уже знакомое нам суммирование списка номеров. Другой – поиск человека, родившегося раньше всех.

Операция высшего порядка такого типа называется reduce (уменьшение; или иногда fold, свёртывание). Можно представить её в виде складывания массива, по одному элементу за раз. При суммировании чисел мы начинали с нуля, и для каждого элемента комбинировали его с текущей суммой при помощи сложения.

Параметры функции reduce, кроме массива – комбинирующая функция и начальное значение. Эта функция чуть менее понятная, чем filter или map, поэтому обратите на неё пристальное внимание.

```
function reduce(array, combine, start) {
  var current = start;
  for (var i = 0; i < array.length; i++)
    current = combine(current, array[i]);
  return current;
}

console.log(reduce([1, 2, 3, 4], function(a, b) {
  return a + b;
}, 0));
// → 10
```

Стандартный метод массивов reduce, который, конечно, работает так же, ещё более удобен. Если массив содержит хотя бы один элемент, вы можете не указывать аргумент start. Метод возьмёт в качестве стартового значения первый элемент массива и начнёт работу со второго.

Чтобы при помощи reduce найти самого древнего из известных моих предков, мы можем написать нечто вроде:

```
console.log(ancestry.reduce(function(min, cur) {
  if (cur.born < min.born) return cur;
  else return min;
}));
// → {name: "Pauwels van Haverbeke", born: 1535, …}
```

Компонуемость

Как бы мы могли написать предыдущий пример (поиск человека с самой ранней датой рождения) без функций высшего порядка? На самом деле, код не такой уж и ужасный:

```
var min = ancestry[0];
for (var i = 1; i < ancestry.length; i++) {
  var cur = ancestry[i];
  if (cur.born < min.born)
    min = cur;
}
console.log(min);
// → {name: "Pauwels van Haverbeke", born: 1535, …}
```

Чуть больше переменных, на две строчки длиннее – но пока достаточно понятный код.

Функции высшего порядка раскрывают свои возможности по-настоящему, когда вам приходится комбинировать функции. К примеру, напишем код, находящий средний возраст мужчин и женщин в наборе.

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }

console.log(average(ancestry.filter(male).map(age)));
// → 61.67
console.log(average(ancestry.filter(female).map(age)));
// → 54.56
```

(Глупо, что нам приходится определять сложение как функцию plus, но операторы в JavaScript не являются значениями, поэтому их не передашь в качестве аргументов.)

Вместо того, чтобы впутывать алгоритм в большой цикл, всё распределено по концепциям, которые нас интересуют – определение пола, подсчёт возраста и усреднение чисел. Мы применяем их по очереди для получения результата.

Для написания понятного кода это прямо-таки сказочная возможность. Конечно, ясность не достаётся бесплатно.

Цена

В счастливом краю элегантного кода и красивых радуг живёт гадское чудище по имени Неэффективность.

Программа, обрабатывающая массив, красивее всего представляется в виде последовательности явно разделённых шагов, каждый из которых что-то делает с массивом и возвращает новый массив. Но наслоение всех этих промежуточных массивов стоит дорого.

Точно так же, передача функции в forEach, чтобы та прошлась по массиву за нас, удобна и проста в понимании. Но вызов функций в JavaScript обходится дороже по сравнению с циклами.

Так же обстоят дела со многими техниками, улучшающими читаемость программ. Абстракции добавляют слои между чистой работой компьютера и теми концепциями, с которыми мы работаем – и в результате компьютер делает больше работы. Это не железное правило – есть языки, которые позволяют добавлять абстракции без ухудшения эффективности, и даже в JavaScript опытный программист может найти способы писать абстрактный и быстрый код. Но это проблема встречается часто.

К счастью, большинство компьютеров безумно быстрые. Если ваш набор данных не слишком велик, или время работы должно быть всего лишь достаточно быстрым с

точки зрения человека (например, делать что-то каждый раз, когда пользователь жмёт на кнопку) – тогда не имеет значения, написали вы красивое решение, которое работает половину миллисекунды, или очень оптимизированное, которое работает одну десятую миллисекунды.

Удобно примерно подсчитывать, как часто будет вызываться данный кусочек кода. Если у вас есть цикл в цикле (напрямую, или же через вызов в цикле функции, которая внутри также работает с циклом), то код будет выполнен N×M раз, где N – количество повторений внешнего цикла, а M – внутреннего. Если во внутреннем цикле есть ещё один цикл, повторяющийся P раз, тогда мы уже получим N×M×P, и так далее. Это может приводить к большим числам, и когда программа тормозит, проблему часто можно локализовать в небольшом кусочке кода, находящемся внутри самого внутреннего цикла.

Пра-пра-пра-пра-пра-…

Мой дед, Филиберт Хавербеке, упомянут в файле с данными. Начиная с него я могу отследить свой род в поисках самого древнего из предков, Паувелса ван

Хавербеке, моего прямого предка. Теперь я хочу подсчитать, какой процент ДНК у меня от него (в теории).

Чтобы пройти от имени предка до объекта, представляющего его, мы строим объект, который сопоставляет имена и людей.

```
var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});

console.log(byName["Philibert Haverbeke"]);
// → {name: "Philibert Haverbeke", …}
```

Задача – не просто найти у каждой из записей отца и посчитать, сколько шагов получается до Паувелса. В истории семьи было несколько браков между двоюродными родственниками (ну, маленькие деревни и т.д.). В связи с этим ветви семейного дерева в некоторых местах соединяются с другими, поэтому генов у меня получается больше, чем 1/2 в степени G (G – количество поколений между Паувелсом и мною). Эта формула исходит из предположения, что каждое поколение расщепляет генетический фонд надвое.

Разумно будет провести аналогию с reduce, где массив низводится до единственного значения путём последовательного комбинирования данных слева

направо. Здесь нам тоже надо получить единственное число, но при этом нужно следовать линиям наследственности. А они формируют не простой список, а дерево.

Мы считаем это значение для конкретного человека, комбинируя эти значения его предков. Это можно сделать рекурсивно. Если нам нужен какой-то человек, нам надо подсчитать нужную величину для его родителей, что в свою очередь требует подсчёта её для его прародителей, и т.п. По идее нам придётся обойти бесконечное множество узлов дерева, но так как наш набор данных конечен, нам надо будет где-то остановиться. Мы просто назначим значение по умолчанию для всех людей, которых нет в нашем списке. Логично будет назначить им нулевое значение – люди, которых нет в списке, не несут в себе ДНК нужного нам предка.

Принимая данные о человеке, функцию для комбинирования значений от двух предков и значение по умолчанию, функция reduceAncestors «конденсирует» значение из семейного древа.

```
function reduceAncestors(person, f, defaultValue) {
  function valueFor(person) {
    if (person == null)
      return defaultValue;
    else
      return f(person, valueFor(byName[person.mother]),
                       valueFor(byName[person.father]));
  }
  return valueFor(person);
}
```

Внутренняя функция valueFor работает с одним человеком. Благодаря рекурсивной магии она может вызвать себя для обработки отца и матери этого человека. Результаты вместе с объектом person передаются в f, которая и вычисляет нужное значение для этого человека.

Теперь мы можем использовать это для подсчёта процента ДНК, которое мой дедушка разделил с Паувелсом ванн Хавербеке, и поделить это на четыре.

```
function sharedDNA(person, fromMother, fromFather) {
  if (person.name == "Pauwels van Haverbeke")
    return 1;
  else
    return (fromMother + fromFather) / 2;
}
var ph = byName["Philibert Haverbeke"];
console.log(reduceAncestors(ph, sharedDNA, 0) / 4);
// → 0.00049
```

Человек по имени Паувелс ванн Хавербеке, очевидно, делит 100% ДНК с Паувелсом ванн Хавербеке (полных тёзок в списке данных нет), поэтому для него функция возвращает 1. Все остальные делят средний процент их родителей.

Статистически, у меня примерно 0,05% ДНК совпадает с моим предком из XVI века. Это, конечно, приблизительное число. Это довольно мало, но так как наш генетический материал составляет примерно 3 миллиарда базовых пар, во мне есть что-то от моего предка.

Можно было бы подсчитать это число и без использования reduceAncestors. Но разделение общего подхода (обход древа) и конкретного случая (подсчёт ДНК) позволяет нам писать более понятный код и

использовать вновь части кода для других задач. Например, следующий код выясняет процент известных предков данного человека, доживших до 70 лет.

```
function countAncestors(person, test) {
  function combine(person, fromMother, fromFather) {
    var thisOneCounts = test(person);
    return fromMother + fromFather + (thisOneCounts ? 1
: 0);
  }
  return reduceAncestors(person, combine, 0);
}
function longLivingPercentage(person) {
  var all = countAncestors(person, function(person) {
    return true;
  });
  var longLiving = countAncestors(person, function(person
) {
    return (person.died - person.born) >= 70;
  });
  return longLiving / all;
}
console.log(longLivingPercentage(byName["Emile Haverbeke"
]));
// → 0.145
```

Не нужно относиться к таким расчётам слишком серьёзно, так как наш набор содержит произвольную выборку людей. Но код демонстрирует, что reduceAncestors – полезная часть общего словаря для работы со структурой данных типа фамильного древа.

Связывание

Метод bind, который есть у всех функций, создаёт новую функцию, которая вызовет оригинальную, но с некоторыми фиксированными аргументами.

Следующий пример показывает, как это работает. В нём мы определяем функцию isInSet, которая говорит, есть ли имя человека в заданном наборе. Для вызова filter мы можем либо написать выражение с функцией, которое вызывает isInSet, передавая ей набор строк в качестве первого аргумента, или применить функцию isInSet частично.

```
var theSet = ["Carel Haverbeke", "Maria van Brussel",
              "Donald Duck"];
function isInSet(set, person) {
  return set.indexOf(person.name) > -1;
}

console.log(ancestry.filter(function(person) {
  return isInSet(theSet, person);
}));
// → [{name: "Maria van Brussel", …},
//    {name: "Carel Haverbeke", …}]
console.log(ancestry.filter(isInSet.bind(null, theSet)))
;
// → … тот же результат
```

Вызов bind возвращает функцию, которая вызовет isInSet с первым аргументом theSet, и последующими аргументами такими же, какие были переданы в bind.

Первый аргумент, который сейчас установлен в null, используется для вызовов методов – так же, как было в apply. Мы поговорим об этом позже.

Итог

Возможность передавать вызов функции другим функциям – не просто игрушка, но очень полезное свойство JavaScript. Мы можем писать выражения «с пробелами» в них, которые затем будут заполнены при помощи значений, возвращаемых функциями.

У массивов есть несколько полезных методов высшего порядка – forEach, чтобы сделать что-то с каждым элементом, filter – чтобы построить новый массив, где некоторые значения отфильтрованы, map – чтобы построить новый массив, каждый элемент которого пропущен через функцию, reduce – для комбинации всех элементов массива в одно значение.

У функций есть метод apply для передачи им аргументов в виде массива. Также у них есть метод bind для создания копии функции с частично заданными аргументами.

Упражнения

Свёртка

Используйте метод reduce в комбинации с concat для свёртки массива массивов в один массив, у которого есть все элементы входных массивов.

```
var arrays = [[1, 2, 3], [4, 5], [6]];
// Ваш код тут
// → [1, 2, 3, 4, 5, 6]
```

Разница в возрасте матерей и их детей

Используя набор данных из примера, подсчитайте среднюю разницу в возрасте между матерями и их детьми (это возраст матери во время появления ребёнка). Можно использовать функцию average, приведённую в главе.

Обратите внимание – не все матери, упомянутые в наборе, присутствуют в нём. Здесь может пригодиться объект byName, который упрощает процедуру поиска объекта человека по имени.

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}

var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});

// Ваш код тут

// → 31.2
```

Историческая ожидаемая продолжительность жизни

Мы считали, что только последнее поколение людей дожило до 90 лет. Давайте рассмотрим этот феномен поподробнее. Подсчитайте средний возраст людей для каждого из столетий. Назначаем столетию людей, беря их год смерти, деля его на 100 и округляя: Math.ceil(person.died / 100).

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}

// Тут ваш код

// → 16: 43.5
//   17: 51.2
//   18: 52.8
//   19: 54.8
//   20: 84.7
//   21: 94
```

В качестве призовой игры напишите функцию groupBy, абстрагирующую операцию группировки. Она должна принимать массив и функцию, которая вычисляет группу для элементов массива, и возвращать объект, который сопоставляет названия групп массивам членов этих групп.

Every и some

У массивов есть ещё стандартные методы every и some. Они принимают как аргумент некую функцию, которая, будучи вызванной с элементом массива в качестве аргумента, возвращает true или false. Так же, как && возвращает true, только если выражения с обеих сторон оператора возвращают true, метод every возвращает true,

когда функция возвращает true для всех элементов массива. Соответственно, some возвращает true, когда заданная функция возвращает true при работе хотя бы с одним из элементов массива. Они не обрабатывают больше элементов, чем необходимо – например, если some получает true для первого элемента, он не обрабатывает оставшиеся.

Напишите функции every и some, которые работают так же, как эти методы, только принимают массив в качестве аргумента.

```
// Ваш код тут

console.log(every([NaN, NaN, NaN], isNaN));
// → true
console.log(every([NaN, NaN, 4], isNaN));
// → false
console.log(some([NaN, 3, 4], isNaN));
// → true
console.log(some([2, 3, 4], isNaN));
// → false
```

Тайная жизнь объектов

Проблема объектно-ориентированных языков в том, что они тащат с собой всё своё неявное окружение. Вам нужен был банан – а вы получаете гориллу с бананом, и целые джунгли впридачу.

Джо Армстронг, в интервью Coders at Work

Термин «объект» в программировании сильно перегружен значениями. В моей профессии объекты – стиль жизни, тема священных войн и любимое заклинание, не теряющее своей магической силы.

Стороннему человеку всё это непонятно. Начнём же с краткой истории объектов как концепции в программировании.

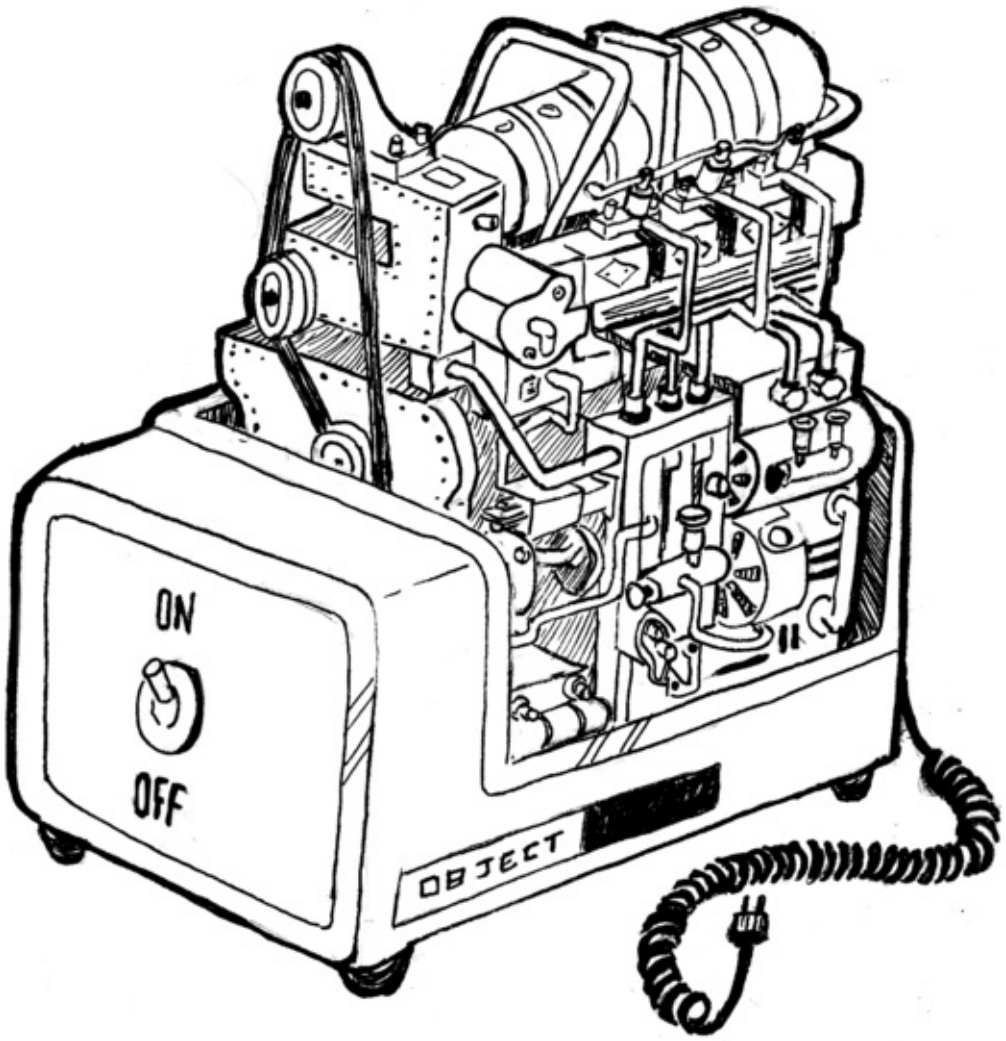
История

Эта история, как большинство историй о программировании, начинается с проблемы сложности. Одна из идей говорит, что сложность можно сделать

управляемой, разделив её на небольшие части, изолированные друг от друга. Эти части стали называть объектами.

Объект – твёрдая скорлупа, скрывающая липкую сложность внутри, и вместо неё предлагающая нам несколько ручек настройки и контактов (вроде методов), представляющих интерфейс, посредством которого объект нужно использовать. Идея в том, что интерфейс относительно прост, и при работе с ним позволяет игнорировать все сложные процессы, происходящие внутри объекта.

Простой интерфейс может спрятать много сложного.



Для примера представьте объект, обеспечивающий интерфейс к участку экрана. С его помощью можно рисовать фигуры или выводить текст на этот участок, но при этом все детали, касающиеся превращения текста или фигур в пиксели, скрыты. У вас есть набор методов, к примеру drawCircle, и это всё, что вам нужно знать для использования такого объекта.

Такие идеи получили развитие в 70-80 годах, а в 90-х их вынесла на поверхность рекламная волна – революция объектно-ориентированного программирования.

Внезапно большой клан людей объявил, что объекты – это правильный способ программирования. А всё, что не имеет объектов, является устаревшей ерундой.

Такой фанатизм всегда приводит к куче бесполезной чуши, и с тех пор идёт что-то вроде контрреволюции. В некоторых кругах объекты вообще имеют крайне плохую репутацию.

Я предпочитаю рассматривать их с практической, а не идеологической точки зрения. Есть несколько полезных идей, в частности инкапсуляция (различие между внутренней сложностью и внешней простотой), которые были популяризованы объектно-ориентированной культурой. Их стоит изучать.

Эта глава описывает довольно эксцентричный подход JavaScript к объектам, и то, как они соотносятся с классическими объектно-ориентированными техниками.

Методы

Методы – свойства, содержащие функции. Простой метод:

```
var rabbit = {};
rabbit.speak = function(line) {
  console.log("Кролик говорит '" + line + "'");
};

rabbit.speak("Я живой.");
// → Кролик говорит 'Я живой.'
```

Обычно метод должен что-то сделать с объектом, через который он был вызван. Когда функцию вызывают в виде метода – как свойство объекта, например object.method() – специальная переменная в её теле будет указывать на вызвавший её объект.

```
function speak(line) {
  console.log("А " + this.type + " кролик говорит '" + l
ine + "'");
}
var whiteRabbit = {type: "белый", speak: speak};
var fatRabbit = {type: "толстый", speak: speak};

whiteRabbit.speak("Ушки мои и усики, я же наверняка опаз
дываю!");
// → А белый кролик говорит 'Ушки мои и усики, я же наве
рняка опаздываю!'
fatRabbit.speak("Мне бы сейчас морковочки.");
// → А толстый кролик говорит 'Мне бы сейчас морковочки.'
```

Код использует ключевое слово this для вывода типа говорящего кролика.

Вспомните, что методы apply и bind принимают первый аргумент, который можно использовать для эмуляции вызова методов. Этот первый аргумент как раз даёт значение переменной this.

Есть метод, похожий на apply, под названием call. Он тоже вызывает функцию, методом которой является, только принимает аргументы как обычно, а не в виде массива. Как apply и bind, в call можно передать значение this.

```
speak.apply(fatRabbit, ["Отрыжка!"]);
// → А толстый кролик говорит 'Отрыжка!'
speak.call({type: "старый"}, "О, господи.");
// → А старый кролик говорит 'О, господи.'
```

Прототипы

Следите за руками.

```
var empty = {};
console.log(empty.toString);
// → function toString(){…}
console.log(empty.toString());
// → [object Object]
```

Я достал свойство пустого объекта. Магия!

Ну, не магия, конечно. Я просто не всё рассказал про то, как работают объекты в JavaScript. В дополнение к набору свойств, почти у всех также есть прототип. Прототип – это ещё один объект, который используется как запасной источник свойств. Когда объект получает запрос на свойство, которого у него нет, это свойство ищется у его прототипа, затем у прототипа прототипа, и т. д.

Ну а кто же прототип пустого объекта? Это великий предок всех объектов, Object.prototype.

```
console.log(Object.getPrototypeOf({}) == Object.prototyp
e);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

Как и следовало ожидать, функция Object.getPrototypeOf возвращает прототип объекта.

Прототипические отношения в JavaScript выглядят как дерево, в корне которого находится Object.prototype. Он предоставляет несколько методов, которые появляются у всех объектов. Например, toString, который преобразует объект в строковый вид.

Прототипом многих объектов служит не непосредственно Object.prototype, а какой-то другой объект, который предоставляет свои свойства по умолчанию. Функции происходят от Function.prototype, массивы – от Array.prototype.

```
console.log(Object.getPrototypeOf(isNaN) == Function.pro
totype);
// → true
console.log(Object.getPrototypeOf([]) == Array.prototype
);
// → true
```

У таких прототипов будет свой прототип – часто Object.prototype, поэтому он всё равно, хоть и не напрямую, предоставляет им методы типа toString.

Функция Object.getPrototypeOf возвращает прототип объекта. Можно использовать Object.create для создания объектов с заданным прототипом.

```
var protoRabbit = {
  speak: function(line) {
    console.log("A " + this.type + " кролик говорит '" +
  line + "'");
  }
};
var killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "убийственный";
killerRabbit.speak("ХРЯЯЯСЬ!");
// → A убийственный кролик говорит 'ХРЯЯЯСЬ!'
```

Прото-кролик работает в качестве контейнера свойств, которые есть у всех кроликов. Конкретный объект-кролик, например убийственный, содержит свойства, применимые только к нему – например, свой тип – и наследует разделяемые с другими свойства от прототипа.

Конструкторы

Более удобный способ создания объектов, наследуемых от некоего прототипа – конструктор. В JavaScript вызов функции с предшествующим ключевым словом new приводит к тому, что функция работает как конструктор. Конструктор создает новый объект и возвращает его, если только явно не задано возвращение другого объекта вместо созданного. При этом свежесозданный объект доступен изнутри конструктора через переменную this.

Говорят, что объект, созданный при помощи new, является экземпляром конструктора.

Вот простой конструктор кроликов. Имена конструкторов принято начинать с заглавной буквы, чтобы отличать их от других функций.

```
function Rabbit(type) {
  this.type = type;
}

var killerRabbit = new Rabbit("убийственный");
var blackRabbit = new Rabbit("чёрный");
console.log(blackRabbit.type);
// → чёрный
```

Конструкторы (а вообще-то, и все функции) автоматически получают свойство под именем prototype, которое по умолчанию содержит простой пустой объект, происходящий от Object.prototype. Каждый экземпляр, созданный этим конструктором, будет иметь этот объект в качестве прототипа. Поэтому, чтобы добавить кроликам, созданным конструктором Rabbit, метод speak, мы просто можем сделать так:

```
Rabbit.prototype.speak = function(line) {
  console.log("А " + this.type + " кролик говорит '" +
 line + "'");
};
blackRabbit.speak("Всем капец...");
// → А чёрный кролик говорит 'Всем капец...'
```

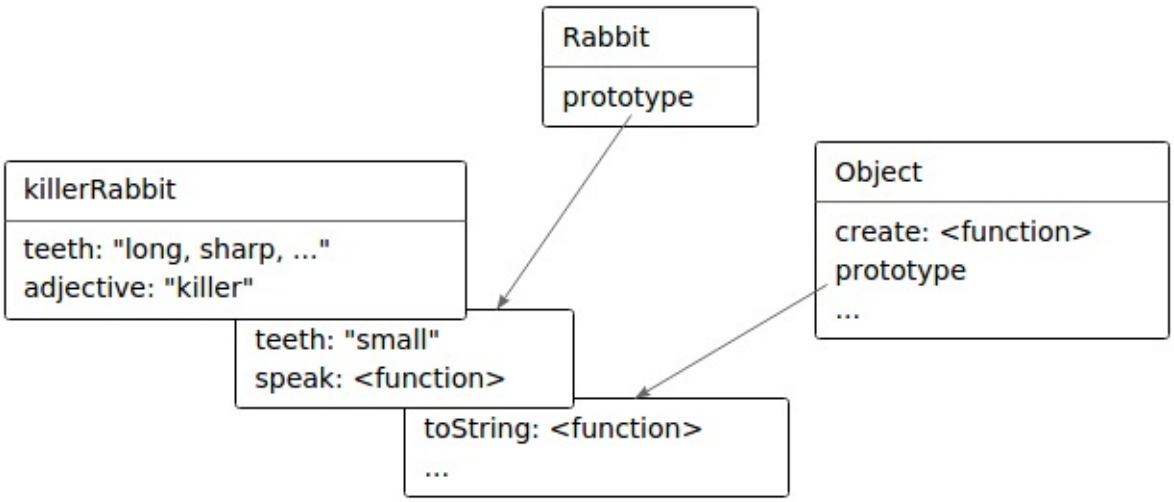
Важно отметить разницу между тем, как прототип связан с конструктором (через свойство prototype) и тем, как у объектов есть прототип (который можно получить через Object.getPrototypeOf). На самом деле прототип конструктора – Function.prototype, поскольку конструкторы – это функции. Его свойство prototype будет прототипом экземпляров, созданных им, но не его прототипом.

Перегрузка унаследованных свойств

Когда вы добавляете свойство объекту, есть оно в прототипе или нет, оно добавляется непосредственно к самому объекту. Теперь это его свойство. Если в прототипе есть одноимённое свойство, оно больше не влияет на объект. Сам прототип не меняется.

```
Rabbit.prototype.teeth = "мелкие";
console.log(killerRabbit.teeth);
// → мелкие
killerRabbit.teeth = "длинные, острые и окровавленные";
console.log(killerRabbit.teeth);
// → длинные, острые и окровавленные
console.log(blackRabbit.teeth);
// → мелкие
console.log(Rabbit.prototype.teeth);
// → мелкие
```

На диаграмме нарисована ситуация после прогона кода. Прототипы Rabbit и Object находятся за killerRabbit на манер фона, и у них можно запрашивать свойства, которых нет у самого объекта.



Перегрузка свойств, существующих в прототипе, часто приносит пользу. Пример с зубами кролика показывает, как её можно использовать для выражения каких-то

исключительных характеристик конкретных экземпляров объектов, оставляя прочим стандартные значения из прототипа.

Та же перегрузка используется, чтобы дать стандартным функциям и массивам свои методы toString, отличные от метода базового объекта.

```
console.log(Array.prototype.toString == Object.prototype
.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Вызов toString массива выводит результат, похожий на .join(",") – получается список, разделённый запятыми. Вызов Object.prototype.toString напрямую для массива приводит к другому результату. Эта функция не знает ничего о массивах:

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Нежелательное взаимодействие прототипов

Прототип помогает в любое время добавлять новые свойства и методы всем объектам, которые основаны на нём. К примеру, нашим кроликам может понадобиться танец.

```
Rabbit.prototype.dance = function() {
  console.log("А " + this.type + " кролик танцует джигу."
);
};
killerRabbit.dance();
// → А убийственный кролик танцует джигу.
```

Это удобно. Но в некоторых случаях это приводит к проблемам. В предыдущих главах мы использовали объект как способ связать значения с именами – мы создавали свойства для этих имён, и давали им соответствующие значения. Вот пример из 4-й главы:

```
var map = {};
function storePhi(event, phi) {
  map[event] = phi;
}

storePhi("пицца", 0.069);
storePhi("тронул дерево", -0.081);
```

Мы можем перебрать все значения фи в объекте через цикл for/in, и проверить наличие в нём имени через оператор in. К сожалению, нам мешается прототип

объекта.

```
Object.prototype.nonsense = "ку";
for (var name in map)
  console.log(name);
// → пицца
// → тронул дерево
// → nonsense
console.log("nonsense" in map);
// → true
console.log("toString" in map);
// → true

// Удалить проблемное свойство
delete Object.prototype.nonsense;
```

Это же неправильно. Нет события под названием “nonsense”. И тем более нет события под названием “toString”.

Занятно, что toString не вылезло в цикле for/in, хотя оператор in возвращает true на его счёт. Это потому, что JavaScript различает счётные и несчётные свойства.

Все свойства, которые мы создаём, назначая им значение – счётные. Все стандартные свойства в Object.prototype – несчётные, поэтому они не вылезают в циклах for/in.

Мы можем объявить свои несчётные свойства через функцию Object.defineProperty, которая позволяет указывать тип создаваемого свойства.

```
Object.defineProperty(Object.prototype, "hiddenNonsense"
, {
  enumerable: false, value: "ку"
});
for (var name in map)
  console.log(name);
// → пицца
// → тронул дерево
console.log(map.hiddenNonsense);
// → ку
```

Теперь свойство есть, а в цикле оно не вылезает. Хорошо. Но нам всё ещё мешает проблема с оператором in, который утверждает, что свойства Object.prototype присутствуют в нашем объекте. Для этого нам понадобится метод hasOwnProperty.

```
console.log(map.hasOwnProperty("toString"));
// → false
```

Он говорит, является ли свойство свойством объекта, без оглядки на прототипы. Часто это более полезная информация, чем выдаёт оператор in.

Если вы волнуетесь, что кто-то другой, чей код вы загрузили в свою программу, испортил основной прототип объектов, я рекомендую писать циклы for/in так:

```
for (var name in map) {
  if (map.hasOwnProperty(name)) {
    // ... это наше личное свойство
  }
}
```

Объекты без прототипов

Но кроличья нора на этом не заканчивается. А если кто-то зарегистрировал имя hasOwnProperty в объекте map и назначил ему значение 42? Теперь вызов map.hasOwnProperty обращается к локальному свойству, в котором содержится номер, а не функция.

В таком случае прототипы только мешаются, и нам бы хотелось иметь объекты вообще без прототипов. Мы видели функцию Object.create, что позволяет создавать объект с заданным прототипом. Мы можем передать null для прототипа, чтобы создать свеженький объект без прототипа. Это то, что нам нужно для объектов типа map, где могут быть любые свойства.

```
var map = Object.create(null);
map["пицца"] = 0.069;
console.log("toString" in map);
// → false
console.log("пицца" in map);
// → true
```

Так-то лучше! Нам уже не нужна приблуда hasOwnProperty, потому что все свойства объекта заданы лично нами. Мы спокойно используем циклы for/in без оглядки на то, что люди творили с Object.prototype

Полиморфизм

Когда вы вызываете функцию String, преобразующую значение в строку, для объекта, он вызовет метод toString, чтобы создать осмысленную строчку. Я упомянул, что некоторые стандартные прототипы объявляют свои версии toString для создания строк, более полезных, чем просто "[object Object]".

Это простой пример мощной идеи. Когда кусок кода написан так, чтобы работать с объектами через определённый интерфейс – в нашем случае через метод toString – любой объект, поддерживающий этот интерфейс, можно подключить к коду, и всё будет просто работать.

Такая техника называется полиморфизм, хотя никто и не меняет своей формы. Полиморфный код может работать со значениями самых разных форм, пока они поддерживают одинаковый интерфейс.

Форматируем таблицу

Давайте рассмотрим пример, чтобы понять, как выглядит полиморфизм, да и вообще объектно-ориентированное программирование. Проект следующий: мы напишем программу, которая получает массив массивов из ячеек таблицы, и строит строку, содержащую красиво отформатированную таблицу. То есть, колонки и ряды выровнены. Типа вот этого:

```
name         height country
------------ ------ -------------
Kilimanjaro    5895 Tanzania
Everest        8848 Nepal
Mount Fuji     3776 Japan
Mont Blanc     4808 Italy/France
Vaalserberg     323 Netherlands
Denali         6168 United States
Popocatepetl   5465 Mexico
```

Работать она будет так: основная функция будет спрашивать каждую ячейку, какой она ширины и высоты, и потом использует эту информацию для определения

ширины колонок и высоты рядов. Затем она попросит ячейки нарисовать себя, и соберёт результаты в одну строку.

Программа будет общаться с объектами ячеек через хорошо определённый интерфейс. Типы ячеек не будут заданы жёстко. Мы сможем добавлять новые стили ячеек – к примеру, подчёркнутые ячейки у заголовка. И если они будут поддерживать наш интерфейс, они просто заработают, без изменений в программе. **Интерфейс:**

- **minHeight()** возвращает число, показывающее минимальную высоту, которую требует ячейка (выраженную в строчках)
- **minWidth()** возвращает число, показывающее минимальную ширину, которую требует ячейка (выраженную в символах)
- **draw(width, height)** возвращает массив длины height, содержащий наборы строк, каждая из которых шириной в width символов. Это содержимое ячейки.

Я буду использовать функции высшего порядка, поскольку они здесь очень уместны.

Первая часть программы вычисляет массивы минимальных ширин колонок и высот строк для матрицы ячеек. Переменная rows будет содержать массив

массивов, где каждый внутренний массив – это строка ячеек.

```
function rowHeights(rows) {
  return rows.map(function(row) {
    return row.reduce(function(max, cell) {
      return Math.max(max, cell.minHeight());
    }, 0);
  });
}

function colWidths(rows) {
  return rows[0].map(function(_, i) {
    return rows.reduce(function(max, row) {
      return Math.max(max, row[i].minWidth());
    }, 0);
  });
}
```

Используя переменную, у которой имя начинается с (или полностью состоит из) подчёркивания (_), мы показываем тому, кто будет читать код, что этот аргумент не будет использоваться.

Функция rowHeights не должна вызвать затруднений. Она использует reduce для подсчёта максимальной высоты массива ячеек, и заворачивает это в map, чтобы пройти все строки в массиве rows.

Ситуация с colWidths посложнее, потому что внешний массив – это массив строк, а не столбцов. Я забыл упомянуть, что map (как и forEach, filter и похожие методы массивов) передаёт в заданную функцию второй аргумент – индекс текущего элемента. Проходя при помощи map элементы первой строки и используя только второй аргумент функции, colWidths строит массив с одним элементом для каждого индекса столбца. Вызов reduce проходит по внешнему массиву rows для каждого индекса, и выбирает ширину широчайшей ячейки в этом индексе.

Код для вывода таблицы:

```
function drawTable(rows) {
  var heights = rowHeights(rows);
  var widths = colWidths(rows);

  function drawLine(blocks, lineNo) {
    return blocks.map(function(block) {
      return block[lineNo];
    }).join(" ");
  }

  function drawRow(row, rowNum) {
    var blocks = row.map(function(cell, colNum) {
      return cell.draw(widths[colNum], heights[rowNum]);
    });
    return blocks[0].map(function(_, lineNo) {
      return drawLine(blocks, lineNo);
    }).join("\n");
  }

  return rows.map(drawRow).join("\n");
}
```

Функциия drawTable использует внутреннюю функцию drawRow для рисования всех строк, соединяя их через символы новой строки.

Функция drawRow сперва превращает объекты ячеек строки в блоки, которые являются массивами строк, представляющими содержимое ячеек, разделённые линиями. Одна ячейка, содержащая число 3776, может

быть представлена массивом из одного элемента ["3776"], а подчёркнутая ячейка может занять две строки и выглядеть как массив ["name", "----"].

Блоки для строки, у которых одинаковая высота, должны выводиться рядом друг с другом. Второй вызов map в drawRow строит этот результат построчно, начиная с ячеек самого левого блока, и для каждой из них дополняя строку до полной ширины таблицы. Полученные строки затем соединяются через символ новой строки, создавая целый ряд, который и возвращает drawRow.

Функция drawLine выцепляет строки, которые должны располагаться рядом друг с другом, из массива блоков и соединяет их через пробел, чтобы создать промежуток в один символ между столбцами таблицы.

Давайте напишем конструктор для ячеек, содержащих текст, который предоставляет интерфейс для ячеек. Он разбивает строчку в массив строк при помощи метода split, который режет строчку каждый раз, когда в ней встречается его аргумент, и возвращает массив полученных кусочков. Метод minWidth находит максимальную ширину строки в массиве.

```
function repeat(string, times) {
  var result = "";
  for (var i = 0; i < times; i++)
    result += string;
  return result;
}

function TextCell(text) {
  this.text = text.split("\n");
}
TextCell.prototype.minWidth = function() {
  return this.text.reduce(function(width, line) {
    return Math.max(width, line.length);
  }, 0);
};
TextCell.prototype.minHeight = function() {
  return this.text.length;
};
TextCell.prototype.draw = function(width, height) {
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(line + repeat(" ", width - line.length))
;
  }
  return result;
};
```

В коде используется вспомогательная функция repeat, которая возвращает переданную первым аргументом строку, повторённую переданное вторым аргументом

количество раз. Метод draw использует её для создания отступов в ячейках, чтобы они все были необходимой длины.

Давайте нарисуем для опыта шахматную доску 5x5.

```
var rows = [];
for (var i = 0; i < 5; i++) {
   var row = [];
   for (var j = 0; j < 5; j++) {
     if ((j + i) % 2 == 0)
       row.push(new TextCell("##"));
     else
       row.push(new TextCell("  "));
   }
   rows.push(row);
}
console.log(drawTable(rows));
// → ##    ##    ##
//      ##    ##
//   ##    ##    ##
//      ##    ##
//   ##    ##    ##
```

Работает! Но так как у всех ячеек один размер, код форматирования таблицы не делает ничего интересного.

Исходные данные для таблицы гор, которую мы строим, содержатся в переменной MOUNTAINS, их можно скачать тут.

Нам нужно выделить верхнюю строку, содержащую названия столбцов, при помощи подчёркивания. Никаких проблем – мы просто создаём тип ячейки, который этим занимается.

```
function UnderlinedCell(inner) {
  this.inner = inner;
};
UnderlinedCell.prototype.minWidth = function() {
  return this.inner.minWidth();
};
UnderlinedCell.prototype.minHeight = function() {
  return this.inner.minHeight() + 1;
};
UnderlinedCell.prototype.draw = function(width, height)
{
  return this.inner.draw(width, height - 1)
    .concat([repeat("-", width)]);
};
```

Подчёркнутая ячейка содержит другую ячейку. Она возвращает такие же размеры, как и у ячейки inner (через вызовы её методов minWidth и minHeight), но добавляет единичку к высоте из-за места, занятого чёрточками.

Рисовать её просто – мы берём содержимое ячейки inner и добавляем одну строку, заполненную чёрточками.

Теперь, имея основной движок, мы можем написать функцию, строящую сетку ячеек из нашего набора данных.

```
function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      return new TextCell(String(row[name]));
    });
  });
  return [headers].concat(body);
}

console.log(drawTable(dataTable(MOUNTAINS)));
// → name         height country
//   ------------ ------ -------------
//   Kilimanjaro  5895   Tanzania
//   … и так далее
```

Стандартная функция Object.keys возвращает массив имён свойств объекта. Верхняя строка таблицы состоит из подчёркнутых ячеек с заголовками столбцов. Всё что ниже – значения из набора данных – имеет вид обычных ячеек. Мы извлекаем эти данные проходом функции map по массиву keys, чтобы гарантировать одинаковый порядок ячеек в каждой из строк.

Итоговая таблица напоминает таблицу из примера, только вот числа не выровнены по правому краю. Мы займёмся этим чуть позже.

Геттеры и сеттеры

При создании интерфейса можно ввести свойства, не являющиеся методами. Мы могли бы определить minHeight и minWidth как переменные для хранения чисел. Но это потребовало бы от нас написать код вычисления их значений в конструкторе, что плохо, поскольку эти операции не связаны напрямую с конструированием объекта. Это может аукнуться, когда, например, внутренняя ячейка подчёркнутой ячейки изменяется – в этот момент размер подчеркивания тоже должен измениться.

Эти соображения привели к тому, что свойства, не являющиеся методами, многие не включают в интерфейс. Вместо прямого доступа к свойствам-значениям, используются методы типа getSomething и setSomething для чтения и записи значений свойств. Но в таком подходе есть и минус – приходится писать (и читать) много дополнительных методов.

К счастью, JavaScript даёт нам технику, использующую лучшее из обоих подходов. Мы можем задать свойства, которые снаружи выглядят обыкновенными, но втайне имеют связанные с ними методы.

```
var pile = {
  elements: ["скорлупа", "кожура", "червяк"],
  get height() {
    return this.elements.length;
  },
  set height(value) {
    console.log("Игнорируем попытку задать высоту", valu
e);
  }
};

console.log(pile.height);
// → 3
pile.height = 100;
// → Игнорируем попытку задать высоту 100
```

В объявлении объекта записи get или set позволяют задать функцию, которая будет вызвана при чтении или записи свойства. Можно также добавить такое свойство в существующий объект, к примеру, в prototype, используя функцию Object.defineProperty (раньше мы её уже использовали, создавая несчётные свойства).

```
Object.defineProperty(TextCell.prototype, "heightProp",
{
  get: function() { return this.text.length; }
});

var cell = new TextCell("да\nну");
console.log(cell.heightProp);
// → 2
cell.heightProp = 100;
console.log(cell.heightProp);
// → 2
```

Так же можно задавать свойство set в объекте, передаваемом в defineProperty, для задания метода-сеттера. Когда геттер есть, а сеттера нет, попытка записи в свойство просто игнорируется.

Наследование

Но мы ещё не закончили с нашим упражнением по форматированию таблицы. Читать её было бы удобнее, если б числовой столбец был выровнен по правому краю. Нам нужно создать ещё один тип ячеек вроде TextCell, но чтобы текст дополнялся пробелами слева, а не справа — для выравнивания по правому краю.

Мы могли бы написать новый конструктор со всеми тремя методами в прототипе. Но прототипы могут сами иметь прототипы, и поэтому мы можем поступить умнее.

```
function RTextCell(text) {
  TextCell.call(this, text);
}
RTextCell.prototype = Object.create(TextCell.prototype);
RTextCell.prototype.draw = function(width, height) {
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(repeat(" ", width - line.length) + line)
;
  }
  return result;
};
```

Мы повторно использовали конструктор и методы minHeight и minWidth из обычного TextCell. И RTextCell теперь в общем эквивалентен TextCell, за исключением того, что в методе draw находится другая функция.

Такая схема называется наследованием. Мы можем строить в чём-то отличные типы данных на основе существующих, не тратя много сил. Обычно новый конструктор вызывает старый (через метод call, чтобы передать ему новый объект и его значение). После этого мы можем предположить, что все поля, которые должны быть в старом объекте, добавлены. Мы наследуем

прототип конструктора от старого так, что экземпляры этого типа будут иметь доступ к свойствам старого прототипа. И наконец, мы можем переопределить некоторые свойства, добавляя их к новому прототипу.

Если мы чуть отредактируем функцию dataTable, чтоб она использовала для числовых ячеек RTextCells, мы получим нужную нам таблицу.

```
function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      var value = row[name];
      // Тут поменяли:
      if (typeof value == "number")
        return new RTextCell(String(value));
      else
        return new TextCell(String(value));
    });
  });
  return [headers].concat(body);
}

console.log(drawTable(dataTable(MOUNTAINS)));
// → … красиво отформатированная таблица
```

Наследование – основная часть объектно-ориентированной традиции, вместе с инкапсуляцией и полиморфизмом. Но, в то время как последние две воспринимают как отличные идеи, первая вызывает споры.

В основном потому, что её обычно путают с полиморфизмом, представляют более мощным инструментом, чем она на самом деле является, и используют не по назначению. Тогда как инкапсуляция и полиморфизм используются для разделения частей кода и уменьшения связанности программы, наследование связывает типы вместе и создаёт большую связанность.

Мы можем использовать полиморфизм без наследования. Я не советую вам полностью избегать наследования – я его использую регулярно в своих программах. Но относитесь к нему как к более хитрому трюку, который позволяет определять новые типы с минимумом кода – а не как к основному принципу организации кода. Предпочтительно расширять типы при помощи композиции – как UnderlinedCell построен на использовании другого объекта ячейки. Он просто хранит его в свойстве и перенаправляет вызовы из своих в его методы.

Оператор instanceof

Иногда удобно знать, произошёл ли объект от конкретного конструктора. Для этого JavaScript даёт нам бинарный оператор instanceof.

```
console.log(new RTextCell("A") instanceof RTextCell);
// → true
console.log(new RTextCell("A") instanceof TextCell);
// → true
console.log(new TextCell("A") instanceof RTextCell);
// → false
console.log([1] instanceof Array);
// → true
```

Оператор проходит и через наследованные типы. RTextCell является экземпляром TextCell, поскольку RTextCell.prototype происходит от TextCell.prototype. Оператор также можно применять к стандартным конструкторам типа Array. Практически все объекты – экземпляры Object.

Итог

Получается, что объекты чуть более сложны, чем я их подавал сначала. У них есть прототипы – это другие объекты, и они ведут себя так, как будто у них есть

свойство, которого на самом деле нет, если это свойство есть у прототипа. Прототипом простых объектов является Object.prototype.

Конструкторы – функции, имена которых обычно начинаются с заглавной буквы – можно использовать с оператором new для создания объектов. Прототипом нового объекта будет объект, содержащийся в свойстве prototype конструктора. Это можно использовать, помещая в прототип свойства, общие для всех экземпляров данного типа. Оператор instanceof, если ему дать объект и конструктор, может сказать, является ли объект экземпляром этого конструктора.

Для объектов можно сделать интерфейс и сказать всем, чтобы они общались с объектом только через этот интерфейс. Остальные детали реализации объекта теперь инкапсулированы, скрыты за интерфейсом.

А после этого никто не запрещал использовать разные объекты при помощи одинаковых интерфейсов. Если разные объекты имеют одинаковые интерфейсы, то и код, работающий с ними, может работать с разными объектами одинаково. Это называется полиморфизмом, и это очень полезная штука.

Определяя несколько типов, различающихся только в мелких деталях, бывает удобно просто наследовать прототип нового типа от прототипа старого типа, чтобы новый конструктор вызывал старый. Это даёт вам тип объекта, сходный со старым, но при этом к нему можно добавлять свойства или переопределять старые.

Упражнения

Векторный тип

Напишите конструктор Vector, представляющий вектор в двумерном пространстве. Он принимает параметры x и y (числа), которые хранятся в одноимённых свойствах.

Дайте прототипу Vector два метода, plus и minus, которые принимают другой вектор в качестве параметра и возвращают новый вектор, который хранит в x и y сумму или разность двух векторов (один this, второй – аргумент).

Добавьте геттер length в прототип, подсчитывающий длину вектора – расстояние от (0, 0) до (x, y).

```
// Ваш код

console.log(new Vector(1, 2).plus(new Vector(2, 3)));
// → Vector{x: 3, y: 5}
console.log(new Vector(1, 2).minus(new Vector(2, 3)));
// → Vector{x: -1, y: -1}
console.log(new Vector(3, 4).length);
// → 5
```

Ещё одна ячейка

Создайте тип ячейки StretchCell(inner, width, height), соответствующий интерфейсу ячеек таблицы из этой главы. Он должен оборачивать другую ячейку (как делает UnderlinedCell), и убеждаться, что результирующая ячейка имеет как минимум заданные ширину и высоту, даже если внутренняя ячейка – меньше.

```
// Ваш код.

var sc = new StretchCell(new TextCell("abc"), 1, 2);
console.log(sc.minWidth());
// → 3
console.log(sc.minHeight());
// → 2
console.log(sc.draw(3, 2));
// → ["abc", "   "]
```

Интерфейс к последовательностям

Разработайте интерфейс, абстрагирующий проход по набору значений. Объект с таким интерфейсом представляет собой последовательность, а интерфейс должен давать возможность в коде проходить по последовательности, работать со значениями, которые её составляют, и как-то сигнализировать о том, что мы достигли конца последовательности.

Задав интерфейс, попробуйте сделать функцию logFive, которая принимает объект-последовательность и вызывает console.log для первых её пяти элементов – или для меньшего количества, если их меньше пяти.

Затем создайте тип объекта ArraySeq, оборачивающий массив, и позволяющий проход по массиву с использованием разработанного вами интерфейса. Создайте другой тип объекта, RangeSeq, который проходит по диапазону чисел (его конструктор должен принимать аргументы from и to).

```
// Ваш код.

logFive(new ArraySeq([1, 2]));
// → 1
// → 2
logFive(new RangeSeq(100, 1000));
// → 100
// → 101
// → 102
// → 103
// → 104
```

Проект: электронная жизнь

Вопрос о том, могут ли машины думать, так же уместен, как вопрос о том, могут ли подводные лодки плавать.

Эдсгер Дейкстра, Угрозы вычислительной науке

В главах-проектах я перестану закидывать вас теорией, и буду работать вместе с вами над программами. Теория незаменима при обучении программированию, но она должна сопровождаться чтением и пониманием нетривиальных программ.

Наш проект – постройка виртуальной экосистемы, небольшого мира, населённого существами, которые двигаются и борются за выживание.

Определение

Чтобы задача стала выполнимой, мы кардинально упростим концепцию мира. А именно – мир будет двумерной сеткой, где каждая сущность занимает одну

клетку. На каждом ходу существа получат возможность выполнить какое-либо действие.

Таким образом, мы порубим время и пространство на единицы фиксированного размера: клетки для пространства и ходы для времени. Конечно, это грубое и неаккуратное приближение. Но наша симуляция должна быть развлекательной, а не аккуратной, поэтому мы свободно «срезаем углы».

Определить мир мы можем при помощи плана – массива строк, который раскладывает мировую сетку, используя один символ на клетку.

```
var plan = ["############################",
            "#      #    #      o      ##",
            "#                          #",
            "#          #####           #",
            "##         #   #    ##     #",
            "###           ##     #     #",
            "#           ###      #     #",
            "#   ####                   #",
            "#   ##       o             #",
            "# o  #         o       ### #",
            "#    #                     #",
            "############################"];
```

Символ “#” обозначает стены и камни, “o” – существо. Пробелы – пустое пространство.

План можно использовать для создания объекта мира. Он следит за размером и содержимым мира. У него есть метод toString, который преобразовывает мир в выводимую строчку (такую, как план, на котором он основан), чтобы мы могли наблюдать за происходящим внутри него. У объекта мир есть метод turn (ход), позволяющий всем существам сделать один ход и обновляющий состояние мира в соответствии с их действиями.

Изображаем пространство

У сетки, моделирующей мир, заданы ширина и высота. Клетки определяются координатами x и y. Мы используем простой тип Vector (из упражнений к предыдущей главе) для представления этих пар координат.

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
```

Потом нам нужен тип объекта, моделирующий саму сетку. Сетка – часть мира, но мы делаем из неё отдельный объект (который будет свойством мирового объекта), чтобы не усложнять мировой объект. Мир должен загружать себя вещами, относящимися к миру, а сетка – вещами, относящимися к сетке.

Для хранения сетки значений у нас есть несколько вариантов. Можно использовать массив из массивов-строк, и использовать двухступенчатый доступ к свойствам:

```
var grid = [["top left",    "top middle",    "top right"
],
            ["bottom left", "bottom middle", "bottom rig
ht"]];
console.log(grid[1][2]);
// → bottom right
```

Или мы можем взять один массив, размера width × height, и решить, что элемент (x, y) находится в позиции x + (y × width).

```
var grid = ["top left",    "top middle",    "top right",
            "bottom left", "bottom middle", "bottom righ
t"];
console.log(grid[2 + (1 * 3)]);
// → bottom right
```

Поскольку доступ будет завёрнут в методах объекта сетки, внешнему коду всё равно, какой подход будет выбран. Я выбрал второй, потому что с ним проще создавать массив. При вызове конструктора Array с одним числом в качестве аргумента он создаёт новый пустой массив заданной длины.

Следующий код объявляет объект Grid (сетка) с основными методами:

```
function Grid(width, height) {
  this.space = new Array(width * height);
  this.width = width;
  this.height = height;
}
Grid.prototype.isInside = function(vector) {
  return vector.x >= 0 && vector.x < this.width &&
         vector.y >= 0 && vector.y < this.height;
};
Grid.prototype.get = function(vector) {
  return this.space[vector.x + this.width * vector.y];
};
Grid.prototype.set = function(vector, value) {
  this.space[vector.x + this.width * vector.y] = value;
};
```

Элементарный тест:

```
var grid = new Grid(5, 5);
console.log(grid.get(new Vector(1, 1)));
// → undefined
grid.set(new Vector(1, 1), "X");
console.log(grid.get(new Vector(1, 1)));
// → X
```

Программный интерфейс существ

Перед тем, как заняться конструктором мира World, нам надо определиться с объектами существ, населяющих его. Я упомянул, что мир будет спрашивать существ, какие они хотят произвести действия. Работать это будет так: у каждого объекта существа есть метод act, который при вызове возвращает действие action. Action – объект типа property, который называет тип действия, которое хочет совершить существо, к примеру “move”. Action может содержать дополнительную информацию – такую, как направление движения.

Существа ужасно близоруки и видят только непосредственно прилегающие к ним клетки. Но и это может пригодиться при выборе действий. При вызове метода act ему даётся объект view, который позволяет существу изучить прилегающую местность. Мы называем

восемь соседних клеток их направлениями по компасу: “n” на север, “ne” на северо-восток, и т. п. Вот какой объект будет использоваться для преобразования из названий направлений в смещения по координатам:

```
var directions = {
  "n":  new Vector( 0, -1),
  "ne": new Vector( 1, -1),
  "e":  new Vector( 1,  0),
  "se": new Vector( 1,  1),
  "s":  new Vector( 0,  1),
  "sw": new Vector(-1,  1),
  "w":  new Vector(-1,  0),
  "nw": new Vector(-1, -1)
};
```

У объекта view есть метод look, который принимает направление и возвращает символ, к примеру "#", если там стена, или пробел, если там ничего нет. Объект также предоставляет удобные методы find и findAll. Оба принимают один из символов, представляющих вещи на карте, как аргумент. Первый возвращает направление, в котором этот предмет можно найти рядом с существом, или же null, если такого предмета рядом нет. Второй возвращает массив со всеми возможными направлениями, где найден такой предмет. Например, существо слева от стены (на западе) получит [«ne», «e», «se»] при вызове findAll с аргументом “#”.

Вот простое тупое существо, которое просто идёт, пока не врезается в препятствие, а затем отскакивает в случайном направлении.

```
function randomElement(array) {
  return array[Math.floor(Math.random() * array.length)]
;
}

function BouncingCritter() {
  this.direction = randomElement(Object.keys(directions)
);
};

BouncingCritter.prototype.act = function(view) {
  if (view.look(this.direction) != " ")
    this.direction = view.find(" ") || "s";
  return {type: "move", direction: this.direction};
};
```

Вспомогательная функция randomElement просто выбирает случайный элемент массива, используя Math.random и немного арифметики, чтобы получить случайный индекс. Мы и дальше будем использовать случайность, так как она – полезная штука в симуляциях.

Конструктор BouncingCritter вызывает Object.keys. Мы видели эту функцию в предыдущей главе – она возвращает массив со всеми именами свойств объекта.

Тут она получает все имена направлений из объекта directions, заданного ранее.

Конструкция `|| "s"` в методе act нужна, чтобы this.direction не получил null, в случае если существо забилось в угол без свободного пространства вокруг – например, окружено другими существами.

Мировой объект

Теперь можно приступать к мировому объекту World. Конструктор принимает план (массив строк, представляющих сетку мира) и объект legend. Это объект, сообщающий, что означает каждый из символов карты. В нём есть конструктор для каждого символа – кроме пробела, который ссылается на null (представляющий пустое пространство).

```
function elementFromChar(legend, ch) {
  if (ch == " ")
    return null;
  var element = new legend[ch]();
  element.originChar = ch;
  return element;
}

function World(map, legend) {
  var grid = new Grid(map[0].length, map.length);
  this.grid = grid;
  this.legend = legend;

  map.forEach(function(line, y) {
    for (var x = 0; x < line.length; x++)
      grid.set(new Vector(x, y),
               elementFromChar(legend, line[x]));
  });
}
```

В elementFromChar мы сначала создаём экземпляр нужного типа, находя конструктор символа и применяя к нему new. Потом добавляем свойство originChar, чтобы было просто выяснить, из какого символа элемент был создан изначально.

Нам понадобится это свойство originChar при изготовлении мирового метода toString. Метод строит карту в виде строки из текущего состояния мира, проходя двумерным циклом по клеткам сетки.

```
function charFromElement(element) {
  if (element == null)
    return " ";
  else
    return element.originChar;
}

World.prototype.toString = function() {
  var output = "";
  for (var y = 0; y < this.grid.height; y++) {
    for (var x = 0; x < this.grid.width; x++) {
      var element = this.grid.get(new Vector(x, y));
      output += charFromElement(element);
    }
    output += "\n";
  }
  return output;
};
```

Стена wall – простой объект. Используется для занятия места и не имеет метода act.

```
function Wall() {}
```

Проверяя объект World, создав экземпляр с использованием плана, заданного в начале главы, и затем вызвав его метод toString, мы получим очень похожую на этот план строку.

```
var world = new World(plan, {"#": Wall, "o": BouncingCri
tter});
console.log(world.toString());
// → ############################
//   #      #    #      o      ##
//   #                          #
//   #           #####          #
//   ##          #   #    ##    #
//   ###           ##     #     #
//   #            ###     #     #
//   #   ####                   #
//   #   ##       o             #
//   # o  #         o       ### #
//   #    #                     #
//   ############################
```

this и его область видимости

В конструкторе World есть вызов forEach. Хочу отметить, что внутри функции, передаваемой в forEach, мы уже не находимся непосредственно в области видимости конструктора. Каждый вызов функции получает своё пространство имён, поэтому this внутри неё уже не ссылается на создаваемый объект, на который ссылается this снаружи функции. И вообще, если функция вызывается не как метод, this будет относиться к глобальному объекту.

Значит, мы не можем писать this.grid для доступа к сетке изнутри цикла. Вместо этого внешняя функция создаёт локальную переменную grid, через которую внутренняя функция получает доступ к сетке.

Это промах в дизайне JavaScript. К счастью, в следующей версии есть решение этой проблемы. А пока есть пути обхода. Обычно пишут var self = this и после этого работают с переменной self.

Другое решение – использовать метод bind, который позволяет привязаться к конкретному объекту this.

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }.bind(this));
  }
};
console.log(test.addPropTo([5]));
// → [15]
```

Функция, передаваемая в map – результат привязки вызова, и посему её this привязан к первому аргументу, переданному в bind, то есть переменной this внешней функции (в которой содержится объект test).

Большинство стандартных методов высшего порядка у массивов, таких как forEach и map, принимают необязательный второй аргумент, который тоже можно использовать для передачи this при вызовах итерационной функции. Вы могли бы написать предыдущий пример чуть проще:

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }, this); // ← без bind
  }
};
console.log(test.addPropTo([5]));
// → [15]
```

Это работает только с теми функциями высшего порядка, у которых есть такой контекстный параметр. Если нет – приходится использовать другие упомянутые подходы.

В нашей собственной функции высшего порядка мы можем включить поддержку контекстного параметра, используя метод call для вызова функции, переданной в качестве аргумента. К примеру, вот вам метод forEach для нашего типа Grid, вызывающий заданную функцию для каждого элемента сетки, который не равен null или undefined:

```
Grid.prototype.forEach = function(f, context) {
  for (var y = 0; y < this.height; y++) {
    for (var x = 0; x < this.width; x++) {
      var value = this.space[x + y * this.width];
      if (value != null)
        f.call(context, value, new Vector(x, y));
    }
  }
};
```

Оживляем мир

Следующий шаг – создание метода turn (ход) для мирового объекта, дающего существам возможность действовать. Он будет обходить сетку методом forEach, и искать объекты, у которых есть метод act. Найдя объект, turn вызывает этот метод, получая объект action и производит это действие, если оно допустимо. Пока мы понимаем только действие “move”.

Есть одна возможная проблема. Догадаетесь, какая? Если мы позволим существам двигаться по мере того, как мы их перебираем, они могут перейти на клетку, которую мы ещё не обработали, и тогда мы позволим им сдвинуться ещё раз, когда очередь дойдёт до этой клетки.

Таким образом, нам надо хранить массив существ, которые уже сделали свой шаг, и игнорировать их при повторном проходе.

```
World.prototype.turn = function() {
  var acted = [];
  this.grid.forEach(function(critter, vector) {
    if (critter.act && acted.indexOf(critter) == -1) {
      acted.push(critter);
      this.letAct(critter, vector);
    }
  }, this);
};
```

Второй параметр метода forEach используется для доступа к правильной переменной this во внутренней функции. Метод letAct содержит логику, которая позволяет существам двигаться.

```
World.prototype.letAct = function(critter, vector) {
  var action = critter.act(new View(this, vector));
  if (action && action.type == "move") {
    var dest = this.checkDestination(action, vector);
    if (dest && this.grid.get(dest) == null) {
      this.grid.set(vector, null);
      this.grid.set(dest, critter);
    }
  }
};

World.prototype.checkDestination = function(action, vect
or) {
  if (directions.hasOwnProperty(action.direction)) {
    var dest = vector.plus(directions[action.direction])
;
    if (this.grid.isInside(dest))
      return dest;
  }
};
```

Сначала мы просто просим существо действовать, передавая ему объект view, который знает про мир и текущее положение существа в мире (мы скоро зададим View). Метод act возвращает какое-либо действие.

Если тип действия не “move”, оно игнорируется. Если “move”, и если у него есть свойство direction, ссылающееся на допустимое направление, и если клетка

в этом направлении пустует (null), мы назначаем клетке, где только что было существо, null, и сохраняем существо в клетке назначения.

Заметьте, что letAct заботится об игнорировании неправильных входных данных. Он не предполагает по умолчанию, что направление допустимо, или, что свойство типа имеет смысл. Такого рода защитное программирование в некоторых ситуациях имеет смысл. В основном это делается для проверки входных данных, приходящих от источников, которые вы не контролируете (ввод пользователя или чтение файла), но оно также полезно для изолирования подсистем друг от друга. В нашем случае его цель – учесть, что существа могут быть запрограммированы неаккуратно. Им не надо проверять, имеют ли их намерения смысл. Они просто запрашивают возможность действия, а мир сам решает, разрешать ли его.

Эти два метода не принадлежат к внешнему интерфейсу мирового объекта. Они являются деталями внутренней реализации. Некоторые языки предусматривают способы объявлять определённые методы и свойства «приватными», и выдавать ошибку при попытке их использования снаружи объекта. JavaScript не предусматривает такого, так что вам придётся полагаться на другие способы сообщить о том, что является частью

интерфейса объекта. Иногда помогает использование схемы именования свойств для различения внутренних и внешних, например, с особыми приставками к именам внутренних, типа подчёркивания (_). Это облегчит выявление случайного использования свойств, не являющихся частью интерфейса.

А пропущенная часть, тип View, выглядит следующим образом:

```
function View(world, vector) {
  this.world = world;
  this.vector = vector;
}
View.prototype.look = function(dir) {
  var target = this.vector.plus(directions[dir]);
  if (this.world.grid.isInside(target))
    return charFromElement(this.world.grid.get(target));
  else
    return "#";
};
View.prototype.findAll = function(ch) {
  var found = [];
  for (var dir in directions)
    if (this.look(dir) == ch)
      found.push(dir);
  return found;
};
View.prototype.find = function(ch) {
  var found = this.findAll(ch);
  if (found.length == 0) return null;
  return randomElement(found);
};
```

Метод look вычисляет координаты, на которые мы пытаемся посмотреть. Если они находятся внутри сетки, то получает символ, соответствующий элементу, находящемуся там. Для координат снаружи сетки look просто притворяется, что там стена – если вы зададите мир без окружающих стен, существа не смогут сойти с края.

Оно двигается

Мы создали экземпляр мирового объекта. Теперь, когда все необходимые методы готовы, у нас должно получиться заставить его двигаться.

```
for (var i = 0; i < 5; i++) {
  world.turn();
  console.log(world.toString());
}
// → … пять ходов
```

Просто выводить пять копий карты – не очень удобный способ наблюдения за миром. Поэтому в песочнице для книги (или в файлах для скачивания) есть волшебная функция animateWorld, которая показывает мир как анимацию на экране, делая по три шага в секунду, пока вы не нажмёте стоп.

```
animateWorld(world);
// → … заработало!
```

Реализация animateWorld пока останется тайной, но после прочтения следующих глав книги, обсуждающих интеграцию JavaScript в браузеры, она уже не будет выглядеть так загадочно.

Больше форм жизни

Одна из интересных ситуаций, происходящих в мире, случается, когда два существа отскакивают друг от друга. Можете придумать другую интересную форму взаимодействий?

Я придумал существо, двигающееся по стенке. Оно держит свою левую руку (лапу, щупальце, что угодно) на стене и двигается вдоль неё. Это, как оказалось, не так-то просто запрограммировать.

Нам нужно будет вычислять, используя направления в пространстве. Так как направления заданы набором строк, нам надо задать свою операцию dirPlus для подсчёта относительных направлений. dirPlus("n", 1) означает поворот по часовой на 45 градусов на север, что приводит к “ne”. dirPlus("s", -2) означает поворот против часовой с юга, то есть на восток.

```
var directionNames = Object.keys(directions);
function dirPlus(dir, n) {
  var index = directionNames.indexOf(dir);
  return directionNames[(index + n + 8) % 8];
}

function WallFollower() {
  this.dir = "s";
}

WallFollower.prototype.act = function(view) {
  var start = this.dir;
  if (view.look(dirPlus(this.dir, -3)) != " ")
    start = this.dir = dirPlus(this.dir, -2);
  while (view.look(this.dir) != " ") {
    this.dir = dirPlus(this.dir, 1);
    if (this.dir == start) break;
  }
  return {type: "move", direction: this.dir};
};
```

Метод act только сканирует окружение существа, начиная с левой стороны и дальше по часовой, пока не находит пустую клетку. Затем он двигается в направлении этой клетки.

Усложняет ситуацию то, что существо может оказаться вдали от стен на пустом пространстве — либо обходя другое существо, либо изначально оказавшись там. Если мы оставим описанный алгоритм, несчастное существо будет каждый ход поворачивать налево, и бегать по кругу.

Так что есть ещё одна проверка через if, что сканирование нужно начинать, если существо только что прошло мимо какого-либо препятствия. То есть, если пространство сзади и слева не пустое. В противном случае сканировать начинаем впереди, поэтому в пустом пространстве он будет идти прямо.

И наконец, есть проверка на совпадение this.dir и start на каждом проходе цикла, чтобы он не зациклился, когда существу некуда идти из-за стен или других существ, и оно не может найти пустую клетку.

Этот небольшой мир показывает существ, двигающихся по стенам.:

```
animateWorld(new World(
  ["############",
   "#      #    #",
   "#   ~     ~ #",
   "#  ##       #",
   "#  ##  o####",
   "#           #",
   "############"],
  {"#": Wall,
   "~": WallFollower,
   "o": BouncingCritter}
));
```

Более жизненная ситуация

Чтобы сделать жизнь в нашем мирке более интересной, добавим понятия еды и размножения. У каждого живого существа появляется новое свойство, energy (энергия), которая уменьшается при совершении действий, и увеличивается при поедании еды. Когда у существа достаточно энергии, он может размножаться, создавая новое существо того же типа. Для упрощения расчётов наши существа размножаются сами по себе.

Если существа только двигаются и едят друг друга, мир вскоре поддастся возрастающей энтропии, в нём закончится энергия и он превратится в пустыню. Для предотвращения этого финала (или оттягивания), мы добавляем в него растения. Они не двигаются. Они просто занимаются фотосинтезом и растут (нарабатывают энергию), и размножаются.

Чтобы это заработало, нам нужен мир с другим методом letAct. Мы могли бы просто заменить метод прототипа World, но я привык к нашей симуляции ходящих по стенам существ и не хотел бы её разрушать.

Одно из решений – использовать наследование. Мы создаём новый конструктор, LifelikeWorld, чей прототип основан на прототипе World, но переопределяет метод letAct. Новый letAct передаёт работу по совершению действий в разные функции, хранящиеся в объекте actionTypes.

```
function LifelikeWorld(map, legend) {
  World.call(this, map, legend);
}
LifelikeWorld.prototype = Object.create(World.prototype)
;

var actionTypes = Object.create(null);

LifelikeWorld.prototype.letAct = function(critter, vector
) {
  var action = critter.act(new View(this, vector));
  var handled = action &&
    action.type in actionTypes &&
    actionTypes[action.type].call(this, critter,
                                  vector, action);
  if (!handled) {
    critter.energy -= 0.2;
    if (critter.energy <= 0)
      this.grid.set(vector, null);
  }
};
```

Новый метод letAct проверяет, было ли передано хоть какое-то действие, затем – есть ли функция, обрабатывающая его, и в конце – возвращает ли эта функция true, показывая, что действие выполнено успешно. Обратите внимание на использование call, чтобы дать функции доступ к мировому объекту через this.

Если действие по какой-либо причине не сработало, действием по умолчанию для существа будет ожидание. Он теряет 0.2 единицы энергии, а когда его уровень энергии падает ниже нуля, он умирает и исчезает с сетки.

Обработчики действий

Самое простое действие – рост, его используют растения. Когда возвращается объект action типа {type: "grow"}, будет вызван следующий метод-обработчик:

```
actionTypes.grow = function(critter) {
  critter.energy += 0.5;
  return true;
};
```

Рост всегда успешен и добавляет половину единицы к энергетическому уровню растения.

Движение получается более сложным.

```
actionTypes.move = function(critter, vector, action) {
  var dest = this.checkDestination(action, vector);
  if (dest == null ||
      critter.energy <= 1 ||
      this.grid.get(dest) != null)
    return false;
  critter.energy -= 1;
  this.grid.set(vector, null);
  this.grid.set(dest, critter);
  return true;
};
```

Это действие вначале проверяет, используя метод checkDestination, объявленный ранее, предоставляет ли действие допустимое направление. Если нет, или же в том направлении не пустой участок, или же у существа недостаёт энергии – move возвращает false, показывая, что действие не состоялось. В ином случае он двигает существо и вычитает энергию.

Кроме движения, существа могут есть.

```
actionTypes.eat = function(critter, vector, action) {
  var dest = this.checkDestination(action, vector);
  var atDest = dest != null && this.grid.get(dest);
  if (!atDest || atDest.energy == null)
    return false;
  critter.energy += atDest.energy;
  this.grid.set(dest, null);
  return true;
};
```

Поедание другого существа также требует предоставления допустимой клетки направления. В этом случае клетка должна содержать что-либо с энергией, например существо (но не стену, их есть нельзя). Если это подтверждается, энергия съеденного переходит к едоку, а жертва удаляется с сетки.

И наконец, мы позволяем существам размножаться.

```
actionTypes.reproduce = function(critter, vector, action
) {
  var baby = elementFromChar(this.legend,
                             critter.originChar);
  var dest = this.checkDestination(action, vector);
  if (dest == null ||
      critter.energy <= 2 * baby.energy ||
      this.grid.get(dest) != null)
    return false;
  critter.energy -= 2 * baby.energy;
  this.grid.set(dest, baby);
  return true;
};
```

Размножение отнимает в два раза больше энергии, чем есть у новорожденного. Поэтому мы создаём гипотетического отпрыска, используя elementFromChar на оригинальном существе. Как только у нас есть отпрыск, мы можем выяснить его энергетический уровень и

проверить, есть ли у родителя достаточно энергии, чтобы родить его. Также нам потребуется допустимая клетка направления.

Если всё в порядке, отпрыск помещается на сетку (и перестаёт быть гипотетическим), а энергия тратится.

Населяем мир

Теперь у нас есть основа для симуляции существ, больше похожих на настоящие. Мы могли бы поместить в новый мир существ из старого, но они бы просто умерли, так как у них нет свойства energy. Давайте сделаем новых. Сначала напишем растение, которое, по сути, довольно простая форма жизни.

```
function Plant() {
  this.energy = 3 + Math.random() * 4;
}
Plant.prototype.act = function(context) {
  if (this.energy > 15) {
    var space = context.find(" ");
    if (space)
      return {type: "reproduce", direction: space};
  }
  if (this.energy < 20)
    return {type: "grow"};
};
```

Растения начинают со случайного уровня энергии от 3 до 7, чтобы они не размножались все в один ход. Когда растение достигает энергии 15, а рядом есть пустая клетка – оно размножается в неё. Если оно не может размножиться, то просто растёт, пока не достигнет энергии 20.

Теперь определим поедателя растений.

```
function PlantEater() {
  this.energy = 20;
}
PlantEater.prototype.act = function(context) {
  var space = context.find(" ");
  if (this.energy > 60 && space)
    return {type: "reproduce", direction: space};
  var plant = context.find("*");
  if (plant)
    return {type: "eat", direction: plant};
  if (space)
    return {type: "move", direction: space};
};
```

Для растений будем использовать символ * — то, что будет искать существо в поисках еды.

Вдохнём жизнь

И теперь у нас есть достаточно элементов для нового мира. Представьте следующую карту как травянистую долину, где пасётся стадо травоядных, лежат несколько валунов и цветёт буйная растительность.

```
var valley = new LifelikeWorld(
  ["############################",
   "#####                 ######",
   "##   ***                **##",
   "#   *##**         **  O  *##",
   "#    ***     O    ##**    *#",
   "#       O         ##***    #",
   "#                 ##**     #",
   "#   O       #*             #",
   "#*          #**       O    #",
   "#***        ##**    O    **#",
   "##****     ###***       *###",
   "############################"],
  {"#": Wall,
   "O": PlantEater,
   "*": Plant}
);
```

Большую часть времени растения размножаются и разрастаются, но затем изобилие еды приводит к взрывному росту популяции травоядных, которые съедают почти всю растительность, что приводит к массовому вымиранию от голода. Иногда экосистема восстанавливается и начинается новый цикл. В других случаях какой-то из видов вымирает. Если травоядные,

тогда всё пространство заполняется растениями. Если растения – оставшиеся существа умирают от голода, и долина превращается в необитаемую пустошь. О, жестокость природы…

Упражнения

Искусственный идиот

Грустно, когда жители нашего мира вымирают за несколько минут. Чтобы справиться с этим, мы можем попробовать создать более умного поедателя растений.

У наших травоядных есть несколько очевидных проблем. Во-первых, они жадные – поедают каждое растение, которое находят, пока полностью не уничтожат всю растительность. Во-вторых, их случайное движение (вспомните, что метод view.find возвращает случайное направление) заставляет их болтаться неэффективно и помирать с голоду, если рядом не окажется растений. И наконец, они слишком быстро размножаются, что делает циклы от изобилия к голоду слишком быстрыми.

Напишите новый тип существа, который старается справится с одним или несколькими проблемами и замените им старый тип PlantEater в мире долины. Последите за ними. Выполните необходимые подстройки.

```
// Ваш код
function SmartPlantEater() {}

animateWorld(new LifelikeWorld(
  ["############################",
   "#####                 ######",
   "##   ***                **##",
   "#   *##**         **  O  *##",
   "#    ***     O    ##**    *#",
   "#       O         ##***    #",
   "#                 ##**     #",
   "#   O       #*             #",
   "#*          #**       O    #",
   "#***        ##**    O    **#",
   "##****     ###***       *###",
   "############################"],
  {"#": Wall,
   "O": SmartPlantEater,
   "*": Plant}
));
```

Хищники

В любой серьёзной экосистеме пищевая цепочка длиннее одного звена. Напишите ещё одно существо, которое выживает, поедая травоядных. Вы заметите, что стабильности ещё труднее достичь, когда циклы происходят на разных уровнях. Попытайтесь найти стратегию, которая позволит экосистеме работать плавно некоторое время.

Увеличение мира может помочь в этом. Тогда локальные демографические взрывы или уменьшение численности имеют меньше шансов полностью изничтожить популяцию, и есть место для относительно большой популяции жертв, которая может поддерживать небольшую популяцию хищников.

```
// Ваш код тут
function Tiger() {}

animateWorld(new LifelikeWorld(
  ["#######################################################"
,
   "#                  ####         ****              ###"
,
   "#   *  @  ##                 ########       OO    ##"
,
   "#   *    ##        O O                 ****       *#"
,
   "#       ##*                        ##########     *#"
,
   "#      ##***  *         ****                    **#"
,
   "#* **  #  *  ***      ########                   **#"
,
   "#* **  #      *               #   *              **#"
,
   "#     ##              #   O   #  ***          ######"
,
   "#*            @       #       #   *        O  #    #"
,
   "#*                    #  ######                 ** #"
,
```

```
    "###           ****           ***                  ** #"
  ,
    "#       O                        @         O       #"
  ,
    "#   *     ##  ##  ##  ##               ###      *  #"
  ,
    "#   **         #              *       #####  O     #"
  ,
    "##  **  O   O  #  #    ***  ***        ###      ** #"
  ,
    "###               #   *****                   ****#"
  ,
    "##################################################"
  ],
    {"#": Wall,
     "@": Tiger,
     "O": SmartPlantEater, // из предыдущего упражнения
     "*": Plant}
  ));
```

Поиск и обработка ошибок

Отладка изначально вдвое сложнее написания кода. Поэтому, если вы пишете код настолько заумный, насколько можете, то по определению вы не способны отлаживать его.

Брайан Керниган и П.Ж.Плауэр, «Основы программного стиля»

Юан-Ма написал небольшую программу, использующую много глобальных переменных и ужасных хаков. Ученик, читая программу, спросил его: «Вы предупреждали нас о подобных техниках, но при этом я нахожу их в вашей же программе. Как это возможно?» Мастер ответил: «Не нужно бежать за поливальным шлангом, если дом не горит».

Мастер Юан-Ма, «Книга программирования».

Программа – это кристаллизованная мысль. Иногда мысли путаются. Иногда при превращении мыслей в программу в код вкрадываются ошибки. В обоих случаях получается повреждённая программа.

Недостатки в программах обычно называют ошибками. Это могут быть ошибки программиста или проблемы в системах, с которыми программа взаимодействует. Некоторые ошибки очевидны, другие – трудноуловимы и могут скрываться в системах годами.

Часто проблема возникает в тех ситуациях, возникновение которых программист изначально не предвидел. Иногда этих ситуаций нельзя избежать. Когда пользователя просят ввести его возраст, а он вводит «апельсин», это ставит программу в непростую ситуацию. Эти ситуации необходимо предвидеть и как-то обрабатывать.

Ошибки программистов

В случае ошибок программистов наша цель ясна. Нам надо найти их и исправить. Таковые ошибки варьируются от простых опечаток, на которые компьютер пожалуется сразу же, как только увидит программу, до скрытых ошибок в нашем понимании того, как программа работает, которые приводят к неправильным результатам в особых случаях. Ошибки последнего рода можно искать неделями.

Разные языки по-разному могут помогать вам в поиске ошибок. К сожалению, JavaScript находится на конце этой шкалы, обозначенном как «вообще почти не помогает». Некоторым языкам надо точно знать типы всех переменных и выражений ещё до запуска программы, и они сразу сообщат вам, если типы использованы некорректно. JavaScript рассматривает типы только во время исполнения программ, и даже тогда он разрешает делать не очень осмысленные вещи без всяких жалоб, например

```
x = true * "обезьяна"
```

На некоторые вещи JavaScript всё-таки жалуется. Написание синтаксически неправильной программы сразу вызовет ошибку. Другие ошибки, например вызов чего-либо, не являющегося функцией, или обращение к свойству неопределённой переменной, возникнут при выполнении программы, когда она сталкивается с такой бессмысленной ситуацией.

Но часто ваши бессмысленные вычисления просто породят NaN (not a number) или undefined. Программа радостно продолжит, будучи уверенной в том, что она делает что-то осмысленное. Ошибка проявит себя позже, когда такое фиктивное значение уже пройдёт через несколько функций. Она может вообще не вызвать

сообщение об ошибке, а просто привести к неправильному результату выполнения. Поиск источника таких проблем – сложная задача.

Процесс поиска ошибок (bugs) в программах называется отладкой (debugging).

Строгий режим (strict mode)

JavaScript можно заставить быть построже, переведя его в строгий режим. Для этого наверху файла или тела функции пишется "use strict". Пример:

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++)
    console.log("Всё будет офигенно");
}

canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

Обычно, когда ты забываешь написать var перед переменной, как в примере перед counter, JavaScript по-тихому создаёт глобальную переменную и использует её. В строгом режиме выдаётся ошибка. Это очень удобно.

Однако, ошибка не выдаётся, когда глобальная переменная уже существует – только тогда, когда присваивание создаёт новую переменную.

Ещё одно изменение – привязка this содержит undefined в тех функциях, которые вызывали не как методы. Когда мы вызываем функцию не в строгом режиме, this ссылается на объект глобальной области видимости. Поэтому если вы случайно неправильно вызовете метод в строгом режиме, JavaScript выдаст ошибку, если попытается прочесть что-то из this, а не будет радостно работать с глобальным объектом.

К примеру, рассмотрим код, вызывающий конструктор без ключевого слова new, в случае чего this не будет ссылаться на создаваемый объект.

```
function Person(name) { this.name = name; }
var ferdinand = Person("Евлампий"); // ой-вэй
console.log(name);
// → Евлампий
```

Некорректный вызов Person успешно происходит, но возвращается как undefined и создаёт глобальную переменную name. В строгом режиме всё по-другому:

```
"use strict";
function Person(name) { this.name = name; }
// Опаньки, мы ж забыли 'new'
var ferdinand = Person("Евлампий");
// → TypeError: Cannot set property 'name' of undefined
```

Нам сразу сообщают об ошибке. Очень удобно.

Строгий режим умеет ещё кое-что. Он запрещает вызывать функцию с несколькими параметрами с одним и тем же именем, и удаляет некоторые потенциально проблемные свойства языка (например, инструкцию with, которая настолько ужасна, что даже не обсуждается в этой книге).

Короче говоря, надпись "use strict" перед текстом программы редко причиняет проблемы, зато помогает вам видеть их.

Тестирование

Если язык не собирается помогать нам в поиске ошибок, приходится искать их сложным способом: запуская программу и наблюдая, делает ли она что-то так, как надо.

Делать это вручную, снова и снова – верный способ сойти с ума. К счастью, часто возможно написать другую программу, которая автоматизирует проверку вашей основной программы.

Для примера вновь обратимся к типу Vector.

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
```

Мы напишем программу, которая проверит, что наша реализация Vector работает, как нужно. Затем после каждого изменения реализации мы будем запускать проверочную программу, чтобы убедиться, что мы ничего не сломали. Когда мы добавим функциональности (к примеру, новый метод) к типу Vector, мы добавим проверок этой новой функциональности.

```
function testVector() {
  var p1 = new Vector(10, 20);
  var p2 = new Vector(-10, 5);
  var p3 = p1.plus(p2);

  if (p1.x !== 10) return "облом: значение x не то";
  if (p1.y !== 20) return "облом: значение y не то";
  if (p2.x !== -10) return "облом: отрицательное значени
e x не то";
  if (p3.x !== 0) return "облом: результат сложения x не
 тот";
  if (p3.y !== 25) return "облом: результат сложения y н
e тот";
  return "всё пучком";
}
console.log(testVector());
// → всё пучком
```

Написание таких проверок приводит к появлению повторяющегося кода. К счастью, есть программные продукты, помогающие писать наборы проверок при помощи специального языка, приспособленного именно для написания проверок. Их называют testing frameworks.

Отладка (debugging)

Когда вы заметили проблему в программе (она ведёт себя неправильно и выдаёт ошибки), самое время выяснить, в чём проблема.

Иногда это очевидно. Сообщение об ошибке наводит вас на конкретную строку программы, и если вы прочтёте описание ошибки и эту строку, вы часто сможете найти проблему.

Но не всегда. Иногда строчка, приводящая к ошибке, просто оказывается первым местом, где некорректное значение, полученное где-то ещё, используется неправильно. Иногда вообще нет сообщения об ошибке – есть просто неверный результат. Если вы делали упражнения из предыдущих глав, вы наверняка попадали в такие ситуации.

Следующий пример пробует преобразовать число заданной системы счисления в строку, отнимая последнюю цифру и совершая деление, чтобы избавиться от этой цифры. Но дикий результат, выдаваемый программой, как бы намекает на присутствие в ней ошибки.

```
function numberToString(n, base) {
  var result = "", sign = "";
  if (n < 0) {
    sign = "-";
    n = -n;
  }
  do {
    result = String(n % base) + result;
    n /= base;
  } while (n > 0);
  return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e-3181.3…
```

Даже если вы нашли проблему – притворитесь, что ещё не нашли. Мы знаем, что программа сбоит, и нам нужно узнать, почему.

Здесь вам надо преодолеть желание начать вносить случайные изменения в код. Вместо этого подумайте. Проанализируйте результат и придумайте теорию, по которой это происходит. Проведите дополнительные наблюдения для проверки теории, а если теории нет – проведите наблюдения, которые бы помогли вам изобрести её.

Размещение нескольких вызовов console.log в стратегических местах – хороший способ получить дополнительную информацию о том, что программа делает. В нашем случае нам нужно, чтобы n принимала значения 13, 1, затем 0. Давайте выведем значения в начале цикла:

```
13
1.3
0.13
0.013
…
1.5e-323
```

Н-да. Деление 13 на 10 выдаёт не целое число. Вместо n /= base нам нужно n = Math.floor(n / base), тогда число будет корректно «сдвинуто» вправо.

Кроме console.log можно воспользоваться отладчиком в браузере. Современные браузеры умеют ставить точку остановки на выбранной строчке кода. Это приведёт к приостановке выполнения программы каждый раз, когда будет достигнута выбранная строчка, и тогда вы сможете просмотреть содержимое переменных. Не буду подробно расписывать процесс, поскольку у разных браузеров он организован по-разному – поищите в вашем браузере “developer tools”, инструменты разработчика. Ещё один способ установить точку остановки – включить в код

инструкцию для отладчика, состоящую из ключевого слова debugger. Если инструменты разработчика активны, исполнение программы будет приостановлено на этой инструкции, и вы сможете изучить состояние программы.

Распространение ошибок

К сожалению, программист может предотвратить появление не всех проблем. Если ваша программа общается с внешним миром, она может получить неправильные входные данные, или же системы, с которыми она пытается взаимодействовать, окажутся сломанными или недоступными.

Простые программы, или программы, работающие под вашим надзором, могут просто «сдаваться» в такой момент. Вы можете изучить проблему и попробовать снова. «Настоящие» приложения не должны просто «падать». Иногда приходится принимать неправильные входные данные и как-то с ними работать. В других случаях нужно сообщить пользователю, что что-то пошло не так, и потом уже сдаваться. В любом случае программа должна что-то сделать в ответ на возникновение проблемы.

Допустим, у вас есть функция promptInteger, которая запрашивает целое число и возвращает его. Что она должна сделать, если пользователь введёт «апельсин»?

Один из вариантов – вернуть особое значение. Обычно для этих целей используют null и undefined.

```
function promptNumber(question) {
  var result = Number(prompt(question, ""));
  if (isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("Сколько пальцев видите?"));
```

Это надёжная стратегия. Теперь любой код, вызывающий promptNumber, должен проверять, было ли возвращено число, и если нет, как-то выйти из ситуации – спросить снова, или задать значение по-умолчанию. Или вернуть специальное значение уже тому, кто его вызвал, сообщая о неудаче.

Во многих таких случаях, когда ошибки возникают часто и вызывающий функцию код должен принимать их во внимание, совершенно допустимо возвращать специальное значение как индикатор ошибки. Но есть и минусы. Во-первых, что, если функция и так может

вернуть любой тип значения? Для неё сложно найти специальное значение, которое будет отличаться от допустимого результата.

Вторая проблема – работа со специальными значениями может замусорить код. Если функция promptNumber вызывается 10 раз, то надо 10 раз проверить, не вернула ли она null. Если реакция на null заключается в возврате null на уровень выше, тогда там, где вызывался этот код, тоже нужно встраивать проверку на null, и так далее.

Исключения

Когда функция не может работать нормально, мы бы хотели остановить работу и перепрыгнуть туда, где такая ошибка может быть обработана. Этим занимается обработка исключений.

Код, встретивший проблему в момент выполнения, может поднять (или выкинуть) исключение (raise exception, throw exception), которое представляет из себя некое значение. Возврат исключения напоминает некий «прокачанный» возврат из функции – он выпрыгивает не только из самой функции, но и из всех вызывавших её функций, до того места, с которого началось выполнение. Это называется развёртыванием стека (unwinding the stack). Может быть,

вы помните стек функций из главы 3… Исключение быстро проматывает стек вниз, выкидывая все контексты вызовов, которые встречает.

Если бы исключения сразу доходили до самого низа стека, пользы от них было бы немного. Они бы просто предоставляли интересный способ взорвать программу. Их сила в том, что на их пути в стеке можно поставить «препятствия», которые будут ловить исключения, мчащиеся по стеку. И тогда с этим можно сделать что-то полезное, после чего программа продолжает выполняться с той точки, где было поймано исключение.

Пример:

```
function promptDirection(question) {
  var result = prompt(question, "");
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Недопустимое направление: " + result)
;
}

function look() {
  if (promptDirection("Куда?") == "L")
    return "дом";
  else
    return "двух разъярённых медведей";
}

try {
  console.log("Вы видите", look());
} catch (error) {
  console.log("Что-то не так: " + error);
}
```

Ключевое слово throw используется для выбрасывания исключения. Ловлей занимается кусок кода, обёрнутый в блок try, за которым следует catch. Когда код в блоке try выкидывает исключение, выполняется блок catch. Переменная, указанная в скобках, будет привязана к значению исключения. После завершения выполнения блока catch, или же если блок try выполняется без проблем, выполнение переходит к коду, лежащему после инструкции try/catch.

В данном случае для создания исключения мы использовали конструктор Error. Это стандартный конструктор, создающий объект со свойством message. В современных окружениях JavaScript экземпляры этого конструктора также собирают информацию о стеке вызовов, который был накоплен в момент выкидывания исключения – так называемое отслеживание стека (stack trace). Эта информация сохраняется в свойстве stack, и может помочь при разборе проблемы – она сообщает, в какой функции случилась проблема и какие другие функции привели к данному вызову.

Обратите внимание, что функция look полностью игнорирует возможность возникновения проблем в promptDirection. Это преимущество исключений – код, обрабатывающий ошибки, нужен только в том месте, где происходит ошибка, и там, где она обрабатывается. Промежуточные функции просто не обращают на это внимания.

Ну, почти.

Подчищаем за исключениями

Представьте следующую ситуацию: функция withContext желает удостовериться, что во время её выполнения переменная верхнего уровня context содержит

специальное значение контекста. В конце выполнения функция восстанавливает прежнее значение переменной.

```
var context = null;

function withContext(newContext, body) {
  var oldContext = context;
  context = newContext;
  var result = body();
  context = oldContext;
  return result;
}
```

Что, если функция body выбросит исключение? В таком случае вызов withContext будет выброшен исключением из стека, и переменной context никогда не будет возвращено первоначальное значение.

Но у инструкции try есть ещё одна особенность. За ней может следовать блок finally, либо вместо catch, либо вместе с catch. Блок finally означает «выполнить код в любом случае после выполнения блока try». Если функции надо что-то подчистить, то подчищающий код нужно включать в блок finally.

```
function withContext(newContext, body) {
  var oldContext = context;
  context = newContext;
  try {
    return body();
  } finally {
    context = oldContext;
  }
}
```

Заметьте, что нам больше не нужно сохранять результат вызова body в отдельной переменной, чтобы вернуть его. Даже если мы возвращаемся из блока try, блок finally всё равно будет выполнен. Теперь мы можем безопасно сделать так:

```
try {
  withContext(5, function() {
    if (context < 10)
      throw new Error("Контекст слишком мал!");
  });
} catch (e) {
  console.log("Игнорируем: " + e);
}
// → Игнорируем: Error: Контекст слишком мал!

console.log(context);
// → null
```

Несмотря на то, что вызываемая из withContext функция «сломалась», сам по себе withContext по-прежнему подчищает значение переменной context.

Выборочный отлов исключений

Когда исключение доходит до низа стека и его никто не поймал, его обрабатывает окружение. Как именно – зависит от конкретного окружения. В браузерах описание ошибки выдаётся в консоль (она обычно доступна в меню «Инструменты» или «Разработка»).

Если речь идёт об ошибках или проблемах, которые программа не может обработать в принципе, допустимо просто пропустить такую ошибку. Необработанное исключение – разумный способ сообщить о проблеме в программе, и консоль в современных браузерах выдаст вам необходимую информацию о том, какие вызовы функций были в стеке в момент возникновения проблемы.

Если возникновение проблемы предсказуемо, программа не должна падать с необработанным исключением – это не очень дружественно по отношению к пользователю.

Недопустимое использование языка – ссылки на несуществующую переменную, запрос свойств у переменной, равной null, или вызов чего-то, что не является функцией – тоже приводит к выбрасыванию исключений. Такие исключения можно отлавливать точно так же, как свои собственные.

При входе в блок catch мы знаем только, что что-то внутри блока try привело к исключению. Мы не знаем, что именно, и какое исключение произошло.

JavaScript (что является вопиющим упущением) не предоставляет непосредственной поддержки выборочного отлова исключений: либо ловим все, либо никакие. Из-за этого люди часто предполагают, что случившееся исключение – именно то, ради которого и писался блок catch.

Но может быть и по-другому. Нарушение произошло где-то ещё, или в программу вкралась ошибка. Вот пример, где мы пробуем вызывать promptDirection до тех пор, пока не получим допустимый ответ:

```
for (;;) {
  try {
    var dir = promtDirection("Куда?"); // ← опечатка!
    console.log("Ваш выбор", dir);
    break;
  } catch (e) {
    console.log("Недопустимое направление. Попробуйте ещ
ё раз.");
  }
}
```

Конструкция for (;;) – способ устроить бесконечный цикл. Мы вываливаемся из него, только когда получаем допустимое направление. Но мы неправильно написали название promptDirection, что приводит к ошибке “undefined variable”. А так как блок catch игнорирует значение исключения e, предполагая, что он разбирается с другой проблемой, он считает, что выброшенное исключение является результатом неправильных входных данных. Это приводит к бесконечному циклу и скрывает полезное сообщение об ошибке насчёт неправильного имени переменной.

Как правило, не стоит так ловить исключения, если только у вас нет цели перенаправить их куда-либо – к примеру, по сети, чтобы сообщить другой системе о падении нашей программы. И даже тогда внимательно смотрите, не будет ли скрыта важная информация.

Значит, нам надо поймать определённое исключение. Мы можем в блоке catch проверять, является ли случившееся исключение интересующим нас исключением, а в противном случае заново выбрасывать его. Но как нам распознать исключение?

Конечно, мы могли бы сравнить свойство message с сообщением об ошибке, которую мы ждём. Но это ненадёжный способ писать код – использовать информацию, предназначающуюся для человека (сообщение), чтобы принять программное решение. Как только кто-нибудь поменяет или переведёт это сообщение, код перестанет работать.

Давайте лучше определим новый тип ошибки и используем instanceof для его распознавания.

```
function InputError(message) {
  this.message = message;
  this.stack = (new Error()).stack;
}
InputError.prototype = Object.create(Error.prototype);
InputError.prototype.name = "InputError";
```

Прототип наследуется от Error.prototype, поэтому instanceof Error тоже будет выполняться для объектов типа InputError. И ему назначено свойство name, как и

другим стандартным типам ошибок (Error, SyntaxError, ReferenceError, и т.п.)

Присвоение свойству stack пытается передать этому объекту отслеживание стека, на тех платформах, которые это поддерживают, путём создания объекта Error и использования его стека.

Теперь promptDirection может сотворить такую ошибку.

```
function promptDirection(question) {
  var result = prompt(question, "");
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}

А в цикле её будет ловить сподручнее.

for (;;) {
  try {
    var dir = promptDirection("Куда?");
    console.log("Ваш выбор", dir);
    break;
  } catch (e) {
    if (e instanceof InputError)
      console.log("Недопустимое направление. Попробуйте
ещё раз.");
    else
      throw e;
  }
}
```

Код отлавливает только экземпляры InputError и пропускает другие исключения. Если вы снова сделаете такую же опечатку, будет корректно выведено сообщение о неопределённой переменной.

Утверждения (Assertions)

Утверждения – инструмент для простой проверки ошибок. Рассмотрим вспомогательную функцию assert:

```
function AssertionFailed(message) {
  this.message = message;
}
AssertionFailed.prototype = Object.create(Error.prototyp
e);

function assert(test, message) {
  if (!test)
    throw new AssertionFailed(message);
}

function lastElement(array) {
  assert(array.length > 0, "пустой массив в lastElement"
);
  return array[array.length - 1];
}
```

Это – компактный способ ужесточения требований к значениям, который выбрасывает исключение в случае, когда заданное условие не выполняется. К примеру, функция lastElement, добывающая последний элемент массива, вернула бы undefined для пустых массивов, если бы мы не использовали assertion. Извлечение последнего элемента пустого массива не имеет смысла, и это явно было бы ошибкой программиста.

Утверждения – способ убедиться в том, что ошибки провоцируют прерывание программы в том месте, где они совершены, а не просто выдают странные величины, которые передаются по системе и вызывают проблемы в каких-то других, не связанных с этим, местах.

Итог

Ошибки и недопустимые входные данные случаются в жизни. Ошибки в программах надо искать и исправлять. Их легче найти, используя автоматические системы проверок и добавляя утверждения в ваши программы.

Проблемы, вызванные чем-то, что неподвластно вашей программе, нужно обрабатывать достойно. Иногда, когда проблему можно решить локально, допустимо

возвращать специальные значения для отслеживания таких случаев. В других случаях предпочтительно использовать исключения.

Выброс исключения приводит к разматыванию стека до тех пор, пока не будет встречен блок try/catch или пока мы не дойдём до дна стека. Значение исключения будет передано в блок catch, который сможет удостовериться в том, что это исключение действительно то, которое он ждёт, и обработать его. Для работы с непредсказуемыми событиями в потоке программы можно использовать блоки finally, чтобы определённые части кода были выполнены в любом случае.

Упражнения

Повтор

Допустим, у вас есть функция primitiveMultiply, которая в 50% случаев перемножает 2 числа, а в остальных случаях выбрасывает исключение типа MultiplicatorUnitFailure. Напишите функцию, обёртывающую эту, и просто вызывающую её до тех пор, пока не будет получен успешный результат.

Убедитесь, что вы обрабатываете только нужные вам исключения.

```
function MultiplicatorUnitFailure() {}

function primitiveMultiply(a, b) {
  if (Math.random() < 0.5)
    return a * b;
  else
    throw new MultiplicatorUnitFailure();
}

function reliableMultiply(a, b) {
  // Ваш код
}

console.log(reliableMultiply(8, 8));
// → 64
```

Запертая коробка

Рассмотрим такой, достаточно надуманный, объект:

```
var box = {
  locked: true,
  unlock: function() { this.locked = false; },
  lock: function() { this.locked = true;  },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Заперто!");
    return this._content;
  }
};
```

Это коробочка с замком. Внутри лежит массив, но до него можно добраться только, когда коробка не заперта. Напрямую обращаться к свойству _content нельзя.

Напишите функцию withBoxUnlocked, принимающую в качестве аргумента функцию, которая отпирает коробку, выполняет функцию, и затем обязательно запирает коробку снова перед выходом – неважно, выполнилась ли переданная функция правильно, или она выбросила исключение.

```
function withBoxUnlocked(body) {
  // Ваш код
}

withBoxUnlocked(function() {
  box.content.push("золотишко");
});

try {
  withBoxUnlocked(function() {
    throw new Error("Пираты на горизонте! Отмена!");
  });
} catch (e) {
  console.log("Произошла ошибка:", e);
}
console.log(box.locked);
// → true
```

В качестве призовой игры убедитесь, что при вызове withBoxUnlocked, когда коробка не заперта, коробка остаётся незапертой.

Регулярные выражения

Некоторые люди, столкнувшись с проблемой, думают: «О, а использую-ка я регулярные выражения». Теперь у них есть две проблемы.

Джейми Завински

Юан-Ма сказал: «Требуется большая сила, чтобы резать дерево поперёк структуры древесины. Требуется много кода, чтобы программировать поперёк структуры проблемы.

Мастер Юан-Ма, «Книга программирования»

Инструменты и техники программирования выживают и распространяются хаотично-эволюционным способом. Иногда выживают не красивые и гениальные, а просто такие, которые достаточно хорошо работают в своей области – к примеру, если их интегрируют в другую успешную технологию.

В этой главе мы обсудим такой инструмент – регулярные выражения. Это способ описывать шаблоны в строковых данных. Они создают небольшой отдельный язык, который входит в JavaScript и во множество других языков и инструментов.

Регулярки одновременно очень странные и крайне полезные. Их синтаксис загадочен, а программный интерфейс в JavaScript для них неуклюж. Но это мощный инструмент для исследования и обработки строк. Разобравшись с ними, вы станете более эффективным программистом.

Создаём регулярное выражение

Регулярка – тип объекта. Её можно создать, вызвав конструктор RegExp, или написав нужный шаблон, окружённый слешами.

```
var re1 = new RegExp("abc");
var re2 = /abc/;
```

Оба этих регулярных выражения представляют один шаблон: символ “a”, за которым следует символ “b”, за которым следует символ “c”.

Если вы используете конструктор RegExp, тогда шаблон записывается как обычная строка, поэтому действуют все правила относительно обратных слешей.

Вторая запись, где шаблон находится между слешами, обрабатывает обратные слеши по-другому. Во-первых, так как шаблон заканчивается прямым слешем, то нужно ставить обратный слеш перед прямым слешем, который мы хотим включить в наш шаблон. Кроме того, обратные слеши, не являющиеся частью специальных символов типа \n, будут сохранены (а не проигнорированы, как в строках), и изменят смысл шаблона. У некоторых символов, таких, как знак вопроса или плюс, есть особое значение в регулярках, и если вам нужно найти такой символ, его также надо предварять обратным слешем.

```
var eighteenPlus = /eighteen\+/;
```

Чтобы знать, какие символы надо предварять слешем, вам надо выучить список всех специальных символов в регулярках. Пока это нереально, поэтому в случае сомнений просто ставьте обратный слеш перед любым символом, не являющимся буквой, числом или пробелом.

Проверяем на совпадения

У регулярок есть несколько методов. Простейший – test. Если передать ему строку, он вернёт булевское значение, сообщая, содержит ли строка вхождение заданного

шаблона.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

Регулярка, состоящая только из неспециальных символов, просто представляет собой последовательность этих символов. Если abc есть где-то в строке, которую мы проверяем (не только в начале), test вернёт true.

Ищем набор символов

Выяснить, содержит ли строка abc, можно было бы и при помощи indexOf. Регулярки позволяют пройти дальше и составлять более сложные шаблоны.

Допустим, нам надо найти любой номер. Когда мы в регулярке помещаем набор символов в квадратные скобки, это означает, что эта часть выражения совпадает с любым из символов в скобках.

Оба выражения находятся в строчках, содержащих цифру.

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

В квадратных скобках тире между двумя символами используется для задания диапазона символов, где последовательность задаётся кодировкой Unicode. Символы от 0 до 9 находятся там просто подряд (коды с 48 до 57), поэтому [0-9] захватывает их все и совпадает с любой цифрой.

У нескольких групп символов есть свои встроенные сокращения:

`\d` – любая цифра

`\w` – алфавитно-цифровой символ

`\s` – пробельный символ (пробел, табуляция, перевод строки, и т.п.)

`\D` – не цифра

`\W` – не алфавитно-цифровой символ

`\S` – не пробельный символ

`.` – любой символ, кроме перевода строки

Таким образом можно задать формат даты и времени вроде 30-01-2003 15:20 следующим выражением:

```
var dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("30-01-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

Выглядит ужасно, не так ли? Слишком много обратных слешей, которые затрудняют понимание шаблона. Позже мы слегка улучшим его.

Обратные слеши можно использовать и в квадратных скобках. Например, [\d.] означает любую цифру или точку. Заметьте, что точка внутри квадратных скобок теряет своё особое значение и превращается просто в точку. То же касается и других специальных символов, типа +.

Инвертировать набор символов – то есть, сказать, что вам надо найти любой символ, кроме тех, что есть в наборе – можно, поставив знак ^ сразу после открывающей квадратной скобки.

```
var notBinary = /[^01]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

Повторяем части шаблона

Мы знаем, как найти одну цифру. А если нам надо найти число целиком – последовательность из одной или более цифр?

Если поставить после чего-либо в регулярке знак +, это будет означать, что этот элемент может быть повторён более одного раза. /\d+/ означает одну или несколько цифр.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

У звёздочки * значение почти такое же, но она разрешает шаблону присутствовать ноль раз. Если после чего-то стоит звёздочка, то оно никогда не препятствует нахождению шаблона в строке – оно просто находится там ноль раз.

Знак вопроса делает часть шаблона необязательной, то есть она может встретиться ноль или один раз. В следующем примере символ u может встречаться, но шаблон совпадает и тогда, когда его нет.

```
var neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

Чтобы задать точное количество раз, которое шаблон должен встретиться, используются фигурные скобки. {4} после элемента означает, что он должен встретиться в строке 4 раза. Также можно задать промежуток: {2,4} означает, что элемент должен встретиться не менее 2 и не более 4 раз.

Ещё одна версия формата даты и времени, где разрешены дни, месяцы и часы из одной или двух цифр. И ещё она чуть более читаема.

```
var dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("30-1-2003 8:45"));
// → true
```

Можно использовать промежутки с открытым концом, опуская одно из чисел. {,5} означает, что шаблон может встретиться от нуля до пяти раз, а {5,} – от пяти и более.

Группировка подвыражений

Чтобы использовать операторы * или + на нескольких элементах сразу, можно использовать круглые скобки. Часть регулярки, заключённая в скобки, считается одним элементом с точки зрения операторов.

```
var cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

Первый и второй плюсы относятся только ко вторым буквам o в словах boo и hoo. Третий + относится к целой группе (hoo+), находя одну или несколько таких последовательностей.

Буква i в конце выражения делает регулярку нечувствительной к регистру симолов – так, что B совпадает с b.

Совпадения и группы

Метод test – самый простой метод проверки регулярок. Он только сообщает, было ли найдено совпадение, или нет. У регулярок есть ещё метод exec, который вернёт null, если ничего не было найдено, а в противном случае вернёт объект с информацией о совпадении.

```
var match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

У возвращаемого exec объекта есть свойство index, где содержится номер символа, с которого случилось совпадение. А вообще объект выглядит как массив строк, где первый элемент – строка, которую проверяли на совпадение. В нашем примере это будет последовательность цифр, которую мы искали.

У строк есть метод match, работающий примерно так же.

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

Когда в регулярке содержатся подвыражения, сгруппированные круглыми скобками, текст, совпавший с этими группами, тоже появится в массиве. Первый элемент всегда совпадение целиком. Второй – часть, совпавшая с первой группой (той, у кого круглые скобки встретились раньше всех), затем со второй группой, и так далее.

```
var quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

Когда группа не найдена вообще (например, если за ней стоит знак вопроса), её позиция в массиве содержит undefined. Если группа совпала несколько раз, то в массиве будет только последнее совпадение.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

Группы полезны для извлечения частей строк. Если нам не просто надо проверить, есть ли в строке дата, а извлечь её и создать представляющий дату объект, мы можем заключить последовательности цифр в круглые скобки и выбрать дату из результата exec.

Но для начала небольшое отступление, в котором мы узнаем предпочтительный способ хранения даты и времени в JavaScript.

Тип даты

В JavaScript есть стандартный тип объекта для дат – а точнее, моментов во времени. Он называется Date. Если просто создать объект даты через new, вы получите текущие дату и ремя.

```
console.log(new Date());
// → Sun Nov 09 2014 00:07:57 GMT+0300 (CET)
```

Также можно создать объект, содержащий заданное время

```
console.log(new Date(2015, 9, 21));
// → Wed Oct 21 2015 00:00:00 GMT+0300 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0300 (CET)
```

JavaScript использует соглашение, в котором номера месяцев начинаются с нуля, а номера дней – с единицы. Это глупо и нелепо. Поберегитесь.

Последние четыре аргумента (часы, минуты, секунды и миллисекунды) необязательны, и в случае отсутствия приравниваются к нулю.

Метки времени хранятся как количество миллисекунд, прошедших с начала 1970 года. Для времени до 1970 года используются отрицательные числа (это связано с

соглашением по Unix time, которое было создано примерно в то время). Метод getTime объекта даты возвращает это число. Оно, естественно, большое.

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

Если задать конструктору Date один аргумент, он будет восприниматься как количество миллисекунд. Текущее значение миллисекунд можно получить, создав объект Date и вызвав метод getTime, или же вызвав функцию Date.now.

У объекта Date для извлечения его компонентов есть методы getFullYear, getMonth, getDate, getHours, getMinutes, и getSeconds. Есть также метод getYear, возвращающий довольно бесполезный двузначный код, типа 93 или 14.

Заключив нужные части шаблона в круглые скобки, мы можем создать объект даты прямо из строки.

```
function findDate(string) {
  var dateTime = /(\d{1,2})-(\d{1,2})-(\d{4})/;
  var match = dateTime.exec(string);
  return new Date(Number(match[3]),
                  Number(match[2]) - 1,
                  Number(match[1]));
}
console.log(findDate("30-1-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

Границы слова и строки

К сожалению, findDate так же радостно извлечёт бессмысленную дату 00-1-3000 из строки «100-1-30000». Совпадение может случиться в любом месте строки, так что в данном случае он просто начнёт со второго символа и закончит на предпоследнем.

Если нам надо принудить совпадение взять всю строку целиком, мы используем метки ^ и $. ^ совпадает с началом строки, а $ с концом. Поэтому /^\d+$/ совпадает со строкой, состоящей только из одной или нескольких цифр, /^!/ совпадает со сторокой, начинающейся с восклицательного знака, а /x^/ не совпадает ни с какой строчкой (перед началом строки не может быть x).

Если, с другой стороны, нам просто надо убедиться, что дата начинается и заканчивается на границе слова, мы используем метку \b. Границей слова может быть начало или конец строки, или любое место строки, где с одной стороны стоит алфавитно-цифровой символ \w, а с другой – не алфавитно-цифровой.

```
console.log(/cat/.test("concatenate"));
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

Отметим, что метка границы не представляет из себя символ. Это просто ограничение, обозначающее, что совпадение происходит только если выполняется определённое условие.

Шаблоны с выбором

Допустим, надо выяснить, содержит ли текст не просто номер, а номер, за которым следует pig, cow, или chicken в единственном или множественном числе.

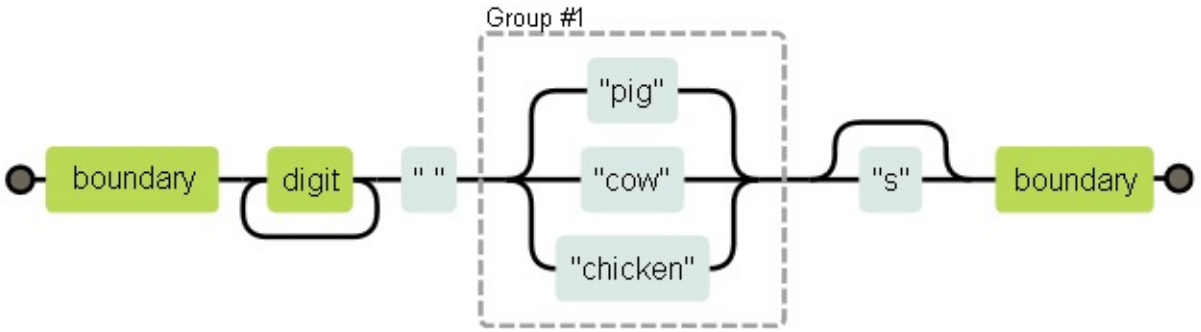
Можно было бы написать три регулярки и проверить их по очереди, но есть способ лучше. Символ | обозначает выбор между шаблонами слева и справа от него. И можно сказать следующее:

```
var animalCount = /\b\d+ (pig|cow|chicken)s?\b/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

Скобки ограничивают часть шаблона, к которой применяется |, и можно поставить много таких операторов друг за другом, чтобы обозначить выбор из более чем двух вариантов.

Механизм поиска

Регулярные выражения можно рассматривать как блок-схемы. Следующая диаграмма описывает последний животноводческий пример.



Выражение совпадает со строкой, если можно найти путь с левой части диаграммы в правую. Мы запоминаем текущее положение в строке, и каждый раз, проходя

прямоугольник, проверяем, что часть строки сразу за нашим положением в ней совпадает с содержимым прямоугольника.

Значит, проверка совпадения нашей регулярки в строке «the 3 pigs» при прохождении по блок-схеме выглядит так:

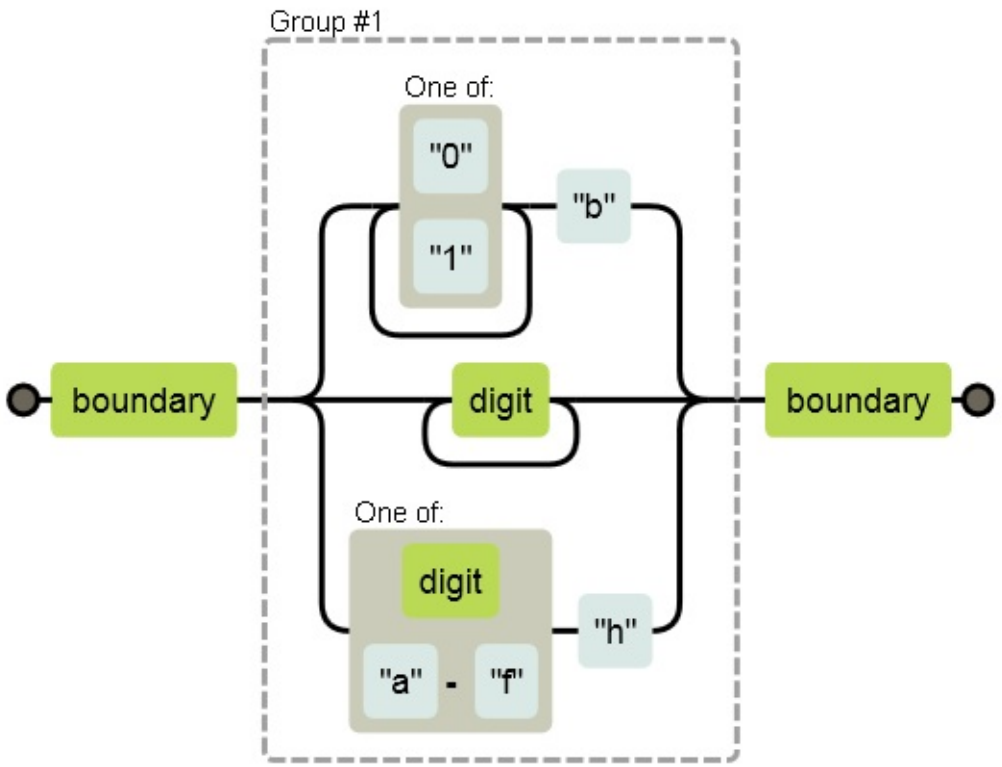
— на позиции 4 есть граница слова, и проходим первый прямоугольник — начиная с 4 позиции находим цифру, и проходим второй прямоугольник — на позиции 5 один путь замыкается назад перед вторым прямоугольником, а второй проходит далее к прямоугольнику с пробелом. У нас пробел, а не цифра, и мы выбираем второй путь. — теперь мы на позиции 6, начало “pigs”, и на тройном разветвлении путей. В строке нет “cow” или “chicken”, зато есть “pig”, поэтому мы выбираем этот путь. — на позиции 9 после тройного разветвления, один путь обходит “s” и направляется к последнему прямоугольнику с границей слова, а второй проходит через “s”. У нас есть “s”, поэтому мы идём туда. — на позиции 10 мы в конце строки, и совпасть может только граница слова. Конец строки считается границей, и мы проходим через последний прямоугольник. И вот мы успешно нашли наш шаблон.

В принципе, работают регулярные выражения следующим образом: алгоритм начинает в начале строки и пытается найти совпадение там. В нашем случае там

есть граница слова, поэтому он проходит первый прямоугольник – но там нет цифры, поэтому на втором прямоугольнике он спотыкается. Потом он двигается ко второму символу в строке, и пытается найти совпадение там… И так далее, пока он не находит совпадение или не доходит до конца строки, в таком случае совпадение не найдено.

Откаты

Регулярка /\b([01]+b|\d+|[\da-f]h)\b/ совпадает либо с двоичным числом, за которым следует b, либо с десятичным числом без суффикса, либо шестнадцатеричным (цифры от 0 до 9 или символы от a до f), за которым идёт h. Соответствующая диаграмма:



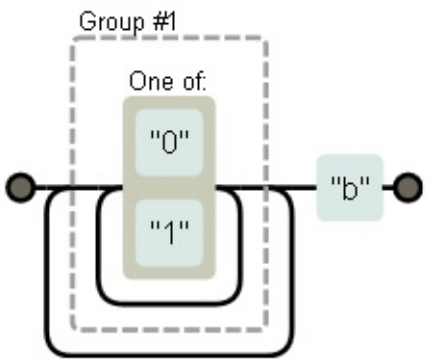
В поисках совпадения может случиться, что алгоритм пошёл по верхнему пути (двоичное число), даже если в строке нет такого числа. Если там есть строка “103”, к примеру, понятно, что только достигнув цифры 3 алгоритм поймёт, что он на неправильном пути. Вообще строка совпадает с регуляркой, просто не в этой ветке.

Тогда алгоритм совершает откат. На развилке он запоминает текущее положение (в нашем случае, это начало строки, сразу после границы слова), чтобы можно было вернуться назад и попробовать другой путь, если выбранный не срабатывает. Для строки “103” после встречи с тройкой он вернётся и попытается пройти путь для десятичных чисел. Это сработает, поэтому совпадение будет найдено.

Алгоритм останавливается, как только найдёт полное совпадение. Это значит, что даже если несколько вариантов могут подойти, используется только один из них (в том порядке, в каком они появляются в регулярке).

Откаты случаются при использовании операторов повторения, таких, как + и *. Если вы ищете /^.*x/ в строке «abcxe», часть регулярки .* попробует поглотить всю строчку. Алгоритм затем сообразит, что ему нужен ещё и “x”. Так как никакого “x” после конца строки нет, алгоритм попробует поискать совпадение, откатившись на один символ. После abcx тоже нет x, тогда он снова откатывается, уже к подстроке abc. И после строчки он находит x и докладывает об успешном совпадении, на позициях с 0 по 4.

Можно написать регулярку, которая приведёт ко множественным откатам. Такая проблема возникает, когда шаблон может совпасть с входными данными множеством разных способов. Например, если мы ошибёмся при написании регулярки для двоичных чисел, мы можем случайно написать что-то вроде /([01]+)+b/.



Если алгоритм будет искать такой шаблон в длинной строке из нолей и единиц, не содержащей в конце “b”, он сначала пройдёт по внутренней петле, пока у него не кончатся цифры. Тогда он заметит, что в конце нет “b”, сделает откат на одну позицию, пройдёт по внешней петле, опять сдастся, попытается откатиться на ещё одну позицию по внутренней петле… И будет дальше искать таким образом, задействуя обе петли. То есть, количество работы с каждым символом строки будет удваиваться. Даже для нескольких десятков символов поиск совпадения займёт очень долгое время.

Метод replace

У строк есть метод replace, который может заменять часть строки другой строкой.

```
console.log("папа".replace("п", "м"));
// → мапа
```

Первый аргумент может быть и регуляркой, в каком случае заменяется первое вхождение регулярки в строке. Когда к регулярке добавляется опция “g” (global, всеобщий), заменяются все вхождения, а не только первое

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

Имело бы смысл передавать опцию «заменить все» через отдельный аргумент, или через отдельный метод типа replaceAll. Но к сожалению, опция передаётся через саму регулярку.

Вся сила регулярок раскрывается, когда мы используем ссылки на найденные в строке группы, заданные в регулярке. Например, у нас есть строка, содержащая имена людей, одно имя на строчку, в формате «Фамилия, Имя». Если нам надо поменять их местами и убрать запятую, чтобы получилось «Имя Фамилия», мы пишем следующее:

```
console.log(
  "Hopper, Grace\nMcCarthy, John\nRitchie, Dennis"
    .replace(/([\w ]+), ([\w ]+)/g, "$2 $1"));
// → Grace Hopper
//   John McCarthy
//   Dennis Ritchie
```

$1 и $2 в строчке на замену ссылаются на группы символов, заключённые в скобки. $1 заменяется текстом, который совпал с первой группой, $2 – со второй группой, и так далее, до $9. Всё совпадение целиком содержится в переменной $&.

Также можно в качестве второго аргумента передавать и функцию. Для каждой замены будет вызвана функция, аргументами которой будут найденные группы (и вся совпадающая часть строки целиком), а её результат будет вставлен в новую строку.

Простой пример:

```
var s = "the cia and fbi";
console.log(s.replace(/\b(fbi|cia)\b/g, function(str) {
  return str.toUpperCase();
}));
// → the CIA and FBI
```

А вот более интересный:

```
var stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) // остался только один, удаляем 's' в
 конце
    unit = unit.slice(0, unit.length - 1);
  else if (amount == 0)
    amount = "no";
  return amount + " " + unit;
}
console.log(stock.replace(/(\d+) (\w+)/g, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

Код принимает строку, находит все вхождения чисел, за которыми идёт слово, и возвращает строчку, где каждое число уменьшено на единицу.

Группа (\d+) попадает в аргумент amount, а (\w+) – в unit. Функция преобразовывает amount в число – и это всегда срабатывает, потому что наш шаблон как раз \d+. И затем вносит изменения в слово, на случай если остался всего 1 предмет.

Жадность

Несложно при помощи replace написать функцию, убирающую все комментарии из кода JavaScript. Вот первая попытка:

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
console.log(stripComments("x = 10;// ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1  1
```

Часть перед оператором «или» совпадает с двумя слешами, за которыми идёт любое количество символов, кроме символов перевода строки. Часть, убирающая многострочные комментарии, более сложна. Мы используем , т.е. любой символ, не являющийся пустым, в качестве способа найти любой символ. Мы не можем использовать точку, потому что блочные комментарии продолжаются и на новой строке, а символ перевода строки не совпадает с точкой.

Но вывод предыдущего примера неправильный. Почему?

Часть *сначала попытается захватить столько символов, сколько может. Если из-за этого следующая часть регулярки не найдёт себе совпадения, произойдёт откат на один символ и попробует снова. В примере, алгоритм пытается захватить всю строку, и затем откатывается. Откатившись на 4 символа назад, он найдёт в строчке* / — а это не то, чего мы

добивались. Мы-то хотели захватить только один комментарий, а не пройти до конца строки и найти последний комментарий.

Из-за этого мы говорим, что операторы повторения (+, *, ?, and {}) жадные, то есть они сначала захватывают, сколько могут, а потом идут назад. Если вы поместите вопрос после такого оператора (+?, ?, ??, {}?), они превратятся в нежадных, и начнут находить самые маленькие из возможных вхождений.

И это то, что нам нужно. Заставив звёздочку находить совпадения в минимально возможном количестве символов строчки, мы поглощаем только один блок комментариев, и не более того.

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

Множество ошибок возникает при использовании жадных операторов вместо нежадных. При использовании оператора повтора сначала всегда рассматривайте вариант нежадного оператора.

Динамическое создание объектов RegExp

В некоторых случаях точный шаблон неизвестен во время написания кода. Например, вам надо будет искать имя пользователя в тексте, и заключать его в подчёркивания. Так как вы узнаете имя только после запуска программы, вы не можете использовать запись со слешами.

Но вы можете построить строку и использовать конструктор RegExp. Вот пример:

```
var name = "гарри";
var text = "А у Гарри на лбу шрам.";
var regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → А у _Гарри_ на лбу шрам.
```

При создании границ слова приходится использовать двойные слеши, потому что мы пишем их в нормальной строке, а не в регулярке с прямыми слешами. Второй аргумент для RegExp содержит опции для регулярок – в нашем случае “gi”, т.е. глобальный и регистро-независимый.

Но что, если имя будет «dea+hl[]rd» (если наш пользователь – кульхацкер)? В результате мы получим бессмысленную регулярку, которая не найдёт в строке совпадений.

Мы можем добавить обратных слешей перед любым символом, который нам не нравится. Мы не можем добавлять обратные слеши перед буквами, потому что \b или \n – это спецсимволы. Но добавлять слеши перед любыми не алфавитно-цифровыми символами можно без проблем.

```
var name = "dea+hl[]rd";
var text = "Этот dea+hl[]rd всех достал.";
var escaped = name.replace(/[^\w\s]/g, "\\$&");
var regexp = new RegExp("\\b(" + escaped + ")\\b", "gi")
;
console.log(text.replace(regexp, "_$1_"));
// → Этот _dea+hl[]rd_ всех достал.
```

Метод search

Метод indexOf нельзя использовать с регулярками. Зато есть метод search, который как раз ожидает регулярку. Как и indexOf, он возвращает индекс первого вхождения, или -1, если его не случилось.

```
console.log("  word".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

К сожалению, никак нельзя задать, чтобы метод искал совпадение, начиная с конкретного смещения (как это можно сделать с indexOf). Это было бы полезно.

Свойство lastIndex

Метод exec тоже не даёт удобного способа начать поиск с заданной позиции в строке. Но неудобный способ даёт.

У объекта регулярок есть свойства. Одно из них – source, содержащее строку. Ещё одно – lastIndex, контролирующее, в некоторых условиях, где начнётся следующий поиск вхождений.

Эти условия включают необходимость присутствия глобальной опции g, и то, что поиск должен идти с применением метода exec. Более разумным решением было бы просто допустить дополнительный аргумент для передачи в exec, но разумность – не основополагающая черта в интерфейсе регулярок JavaScript.

```
var pattern = /y/g;
pattern.lastIndex = 3;
var match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

Если поиск был успешным, вызов exec обновляет свойство lastIndex, чтоб оно указывало на позицию после найденного вхождения. Если успеха не было, lastIndex устанавливается в ноль – как и lastIndex у только что созданного объекта.

При использовании глобальной переменной-регулярки и нескольких вызовов exec эти автоматические обновления lastIndex могут привести к проблемам. Ваша регулярка может начать поиск с позиции, оставшейся с предыдущего вызова.

```
var digit = /\d/g;
console.log(digit.exec("here it is: 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

Ещё один интересный эффект опции g в том, что она меняет работу метода match. Когда он вызывается с этой опцией, вместо возврата массива, похожего на результат

работы exec, он находит все вхождения шаблона в строке и возвращает массив из найденных подстрок.

```
console.log("Банан".match(/ан/g));
// → ["ан", "ан"]
```

Так что поосторожнее с глобальными переменными-регулярками. В случаях, когда они необходимы – вызовы replace или места, где вы специально используете lastIndex – пожалуй и все случаи, в которых их следует применять.

Циклы по вхождениям

Типичная задача – пройти по всем вхождениям шаблона в строку так, чтобы иметь доступ к объекту match в теле цикла, используя lastIndex и exec.

```
var input = "Строчка с 3 числами в ней... 42 и 88.";
var number = /\b(\d+)\b/g;
var match;
while (match = number.exec(input))
  console.log("Нашёл ", match[1], " на ", match.index);
// → Нашёл 3 на 10
//   Нашёл 42 на 29
//   Нашёл 88 на 34
```

Используется тот факт, что значением присвоения является присваиваемое значение. Используя конструкцию match = re.exec(input) в качестве условия в цикле while, мы производим поиск в начале каждой итерации, сохраняем результат в переменной, и заканчиваем цикл, когда все совпадения найдены.

Разбор INI файлы

В заключение главы рассмотрим задачу с использованием регулярок. Представьте, что мы пишем программу, собирающую сведения о наших врагах через интернет в автоматическом режиме. (Всю программу писать не будем, только ту часть, которая читает файл с настройками. Извините.) Файл выглядит так:

```
searchengine=http://www.google.com/search?q=$1
spitefulness=9.7

; перед комментариями ставится точка с запятой
; каждая секция относится к отдельному врагу
[larry]
fullname=Larry Doe
type=бычара из детсада
website=http://www.geocities.com/CapeCanaveral/11451

[gargamel]
fullname=Gargamel
type=злой волшебник
outputdir=/home/marijn/enemies/gargamel
```

Точный формат файла (который довольно широко используется, и обычно называется INI), следующий:

— пустые строки и строки, начинающиеся с точки с запятой, игнорируются — строки, заключённые в квадратные скобки, начинают новую секцию — строки, содержащие алфавитно-цифровой идентификатор, за которым следует =, добавляют настройку в данной секции

Всё остальное – неверные данные.

Наша задача – преобразовать такую строку в массив объектов, каждый со свойством name и массивом настроек. Для каждой секции нужен один объект, и ещё один – для глобальных настроек сверху файла.

Так как файл надо разбирать построчно, неплохо начать с разбиения файла на строки. Для этого в главе 6 мы использовали string.split("\n"). Некоторые операционки используют для перевода строки не один символ \n, а два — \r\n. Так как метод split принимает регулярки в качестве аргумента, мы можем делить линии при помощи выражения /\r?\n/, разрешающего и одиночные \n и \r\n между строками.

```
function parseINI(string) {
  // Начнём с объекта, содержащего настройки верхнего ур
овня
  var currentSection = {name: null, fields: []};
  var categories = [currentSection];

  string.split(/\r?\n/).forEach(function(line) {
    var match;
    if (/^\s*(;.*)?$/.test(line)) {
      return;
    } else if (match = line.match(/^\[(.*)\]$/)) {
      currentSection = {name: match[1], fields: []};
      categories.push(currentSection);
    } else if (match = line.match(/^(\w+)=(.*)$/)) {
      currentSection.fields.push({name: match[1],
                                  value: match[2]});
    } else {
      throw new Error("Строчка '" + line + "' содержит н
еверные данные.");
    }
  });

  return categories;
}
```

Код проходит все строки, обновляя объект текущей секции “current section”. Сначала он проверяет, можно ли игнорировать строчку, при помощи регулярки /^\s(;.)?$/. Соображаете, как это работает? Часть между скобок совпадает с комментариями, а ? делает так, что регулярка совпадёт и со строчками, состоящими из одних пробелов.

Если строка – не комментарий, код проверяет, начинает ли она новую секцию. Если да, он создаёт новый объект для текущей секции, к которому добавляются последующие настройки.

Последняя осмысленная возможность – строка является обычной настройкой, и в этом случае она добавляется к текущему объекту.

Если ни один вариант не сработал, функция выдаёт ошибку.

Заметьте, как частое использование ^ и $ заботится о том, что выражение совпадает со всей строкой целиком, а не с частью. Если их не использовать, код в целом будет работать, но иногда будет выдавать странные результаты, и такую ошибку будет трудно отследить.

Конструкция if (match = string.match(...)) похожа на трюк, использующий присвоение как условие в цикле while. Часто вы не знаете, что вызов match будет успешным, поэтому вы можете получить доступ к результирующему объекту только внутри блока if, который это проверяет. Чтоб не разбивать красивую цепочку проверок if, мы присваиваем результат поиска переменной, и сразу используем это присвоение как проверку.

Международные символы

Из-за изначально простой реализации языка, и последующей фиксации такой реализации «в граните», регулярки JavaScript тупят с символами, не встречающимися в английском языке. К примеру, символ «буквы» с точки зрения регулярок JavaScript, может быть одним из 26 букв английского алфавита, и почему-то ещё подчёркиванием. Буквы типа é или β, однозначно являющиеся буквами, не совпадают с \w (и совпадут с \W, то есть с не-буквой).

По странному стечению обстоятельств, исторически \s (пробел) совпадает со всеми символами, которые в Unicode считаются пробельными, включая такие штуки, как неразрывный пробел или монгольский разделитель гласных.

У некоторых реализаций регулярок в других языках есть особый синтаксис для поиска специальных категорий символов Unicode, типа «все прописные буквы», «все знаки препинания» или «управляющие символы». Есть планы по добавлению таких категорий и в JavaScript, но они, видимо, будут реализованы не скоро.

Итог

Регулярки – это объекты, представляющие шаблоны поиска в строках. Они используют свой синтаксис для выражения этих шаблонов.

`/abc/` Последовательность символов `/[abc]/` Любой символ из списка `/[^abc]/` Любой символ, кроме символов из списка `/[0-9]/` Любой символ из промежутка `/x+/` Одно или более вхождений шаблона x `/x+?/` Одно или более вхождений, нежадное `/x*/` Ноль или более вхождений `/x?/` Ноль или одно вхождение `/x{2,4}/` От двух до четырёх вхождений `/(abc)/` Группа `/a|b|c/` Любой из нескольких шаблонов `/\d/` Любая цифра `/\w/` Любой алфавитно-цифровой символ («буква») `/\s/` Любой пробельный символ `/./` Любой символ, кроме переводов строки `/\b/` Граница слова `/^/` Начало строки `/$/` Конец строки

У регулярки есть метод test, для проверки того, есть ли шаблон в строке. Есть метод exec, возвращающий массив, содержащий все найденные группы. У массива есть свойство index, показывающее, где начался поиск.

У строк есть метод match для поиска шаблонов, и метод search, возвращающий только начальную позицию вхождения. Метод replace может заменять вхождения шаблона на другую строку. Кроме этого, вы можете

передать в replace функцию, которая будет строить строчку на замену, основываясь на шаблоне и найденных группах.

У регулярок есть настройки, которые пишут после закрывающего слеша. Опция i делает регулярку регистронезависимой, а опция g делает её глобальной, что, кроме прочего, заставляет метод replace заменять все найденные вхождения, а не только первое.

Конструктор RegExp можно использовать для создания регулярок из строк.

Регулярки – острый инструмент с неудобной ручкой. Они сильно упрощают одни задачи, и могут стать неуправляемыми при решении других, сложных задач. Часть умения пользоваться регулярками состоит в том, чтобы уметь сопротивляться искушению запихнуть в них задачу, для которой они не предназначены.

Упражнения

Неизбежно при решении задач у вас возникнут непонятные случаи, и вы можете иногда отчаиваться, видя непредсказуемое поведение некоторых регулярок.

Иногда помогает изучить поведение регулярки через онлайн-сервис типа debuggex.com, где можно посмотреть её визуализацию и сравнить с желаемым эффектом.

Регулярный гольф

«Гольфом» в коде называют игру, где нужно выразить заданную программу минимальным количеством символов. Регулярный гольф – практическое упражнение по написанию наименьших возможных регулярок для поиска заданного шаблона, и только его.

Для каждой из подстрочек напишите регулярку для проверки их нахождения в строке. Регулярка должна находить только эти указанные подстроки. Не волнуйтесь насчёт границ слов, если это не упомянуто особо. Когда у вас получится работающая регулярка, попробуйте её уменьшить.

— car и cat — pop и prop — ferret, ferry, и ferrari — Любое слово, заканчивающееся на ious — Пробел, за которым идёт точка, запятая, двоеточие или точка с запятой. — Слово длинее шести букв — Слово без букв e

```
// Впишите свои регулярки

verify(/.../,
       ["my car", "bad cats"],
       ["camper", "high art"]);
```

```
verify(/.../,
       ["pop culture", "mad props"],
       ["plop"]);

verify(/.../,
       ["ferret", "ferry", "ferrari"],
       ["ferrum", "transfer A"]);

verify(/.../,
       ["how delicious", "spacious room"],
       ["ruinous", "consciousness"]);

verify(/.../,
       ["bad punctuation ."],
       ["escape the dot"]);

verify(/.../,
       ["hottentottententen"],
       ["no", "hotten totten tenten"]);

verify(/.../,
       ["red platypus", "wobbling nest"],
       ["earth bed", "learning ape"]);

function verify(regexp, yes, no) {
  // Ignore unfinished exercises
  if (regexp.source == "...") return;
  yes.forEach(function(s) {
    if (!regexp.test(s))
      console.log("Не нашлось '" + s + "'");
  });
  no.forEach(function(s) {
    if (regexp.test(s))
      console.log("Неожиданное вхождение '" + s + "'");
  });
```

```
}
```

Кавычки в тексте

Допустим, вы написали рассказ, и везде для обозначения диалогов использовали одинарные кавычки. Теперь вы хотите заменить кавычки диалогов на двойные, и оставить одинарные в сокращениях слов типа aren't.

Придумайте шаблон, различающий два этих использования кавычек, и напишите вызов метода replace, который производит замену.

Снова числа

Последовательности цифр можно найти простой регуляркой /\d+/.

Напишите выражение, находящее только числа, записанные в стиле JavaScript. Оно должно поддерживать возможный минус или плюс перед числом, десятичную точку, и экспоненциальную запись 5e-3 или 1E10 – опять-таки с возможными плюсом или минусом. Также заметьте, что до или после точки не обязательно могут стоять цифры, но при этом число не может состоять из одной точки. То есть, .5 или 5. – допустимые числа, а одна точка сама по себе – нет.

```
// Впишите сюда регулярку.
var number = /^...$/;

// Tests:
["1", "-1", "+15", "1.55", ".5", "5.", "1.3e2", "1E-4",
 "1e+12"].forEach(function(s) {
  if (!number.test(s))
    console.log("Не нашла '" + s + "'");
});
["1a", "+-1", "1.2.3", "1+1", "1e4.5", ".5.", "1f5",
 "."].forEach(function(s) {
  if (number.test(s))
    console.log("Неправильно принято '" + s + "'");
});
```

Модули

Начинающий программист пишет программы так, как муравьи строят муравейник – по кусочку, без размышления над общей структурой. Его программы как песок. Они могут недолго простоять, но вырастая, они разваливаются.
Поняв проблему, программист тратит много времени на размышления о структуре. Его программы получаются жёстко структурированными, как каменные изваяния. Они тверды, но когда их нужно менять, над ними приходится совершать насилие.
Мастер-программист знает, когда нужна структура, а когда нужно оставить вещи в простом виде. Его программы словно глина – твёрдые, но податливые.

Мастер Юан-Ма, Книга программирования

У каждой программы есть структура. В частностях она определяется тем, как программист делит код на функции и блоки внутри этих функций. Программисты вольны в создании структуры своей программы. Структура определяется больше вкусом программиста, нежели функциональностью программы.

В случае больших программ отдельные функции уже теряются в коде, и нам необходима единица организации кода больших масштабов. Модули группируют программный код по каким-то определённым признакам. В этой главе мы рассмотрим преимущества такого деления и техники создания модулей в JavaScript.

Зачем нужны модули

Есть несколько причин, по которым авторы делят свои книги на главы и секции. Это помогает читателю понять, как построена книга, и найти нужные им части. Автору это помогает концентрироваться на каждой конкретной части. Преимущества организации программ в нескольких файлах, или модулях, примерно те же. Структуризация помогает незнакомым с кодом людям найти то, что им нужно, и помогает программистам хранить связанные друг с другом вещи в одном месте.

Некоторые программы организованы по модели обычного текста, где порядок следования чётко определён, и где читателю предлагается последовательное изучение программы и множество прозы (комментариев) для описания кода. Это делает чтение кода менее пугающим (а чтение чужого кода обычно пугает), но требует больших усилий при создании кода. Также такую

программу сложнее менять, потому что части прозы связаны между собой сильнее, чем части кода. Этот стиль называется литературным программированием. Те главы книги, в которых обсуждаются проекты, можно считать литературным кодом.

Обычно структурирование чего-либо требует затрат энергии. На ранних стадиях проекта, когда вы ещё не уверены, что где будет, и какие модули вообще нужны, я пропагандирую бесструктурную минималистическую организацию кода. Просто размещайте все части там, где удобно, пока код не стабилизируется. Таким образом не придётся тратить время на перестановку кусков программы, и вы не поймаете себя в такую структуру, которая не подходит для вашей программы.

Пространство имён

У большинства современных ЯП есть промежуточные области видимости (ОВ) между глобальной (видно всем) и локальной (видно только этой функции). У JavaScript такого нет. По умолчанию, всё, что необходимо видеть снаружи функции верхнего уровня, находится в глобальной ОВ.

Загрязнение пространства имён (ПИ), когда не связанные друг с другом части кода делят один набор переменных, упоминалась в главе 4. Там объект Math был приведён в качестве примера объекта, который группирует функциональность, связанную с математикой, в виде модуля.

Хотя JavaScript не предлагает непосредственно конструкции для создания модуля, мы можем использовать объекты для создания подпространств имён, доступных отовсюду. А функции можно использовать для создания изолированных частных пространств имён внутри модуля. Чуть дальше мы обсудим способ построения достаточно удобных модулей, изолирующих ПИ при помощи базовых концепций языка.

Повторное использование

В проекте, не разбитом на модули, непонятно, какие части кода необходимы для конкретной функции. В моей программе, шпионящей за врагами (глава 9), я написал функцию чтения файлов с настройками. Если мне понадобится использовать её в другом проекте, я должен буду скопировать части старой программы, которые вроде бы связаны с этой функцией, в мой новый проект. А

если я найду там ошибку, я её исправлю только в том проекте, над которым работаю в данный момент, и забуду исправить её во всех остальных.

Когда у вас будет множество таких сдублированных кусков кода, вы обнаружите, что тратите кучу времени и сил на их копирование и обновление. Если разместить связанные между собой части программ в отдельные модули, их будет проще отслеживать, исправлять и обновлять, потому что везде, где этот функционал потребуется, вы сможете просто загрузить этот модуль из файла.

Эту идею можно использовать ещё лучше, если чётко прописать взаимоотношения разных модулей – кто от кого зависит. Тогда можно автоматизировать процесс установки и обновления внешних модулей (библиотек).

Если ещё развить идею – представьте себе онлайн-сервис, который отслеживает и распространяет сотни тысяч таких библиотек, и вы можете искать нужную вам функциональность среди них, а когда найдёте – ваш проект автоматически скачает её.

И такой сервис есть! Он называется NPM (npmjs.org). NPM – онлайн-база модулей и инструмент для скачивания и апгрейда модулей, от которых зависит ваша

программа. Он вырос из Node.js, окружения JavaScript, не требующего браузера, которое мы обсудим в главе 20, но также может использоваться и в браузерных программах.

Устранение связей (Decoupling)

Ещё одна задача модулей – изолировать несвязанные между собой части кода так, как это делают интерфейсы объектов. Хорошо продуманный модуль предоставляет интерфейс для его использования вовне. Когда модуль обновляют или исправляют, существующий интерфейс остаётся неизменным, чтобы другие модули могли использовать новую, обновлённую версию без изменений в них самих.

Стабильный интерфейс не означает, что в него не добавляют новые функции, методы или переменные. Главное, что существующая функциональность не удаляется и её смысл не меняется. Хороший интерфейс позволяет модулю расти, не ломая старый интерфейс. А это значит – выставлять наружу как можно меньше внутренней кухни модуля, при этом язык интерфейса должен быть достаточно гибким и мощным для применения в различных ситуациях.

Интерфейсы, выполняющие простую задачу, вроде чтения настроек из файла, выходят такими естественным образом. Для других – к примеру, для редактора текстов, у которого есть множество разных аспектов, требующих доступа извне (содержимое, стили, действия пользователя и т.п.) интерфейс необходимо скрупулёзно продумывать.

Использование функций в качестве пространств имён

Функции – единственная вещь в JavaScript, создающая новую область видимости. Если нам нужно, чтобы у модулей была своя область видимости, придётся основывать их на функциях.

Обратите внимание на простейший модуль, связывающий имена с номерами дней недели – как делает метод getDay объекта Date.

```
var names = ["Понедельник", "Вторник", "Среда", "Четверг"
, "Пятница", "Суббота", "Воскресенье"];
function dayName(number) {
  return names[number];
}

console.log(dayName(1));
// → Вторник
```

Функция dayName – часть интерфейса модуля, а переменная names – нет. Но хотелось бы не загрязнять глобальное пространство имён.

Можно сделать так:

```
var dayName = function() {
  var names = ["Понедельник", "Вторник", "Среда", "Четве
рг", "Пятница", "Суббота", "Воскресенье"];
  return function(number) {
    return names[number];
  };
}();

console.log(dayName(3));
// → Четверг
```

Теперь names – локальная переменная безымянной функции. Функция создаётся и сразу вызывается, а её возвращаемое значение (уже нужная нам функция dayName) хранится в переменной. Мы можем написать

много страниц кода в функции, объявить там сотню переменных, и все они будут внутренними для нашего модуля, а не для внешнего кода.

Подобный шаблон можно использовать для изолирования кода. Следующий модуль пишет в консоль значение, но не предоставляет никаких значений для использования другими модулями.

```
(function() {
  function square(x) { return x * x; }
  var hundred = 100;

  console.log(square(hundred));
})();
// → 10000
```

Этот код выводит квадрат сотни, но в реальности это мог бы быть модуль, добавляющий метод к какому-то прототипу, или настраивающий виджет на веб-странице. Он обёрнут в функцию для предотвращения загрязнения глобальной ОВ используемыми им переменными.

А зачем мы заключили функцию в круглые скобки? Это связано с глюком синтаксиса JavaScript. Если выражение начинается с ключевого слова function, это функциональное выражение. А если инструкция начинается с function, это объявление функции, которое требует названия, и, так как это не выражение, не может

быть вызвано при помощи скобок () после неё. Можно представлять себе заключение в скобки как трюк, чтобы функция принудительно интерпретировалась как выражение.

Объекты в качестве интерфейсов

Представьте, что нам надо добавить ещё одну функцию в наш модуль «день недели». Мы уже не можем возвращать функцию, а должны завернуть две функции в объект.

```
var weekDay = function() {
  var names = ["Понедельник", "Вторник", "Среда", "Четве
рг", "Пятница", "Суббота", "Воскресенье"];
  return {
    name: function(number) { return names[number]; },
    number: function(name) { return names.indexOf(name);
 }
  };
}();

console.log(weekDay.name(weekDay.number("Воскресенье")))
;
// → Воскресенье
```

Когда модуль большой, собирать все возвращаемые значения в объект в конце функции неудобно, потому что многие возвращаемые функции будут большими, и вам было бы удобнее их записывать где-то в другом месте, рядом со связанным с ними кодом. Удобно объявить объект (обычно называемый exports) и добавлять к нему свойства каждый раз, когда нам надо что-то экспортировать. В следующем примере функция модуля принимает объект интерфейса как аргумент, позволяя коду снаружи функции создать его и сохранить в переменной. Снаружи функции this ссылается на объект глобальной области видимости.

```
(function(exports) {
  var names = ["Понедельник", "Вторник", "Среда", "Четве
рг", "Пятница", "Суббота", "Воскресенье"];

  exports.name = function(number) {
    return names[number];
  };
  exports.number = function(name) {
    return names.indexOf(name);
  };
})(this.weekDay = {});

console.log(weekDay.name(weekDay.number("Четверг")));
// → Четверг
```

Отсоединяемся от глобальной области видимости

Такой шаблон часто используется в модулях JavaScript, предназначающихся для браузера. Модуль возьмёт одну глобальную переменную и обернёт свой код в функцию, чтобы у него было своё личное пространство имён. Но с этим шаблоном бывают проблемы, когда много модулей требуют одно и то же имя, или когда вам надо загрузить две версии модуля одновременно.

Подкрутив кое-что, мы можем сделать систему, разрешающую одному модулю обращаться к интерфейсному объекту другого, без выхода в глобальную ОВ. Наша цель – функция require, которая, получая имя модуля, загрузит его файл (с диска или из сети, в зависимости от платформы) и вернёт соответствующее значение с интерфейсом.

Этот подход решает проблемы, упомянутые ранее, и у него есть ещё одно преимущество – зависимости вашей программы становятся явными, и поэтому сложнее случайно вызвать ненужный вам модуль без чёткого его объявления.

Нам понадобятся две вещи. Во-первых, функция readFile, возвращающая содержимое файла в виде строки. В стандартном JavaScript такой функции нет, но разные окружения, такие как браузер или Node.js, предоставляют свои способы доступа к файлам. Пока притворимся, что у нас есть такая функция. Во-вторых, нам нужна возможность выполнить содержимое этой строки как код.

Выполняем данные как код

Есть несколько способов получить данные (строку кода) и выполнить их как часть текущей программы.

Самый очевидный – оператор eval, который выполняет строку кода в текущем окружении. Это плохая идея – он нарушает некоторые свойства окружения, которые обычно у него есть, например изоляция от внешнего мира.

```
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
```

Способ лучше – использовать конструктор Function. Он принимает два аргумента – строку, содержащую список имён аргументов через запятую, и строку, содержащую тело функции.

```
var plusOne = new Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

Это то, что нам надо. Мы обернём код модуля в функцию, и её область видимости станет областью видимости нашего модуля.

Require

Вот минимальная версия функции require:

```
function require(name) {
  var code = new Function("exports", readFile(name));
  var exports = {};
  code(exports);
  return exports;
}

console.log(require("weekDay").name(1));
// → Вторник
```

Так как конструктор new Function оборачивает код модуля в функцию, нам не надо писать функцию, оборачивающую пространство имён, внутри самого модуля. А так как exports является аргументом функции модуля, модулю не нужно его объявлять. Это убирает много мусора из нашего модуля-примера.

```
var names = ["Понедельник", "Вторник", "Среда", "Четверг"
, "Пятница", "Суббота", "Воскресенье"];

exports.name = function(number) {
  return names[number];
};
exports.number = function(name) {
  return names.indexOf(name);
};
```

При использовании такого шаблона модуль обычно начинается с объявления нескольких переменных, которые загружают модули, от которых он зависит.

```
var weekDay = require("weekDay");
var today = require("today");

console.log(weekDay.name(today.dayNumber()));
```

У такого простого варианта require есть недостатки. Во-первых, он загрузит и выполнит модуль каждый раз, когда его грузят через require – если у нескольких модулей есть

одинаковые зависимости, или вызов require находится внутри функции, которая вызывается многократно, будет потеряно время и энергия.

Это можно решить, храня уже загруженные модули в объекте, и возвращая существующее значение, когда он грузится несколько раз.

Вторая проблема – модуль не может экспортировать переменную напрямую, только через объект export. К примеру, модулю может потребоваться экспортировать только конструктор объекта, объявленного в нём. Сейчас это невозможно, поскольку require всегда использует объект exports в качестве возвращаемого значения.

Традиционное решение – предоставить модули с другой переменной, module, которая является объектом со свойством exports. Оно изначально указывает на пустой объект, созданный require, но может быть перезаписано другим значением, чтобы экспортировать что-либо ещё.

```
function require(name) {
  if (name in require.cache)
    return require.cache[name];

  var code = new Function("exports, module", readFile(na
me));
  var exports = {}, module = {exports: exports};
  code(exports, module);

  require.cache[name] = module.exports;
  return module.exports;
}
require.cache = Object.create(null);
```

Сейчас у нас есть система модулей, использующих одну глобальную переменную require, чтобы позволять модулям искать и использовать друг друга без выхода в глобальную область видимости.

Такой стиль системы модулей называется CommonJS, по имени псевдостандарта, который первым его описал. Он встроен в систему Node.js. Настоящие реализации делают гораздо больше описанного мною. Главное, что у них есть более умный способ перехода от имени модуля к его коду, который разрешает загружать модули по относительному пути к файлу, или же по имени модуля, указывающему на локально установленные модули.

Медленная загрузка модулей

Хотя и возможно использовать стиль CommonJS для браузера, но он не очень подходит для этого. Загрузка файла из Сети происходит медленнее, чем с жёсткого диска. Пока скрипт в браузере работает, на сайте ничего другого не происходит (по причинам, которые станут ясны к 14 главе). Значит, если бы каждый вызов require скачивал что-то с дальнего веб-сервера, страница бы зависла на очень долгое время при загрузке.

Можно обойти это, запуская программу типа Browserify с вашим кодом перед выкладыванием её в веб. Она просмотрит все вызовы require, обработает все зависимости и соберёт нужный код в один большой файл. Веб-сайт просто грузит этот файл и получает все необходимые модули.

Второй вариант – оборачивать код модуля в функцию, чтобы загрузчик модулей сначала грузил зависимости в фоне, а потом вызывал функцию, инициализирующую модуль, после загрузки зависимостей. Этим занимается система AMD (асинхронное определение модулей).

Наша простая программа с зависимостями выглядела бы в AMD так:

```
define(["weekDay", "today"], function(weekDay, today) {
  console.log(weekDay.name(today.dayNumber()));
});
```

Функция define здесь самая важная. Она принимает массив имён модулей, а затем функцию, принимающую один аргумент для каждой из зависимостей. Она загрузит зависимости (если они не были загружены ранее) в фоне, позволяя странице работать, пока файлы скачиваются. Когда всё загружено, define вызывает данную ему функцию, с интерфейсами этих зависимостей в качестве аргументов.

Загруженные таким образом модули должны содержать вызовы define. В качестве их интерфейса используется то, что было возвращено функцией, переданной в define. Вот модуль weekDay:

```
define([], function() {
  var names = ["Понедельник", "Вторник", "Среда", "Четве
рг", "Пятница", "Суббота", "Воскресенье"];
  return {
    name: function(number) { return names[number]; },
    number: function(name) { return names.indexOf(name);
 }
  };
});
```

Чтобы показать минимальную реализацию define, притворимся, что у нас есть функция backgroundReadFile, которая принимает имя файла и функцию, и вызывает эту

функцию с содержимым этого файла, как только он будет загружен. (В главе 17 будет объяснено, как написать такую функцию).

Чтоб отслеживать модули, пока они загружаются, define использует объекты, описывающие состояние модулей, сообщает нам, доступны ли они уже, и предоставляет их интерфейс по доступности.

Функция getModule принимает имя и возвращает такой объект, и убеждается в том, что модуль поставлен в очередь загрузки. Она использует кеширующий объект, чтобы не грузить один модуль дважды.

```
var defineCache = Object.create(null);
var currentMod = null;

function getModule(name) {
  if (name in defineCache)
    return defineCache[name];

  var module = {exports: null,
                loaded: false,
                onLoad: []};
  defineCache[name] = module;
  backgroundReadFile(name, function(code) {
    currentMod = module;
    new Function("", code)();
  });
  return module;
}
```

Мы предполагаем, что загружаемый файл тоже содержит вызов define. Переменная currentMod используется, чтобы сообщить этому вызову о загружаемом объекте модуля, чтобы тот смог обновить этот объект после загрузки. Мы ещё вернёмся к этому механизму.

Функция define сама использует getModule для загрузки или создания объектов модулей для зависимостей текущего модуля. Её задача – запланировать запуск функции moduleFunction (содержащей сам код модуля) после загрузки зависимостей. Для этого она определяет функцию whenDepsLoaded, добавляемую в массив onLoad, содержащий все пока ещё не загруженные зависимости. Эта функция сразу прекращает работу, если есть ещё незагруженные зависимости, так что она выполнит свою работу только раз, когда последняя зависимость загрузится. Она также вызывается сразу из самого define, в случае когда никакие зависимости не нужно грузить.

```
function define(depNames, moduleFunction) {
  var myMod = currentMod;
  var deps = depNames.map(getModule);

  deps.forEach(function(mod) {
    if (!mod.loaded)
      mod.onLoad.push(whenDepsLoaded);
  });

  function whenDepsLoaded() {
    if (!deps.every(function(m) { return m.loaded; }))
      return;

    var args = deps.map(function(m) { return m.exports;
});
    var exports = moduleFunction.apply(null, args);
    if (myMod) {
      myMod.exports = exports;
      myMod.loaded = true;
      myMod.onLoad.every(function(f) { f(); });
    }
  }
  whenDepsLoaded();
}
```

Когда все зависимости доступны, whenDepsLoaded вызывает функцию, содержащую модуль, передавая в виде аргументов интерфейсы зависимостей.

Первое, что делает define, это сохраняет значение currentMod, которое было у него при вызове, в переменной myMod. Вспомните, что getModule прямо

перед исполнением кода модуля сохранил соответствующий объект модуля в currentMod. Это позволяет whenDepsLoaded хранить возвращаемое значение функции модуля в свойстве exports этого модуля, установить свойство loaded модуля в true, и вызвать все функции, ждавшие загрузки модуля.

Этот код изучать тяжелее, чем функцию require. Его выполнение идёт не по простому и предсказуемому пути. Вместо этого, несколько операций должны быть выполнены в неопределённые моменты в будущем, что затрудняет изучения того, как выполняется этот код.

Настоящая реализация AMD гораздо умнее подходит к превращению имён модулей в URL и более надёжна, чем показано в примере. Проект RequireJS предоставляет популярную реализацию такого стиля загрузчика модулей.

Разработка интерфейса

Разработка интерфейсов – один из самых тонких моментов в программировании. Любую нетривиальную функциональность можно реализовать множеством способов. Поиск работающего способа требует проницательности и предусмотрительности.

Лучший способ познать значимость хорошего интерфейса – использовать много интерфейсов. Некоторые будут плохие, некоторые хорошие. Опыт покажет вам, что работает, а что – нет. Никогда не принимайте как должное плохой интерфейс. Исправьте его, или заключите в другой интерфейс, который лучше вам подходит.

Предсказуемость

Если программист может предсказать, как работает ваш интерфейс, ему не придётся часто отвлекаться и смотреть подсказку по его использованию. Постарайтесь следовать общепринятым соглашениям. Если есть модуль или часть языка JavaScript, которая делает что-то похожее на то, что вы пытаетесь реализовать – будет неплохо, если ваш интерфейс будет напоминать существующий. Таким образом, он будет привычен для людей, знакомых с существующим интерфейсом.

В поведении вашего кода предсказуемость также важна. Вас может постичь искушение сделать интерфейс слишком заумным якобы потому, что его удобнее использовать. К примеру, вы можете принимать любые виды типов и комбинаций аргументов и проделывать с ними «то, что надо». Или предоставлять десятки специализированных функций, которые предлагают

незначительно отличающуюся функциональность. Это может сделать код, опирающийся на ваш интерфейс, немного короче, зато затруднить людям, работающим с ним, строить чёткую мысленную модель работы вашего модуля.

Компонуемость

Старайтесь использовать в интерфейсах настолько простые структуры данных, насколько это возможно. Делайте так, чтобы функции выполняли простые и понятные вещи. Если это применимо, делайте функции чистыми (см. Главу 3).

К примеру, частенько модули предлагают свою версию массивоподобных коллекций объектов со своим интерфейсом для подсчёта и извлечения элементов. У таких объектов нет методов map или forEach, и никакая функция, ожидающая настоящий массив, не сможет с ними работать. Это пример плохой компонуемости – модуль нельзя легко скомпоновать с другим кодом.

Примером может служить модуль для орфографической проверки текста, который может пригодиться в текстовом редакторе. Проверочный модуль можно сделать таким, чтобы он работал с любыми сложными структурами, используемыми самим редактором, и вызывал

внутренние функции редактора для предоставления пользователю выбора вариантов написания. Если вы поступите таким образом, модуль нельзя будет использовать с другими программами. С другой стороны, если мы определим интерфейс модуля проверки, который принимает простую строку и возвращает позицию, на которой в строке есть возможная ошибка, а впридачу – массив предлагаемых поправок, тогда у нас будет интерфейс, который можно скомпоновать с другими системами, потому что строки и массивы всегда доступны в JavaScript.

Многослойные интерфейсы

Разрабатывая интерфейс для сложной системы (к примеру, отправка емейл), часто приходишь к дилемме. С одной стороны, не нужно перегружать пользователя интерфейса деталями. Не надо заставлять их изучать его 20 минут перед тем, как они смогут отправить емейл. С другой стороны, не хочется и прятать все детали – когда людям надо сделать что-то сложное при помощи вашего модуля, у них должна быть такая возможность.

Часто приходится предлагать два интерфейса: детализированный низкоуровневый для сложных ситуаций, и простой высокоуровневый для обычного

использования. Второй можно построить на основе первого. В модуле для отправки емейлов высокоуровневый интерфейс может быть просто функцией, которая принимает сообщение, адрес получателя и отправителя, и отправляет письмо. Низкоуровневый должен давать доступ к заголовкам, приложенным файлам, HTML письмам и т.д.

Итог

Модули позволяют структурировать большие программы, разделяя код по разным файлам и пространствам имён. Если обеспечить их хорошо разработанными интерфейсами, их будет просто использовать, применять в других проектах и продолжать использовать при развитии и эволюции самого проекта.

Хотя JavaScript совершенно не помогает делать модули, его гибкие функции и объекты позволяют сделать достаточно неплохую систему модулей. Область видимости функций используется как внутреннее пространство имён модуля, а объекты используются для хранения наборов переменных.

Есть два популярных подхода к использованию модулей. Один – CommonJS, построенный на функции require, которая вызывает модули по имени и возвращает их

интерфейс. Другой – AMD, использующий функцию define, принимающую массив имён модулей и, после их загрузки, исполняющую функцию, аргументами которой являются их интерфейсы.

Упражнения

Названия месяцев

Напишите простой модуль типа weekday, преобразующий номера месяцев (начиная с нуля) в названия и обратно. Выделите ему собственное пространство имён, т.к. ему потребуется внутренний массив с названиями месяцев, и используйте чистый JavaScript, без системы загрузки модулей.

```
// Ваш код

console.log(month.name(2));
// → March
console.log(month.number("November"));
// → 10
```

Вернёмся к электронной жизни

Надеюсь, что глава 7 ещё не стёрлась из вашей памяти. Вернитесь к разработанной там системе и предложите способ разделения кода на модули. Чтобы освежить вам память – вот список функций и типов, по порядку появления:

```
Vector
Grid
directions
directionNames
randomElement
BouncingCritter
elementFromChar
World
charFromElement
Wall
View
WallFollower
dirPlus
LifelikeWorld
Plant
PlantEater
SmartPlantEater
Tiger
```

Не надо создавать слишком много модулей. Книга, в которой на каждой странице была бы новая глава, действовала бы вам на нервы (хотя бы потому, что всё место съедали бы заголовки). Не нужно делать десять файлов для одного мелкого проекта. Рассчитывайте на 3-5 модулей.

Некоторые функции можно сделать внутренними, недоступными из других модулей. Правильного варианта здесь не существует. Организация модулей – вопрос вкуса.

Круговые зависимости

Запутанная тема в управлении зависимостями – круговые зависимости, когда модуль А зависит от Б, а Б зависит от А. Многие системы модулей это просто запрещают. Модули CommonJS допускают ограниченный вариант: это работает, пока модули не заменяют объект exports, существующий по-умолчанию, другим значением, и начинают использовать интерфейсы друг друга только после окончания загрузки.

Можете ли вы придумать способ, который позволил бы воплотить систему поддержки таких зависимостей? Посмотрите на определение require и подумайте, что нужно сделать этой функции для этого.

Проект: язык программирования

То, что проверяет и определяет смысл выражений в языке программирования, является в свою очередь просто программой.

Хэл Абельсон и Жеральд Сасман, «Структура и интерпретация компьютерных программ».

Когда ученик спросил учителя о природе цикла Данных и Контроля, Юань-Ма ответил: «Подумай о компиляторе, компилирующем самого себя».

Мастер Юань-Ма, «Книга программирования»

Создать свой язык программирования удивительно легко (пока вы не ставите запредельных целей) и довольно поучительно.

Главное, что я хочу продемонстрировать в этой главе – в построении языка нет никакой магии. Мне часто казалось, что некоторые человеческие изобретения настолько сложны и заумны, что мне их никогда не понять. Однако после небольшого самообразования и ковыряния такие штуки часто оказываются довольно обыденными.

Мы построим язык программирования Egg (Яйцо). Он будет небольшим, простым, но достаточно мощным для выражения любых расчётов. Он также будет осуществлять простые абстракции, основанные на функциях.

Разбор (parsing)

То, что лежит на поверхности языка – синтаксис, запись. Грамматический анализатор, или парсер – программа, читающая кусок текста и выдающая структуру данных, описывающую структуры программы, содержавшейся в тексте. Если текст не описывает корректную программу, парсер должен пожаловаться и указать на ошибку.

У нашего языка будет простой и однородный синтаксис. В Egg всё будет являться выражением. Выражение может быть переменной, число, строка или приложение. Приложения используются для вызова функций и конструкций типа if или while.

Для упрощения парсинга строки в Egg не будут поддерживать обратных слешей и подобных вещей. Строка – просто последовательность символов, не являющихся двойными кавычками, заключённая в двойные кавычки. Число – последовательность цифр.

Имена переменных могут состоять из любых символов, не являющихся пробелами и не имеющих специального значения в синтаксисе.

Приложения записываются так же, как в JS — при помощи скобок после выражения и с любым количеством аргументов в скобках, разделённых запятыми.

```
do(define(x, 10),
   if(>(x, 5)),
      print("много"),
      print("мало"))
```

Однородность языка означает, что то, что в JS является операторами, применяется так же, как и остальные функции. Так как в синтаксисе нет концепции блоков, нам нужна конструкция do для обозначения нескольких вещей, выполняемых последовательно.

Структура данных, описывающая программу, будет состоять из объектов выражений, у каждого из которых будет свойство type, отражающее тип этого выражения и другие свойства, описывающие содержимое.

Выражения типа “value” представляют строки или числа. Их свойство value содержит строку или число, которое они представляют. Выражения типа “word” используются для идентификаторов (имён). У таких объектов есть

свойство name, содержащее имя идентификатора в виде строки. И наконец, выражения “apply” представляют приложения. У них есть свойство “operator”, ссылающееся на применяемое выражение, и свойство “args” с массивом аргументов.

Часть >(x, 5) будет представлена так:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

Такая структура данных называется синтаксическим деревом. Если вы представите объекты в виде точек, а связи между ними в виде линий, то получите древовидную структуру. То, что выражения содержат другие выражения, которые в свою очередь могут содержать свои выражения, сходно с тем, как разветвляются ветки.

Структура синтаксического дерева

Сравните это с парсером, написанным нами для файла настроек в главе 9, у которого была простая структура: он делил ввод на строки и обрабатывал их одну за другой. Там было всего несколько форм, которые разрешено принимать строке.

Здесь нам нужен другой подход. Выражения не разделяются на строчки, и их структура рекурсивна. Выражения-приложения содержат другие выражения. К счастью, эта задача элегантно решается применением рекурсивной функции, отражающей рекурсивность языка.

Мы определяем функцию parseExpression, принимающую строку на вход и возвращающую объект, содержащий структуру данных для выражения с начала строки, вместе с частью строки, оставшейся после парсинга. При разборе подвыражений (таких, как аргумент приложения), эта функция снова вызывается, возвращая выражение аргумента вместе с оставшимся текстом. Тот текст может, в свою очередь, содержать ещё аргументы, или же быть закрывающей скобкой, завершающей список аргументов.

Первая часть парсера:

```
function parseExpression(program) {
  program = skipSpace(program);
  var match, expr;
  if (match = /^"([^"]*)"/.exec(program))
    expr = {type: "value", value: match[1]};
  else if (match = /^\d+\b/.exec(program))
    expr = {type: "value", value: Number(match[0])};
  else if (match = /^[^\s(),"]+/.exec(program))
    expr = {type: "word", name: match[0]};
  else
    throw new SyntaxError("Неожиданный синтаксис: " + pr
ogram);

  return parseApply(expr, program.slice(match[0].length)
);
}

function skipSpace(string) {
  var first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

Поскольку Egg разрешает любое количество пробелов в элементах, нам надо постоянно вырезать пробелы с начала строки. С этим справляется skipSpace.

Пропустив начальные пробелы, parseExpression использует три регулярки для распознавания трёх простых (атомарных) элементов, поддерживаемых языком: строк, чисел и слов. Парсер создаёт разные структуры для разных типов. Если ввод не подходит ни

под одну из форм, это не является допустимым выражением, и он выбрасывает ошибку. SyntaxError – стандартный объект для ошибок, который создаётся при попытке запуска некорректной программы JavaScript.

Мы можем отрезать обработанную часть программы, и передать его, вместе с объектом выражения, в parseApply, определяющая, не является ли выражение приложением. Если так и есть, он парсит список аргументов в скобках.

```
function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(")
    return {expr: expr, rest: program};

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
    var arg = parseExpression(program);
    expr.args.push(arg.expr);
    program = skipSpace(arg.rest);
    if (program[0] == ",")
      program = skipSpace(program.slice(1));
    else if (program[0] != ")")
      throw new SyntaxError("Ожидается ',' or ')'");
  }
  return parseApply(expr, program.slice(1));
}
```

Если следующий символ программы – не открывающая скобка, то это не приложение, и parseApply просто возвращает данное ей выражение.

В ином случае, она пропускает открывающую скобку и создаёт объект синтаксического дерева для этого выражения. Затем она рекурсивно вызывает parseExpression для разбора каждого аргумента, пока не встретит закрывающую скобку. Рекурсия непрямая, parseApply и parseExpression вызывают друг друга.

Поскольку приложение само по себе может быть выражением (multiplier(2)(1)), parseApply должна, после разбора приложения, вызвать себя снова, проверив, не идёт ли далее другая пара скобок.

Вот и всё, что нам нужно для разбора Egg. Мы обернём это в удобную функцию parse, проверяющую, что она дошла до конца строки после разбора выражения (программа Egg – это одно выражение), и это даст нам структуру данных программы.

```
function parse(program) {
  var result = parseExpression(program);
  if (skipSpace(result.rest).length > 0)
    throw new SyntaxError("Неожиданный текст после прогр
аммы");
  return result.expr;
}

console.log(parse("+(a, 10)"));
// → {type: "apply",
//    operator: {type: "word", name: "+"},
//    args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]}
```

Работает! Она не выдаёт полезной информации при ошибке, и не хранит номера строки и столбца, с которых начинается каждое выражение, что могло бы пригодиться при разборе ошибок – но для нас и этого хватит.

Интерпретатор

А что нам делать с синтаксическим деревом программы? Запускать её! Этим занимается интерпретатор. Вы даёте ему синтаксическое дерево и объект окружения, который связывает имена со значениями, а он интерпретирует выражение, представляемое деревом, и возвращает результат.

```
function evaluate(expr, env) {
  switch(expr.type) {
    case "value":
      return expr.value;

    case "word":
      if (expr.name in env)
        return env[expr.name];
      else
        throw new ReferenceError("Неопределённая перемен
ная: " +
                                 expr.name);
    case "apply":
      if (expr.operator.type == "word" &&
          expr.operator.name in specialForms)
        return specialForms[expr.operator.name](expr.arg
s,
                                                 env);
      var op = evaluate(expr.operator, env);
      if (typeof op != "function")
        throw new TypeError("Приложение не является функ
цией.");
      return op.apply(null, expr.args.map(function(arg)
{
        return evaluate(arg, env);
      }));
  }
}

var specialForms = Object.create(null);
```

У интерпретатора есть код для каждого из типов выражений. Для литералов он возвращает их значение. Например, выражение 100 интерпретируется в число 100. У переменной мы должны проверить, определена ли она в окружении, и если да – запросить её значение.

С приложениями сложнее. Если это особая форма типа if, мы ничего не интерпретируем, а просто передаём аргументы вместе с окружением в функцию, обрабатывающую форму. Если это простой вызов, мы интерпретируем оператор, проверяем, что это функция и вызываем его с результатом интерпретации аргументов.

Для представления значений функций Egg мы будем использовать простые значения функций JavaScript. Мы вернёмся к этому позже, когда определим специальную форму fun.

Рекурсивная структура интерпретатора напоминает парсер. Оба отражают структуру языка. Можно было бы интегрировать парсер в интерпретатор и интерпретировать во время разбора, но их разделение делает программу более читаемой.

Вот и всё, что нужно для интерпретации Egg. Вот так просто. Но без определения нескольких специальных форм и добавления полезных значений в окружение, вы с этим языком ничего не сможете сделать.

Специальные формы

Объект specialForms используется для определения особого синтаксиса Egg. Он сопоставляет слова с функциями, интерпретирующими эти специальные формы. Пока он пуст. Давайте добавим несколько форм.

```
specialForms["if"] = function(args, env) {
  if (args.length != 3)
    throw new SyntaxError("Неправильное количество аргум
ентов для if");

  if (evaluate(args[0], env) !== false)
    return evaluate(args[1], env);
  else
    return evaluate(args[2], env);
};
```

Конструкция if языка Egg ждёт три аргумента. Она вычисляет первый, и если результат не false, вычисляет второй. В ином случае вычисляет третий. Этот if больше похож на тернарный оператор ?:. Это выражение, а не инструкция, и она выдаёт значение, а именно, результат второго или третьего выражения.

Egg отличается от JavaScript тем, как он обрабатывает условие if. Он не будет считать ноль или пустую строку за false.

if представлено в виде особой формы а не обычной функции, потому что аргументы функций вычисляются перед вызовом, а if должен интерпретировать один из двух аргументов – второй или третий, в зависимости от значения первого.

Форма для while схожая.

```
specialForms["while"] = function(args, env) {
  if (args.length != 2)
    throw new SyntaxError("Неправильное количество аргум
ентов для while");

  while (evaluate(args[0], env) !== false)
    evaluate(args[1], env);

  // Поскольку undefined не задано в Egg,
  // за отсутствием осмысленного результата возвращаем f
alse
  return false;
};
```

Ещё одна основная часть языка – do, выполняющий все аргументы сверху вниз. Его значение – это значение, выдаваемое последним аргументом.

```
specialForms["do"] = function(args, env) {
  var value = false;
  args.forEach(function(arg) {
    value = evaluate(arg, env);
  });
  return value;
};
```

Чтобы создавать переменные и давать им значения, мы создаём форму define. Она ожидает word в качестве первого аргумента, и выражение, производящее значение, которое надо присвоить этому слову в качестве второго. define, как и всё, является выражением, поэтому оно должно возвращать значение. Пусть оно возвращает присвоенное значение (прям как оператор = в JavaScript).

```
specialForms["define"] = function(args, env) {
  if (args.length != 2 || args[0].type != "word")
    throw new SyntaxError("Bad use of define");
  var value = evaluate(args[1], env);
  env[args[0].name] = value;
  return value;
};
```

Окружение

Окружение, принимаемое интерпретатором — это объект со свойствами, чьи имена соответствуют именам переменных, а значения – значениям этих переменных. Давайте определим объект окружения, представляющий глобальную область видимости.

Для использования конструкции if мы должны создать булевские значения. Так как их всего два, особый синтаксис для них не нужен. Мы просто делаем две переменные со значениями true и false.

```
var topEnv = Object.create(null);

topEnv["true"] = true;
topEnv["false"] = false;
```

Теперь мы можем вычислить простое выражение, меняющее булевское значение на обратное.

```
var prog = parse("if(true, false, true)");
console.log(evaluate(prog, topEnv)); // → false
```

Для поддержки простых арифметических операторов и сравнения мы добавим несколько функций в окружение. Для упрощения кода мы будем использовать new Function для создания набора функций-операторов в цикле, а не определять их все по отдельности.

```
["+", "-", "*", "/", "==", "<", ">"].forEach(function(op
) {
  topEnv[op] = new Function("a, b", "return a " + op + "
 b;");
});
```

Также пригодится способ вывода значений, так что мы обернём console.log в функцию и назовём её print.

```
topEnv["print"] = function(value) {
  console.log(value);
  return value;
};
```

Это даёт нам достаточно элементарных инструментов для написания простых программ. Следующая функция run даёт удобный способ записи и запуска. Она создаёт свежее окружение, парсит и разбирает строчки, которые мы ей передаём, так, как будто они являются одной программой.

```
function run() {
  var env = Object.create(topEnv);
  var program = Array.prototype.slice
    .call(arguments, 0).join("\n");
  return evaluate(parse(program), env);
}
```

Использование Array.prototype.slice.call – уловка для превращения объекта, похожего на массив, такого как аргументы, в настоящий массив, чтобы мы могли применить к нему join. Она принимает все аргументы, переданные в run, и считает, что все они – строчки программы.

```
run("do(define(total, 0),",
    "   define(count, 1),",
    "   while(<(count, 11),",
    "         do(define(total, +(total, count)),",
    "            define(count, +(count, 1)))),",
    "   print(total))");
// → 55
```

Эту программу мы видели уже несколько раз – она подсчитывает сумму чисел от 1 до 10 на языке Egg. Она уродливее эквивалентной программы на JavaScript, но не так уж и плоха для языка, заданного менее чем 150 строчками кода.

Функции

Язык программирования без функций – плохой язык.

К счастью, несложно добавить конструкцию fun, которая расценивает последний аргумент как тело функции, а все предыдущие – имена аргументов функции.

```
specialForms["fun"] = function(args, env) {
  if (!args.length)
    throw new SyntaxError("Функции нужно тело");
  function name(expr) {
    if (expr.type != "word")
      throw new SyntaxError("Имена аргументов должны быт
ь типа word");
    return expr.name;
  }
  var argNames = args.slice(0, args.length - 1).map(name
);
  var body = args[args.length - 1];

  return function() {
    if (arguments.length != argNames.length)
      throw new TypeError("Неверное количество аргументо
в");
    var localEnv = Object.create(env);
    for (var i = 0; i < arguments.length; i++)
      localEnv[argNames[i]] = arguments[i];
    return evaluate(body, localEnv);
  };
};
```

У функций в Egg своё локальное окружение, как и в JavaScript. Мы используем Object.create для создания нового объекта, имеющего доступ к переменным во внешнем окружении (своего прототипа), но он также может содержать новые переменные, не меняя внешней области видимости.

Функция, созданная формой fun, создаёт своё локальное окружение и добавляет к нему переменные-аргументы. Затем она интерпретирует тело в этом окружении и возвращает результат.

```
run("do(define(plusOne, fun(a, +(a, 1))),",
    "   print(plusOne(10)))");
// → 11

run("do(define(pow, fun(base, exp,",
    "     if(==(exp, 0),",
    "        1,",
    "        *(base, pow(base, -(exp, 1))))),",
    "   print(pow(2, 10)))");
// → 1024
```

Компиляция

Мы с вами построили интерпретатор. Во время интерпретации он работает с представлением программы, созданным парсером.

Компиляция – добавление ещё одного шага между парсером и запуском программы, которая превращает в программу в нечто, что можно выполнять более эффективно, путём проделывания большинства работы заранее. К примеру, в хорошо организованных языках при каждом использовании переменной очевидно, к какой

переменной обращаются, даже без запуска программы. Это можно использовать, чтобы не искать переменную по имени каждый раз, когда к ней обращаются, а напрямую вызывать её из какой-то заранее определённой области памяти.

По традиции компиляция также превращает программу в машинный код – сырой формат, пригодный для исполнения процессором. Но каждый процесс превращения программы в другой вид, по сути, является компиляцией.

Можно было бы создать другой интерпретатор Egg, который сначала превращает программу в программу на языке JavaScript, использует new Function для вызова компилятора JavaScript и возвращает результат. При правильной реализации Egg выполнялся бы очень быстро при относительно простой реализации.

Если вам это интересно, и вы хотите потратить на это время, я поощряю вас попробовать сделать такой компилятор в качестве упражнения.

Мошенничество

Когда мы определяли if и while, вы могли заметить, что они представляли собой простые обёртки вокруг if и while в JavaScript. Значения в Egg – также обычные значения JavaScript.

Сравнивая реализацию Egg, построенную на JavaScript, с объёмом работы, необходимой для создания языка программирования непосредственно на машинном языке, то разница становится огромной. Тем не менее, этот пример, надеюсь, даёт вам представление о работе языков программирования.

И когда вам надо что-то сделать, смошенничать будет более эффективно, нежели делать всё с нуля самому. И хотя игрушечный язык ничем не лучше JavaScript, в некоторых ситуациях написание своего языка помогает быстрее сделать работу.

Такой язык не обязан напоминать обычный ЯП. Если бы JavaScript не содержал регулярных выражений, вы могли бы написать свои парсер и интерпретатор для такого суб-языка.

Или представьте, что вы строите гигантского робота-динозавра и вам нужно запрограммировать его поведение. JavaScript – не самый эффективный способ сделать это. Можно вместо этого выбрать язык примерно такого свойства:

```
behavior walk
  perform when
    destination ahead
  actions
    move left-foot
    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets
```

Обычно это называют языком для выбранной области (domain-specific language) – язык, специально предназначенный для работы в узком направлении. Такой язык может быть более выразительным, чем язык общего назначения, потому что он разработан для выражения именно тех вещей, которые надо выразить в этой области – и больше ничего.

Упражнения

Массивы

Добавьте поддержку массивов в Egg. Для этого добавьте три функции в основную область видимости: array(...) для создания массива, содержащего значения аргументов, length(array) для возврата длины массива и element(array, n) для возврата n-ного элемента.

```
// Добавьте кода
topEnv["array"] = "...";
topEnv["length"] = "...";
topEnv["element"] = "...";

run("do(define(sum, fun(array,",
    "     do(define(i, 0),",
    "        define(sum, 0),",
    "        while(<(i, length(array)),",
    "          do(define(sum, +(sum, element(array, i)))
,",
    "             define(i, +(i, 1)))),",
    "        sum))),",
    "   print(sum(array(1, 2, 3))))");
// → 6
```

Замыкания

Способ определения fun позволяет функциям в Egg замыкаться вокруг окружения, и использовать локальные переменные в теле функции, которые видны во время определения, точно как в функциях JavaScript.

Следующая программа иллюстрирует это: функция f возвращает функцию, добавляющую её аргумент к аргументу f, то есть, ей нужен доступ к локальной области видимости внутри f для использования переменной a.

```
run("do(define(f, fun(a, fun(b, +(a, b)))),",
    "   print(f(4)(5)))");
// → 9
```

Объясните, используя определение формы fun, какой механизм позволяет этой конструкции работать.

Комментарии

Хорошо было бы иметь комментарии в Egg. К примеру, мы могли бы игнорировать оставшуюся часть строки, встречая символ “#” – так, как это происходит с “//” в JS.

Большие изменения в парсере делать не придётся. Мы просто поменяем skipSpace, чтобы она пропускала комментарии, будто они являются пробелами – и во всех местах, где вызывается skipSpace, комментарии тоже будут пропущены. Внесите это изменение.

```
// Поменяйте старую функцию
function skipSpace(string) {
  var first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}

console.log(parse("# hello\nx"));
// → {type: "word", name: "x"}

console.log(parse("a # one\n   # two\n()"));
// → {type: "apply",
//    operator: {type: "word", name: "a"},
//    args: []}
```

Чиним область видимости

Сейчас мы можем присвоить переменной значение только через define. Эта конструкция работает как при присвоении старым переменным, так и при создании новых.

Эта неоднозначность приводит к проблемам. Если вы пытаетесь присвоить новое значение нелокальной переменной, вместо этого вы определяете локальную с таким же именем. (Некоторые языки так и делают, но мне это всегда казалось дурацким способом работы с областью видимости).

Добавьте форму set, схожую с define, которая присваивает переменной новое значение, обновляя переменную во внешней области видимости, если она не задана в локальной. Если переменная вообще не задана, швыряйте ReferenceError (ещё один стандартный тип ошибки).

Техника представления областей видимости в виде простых объектов, до сего момента бывшая удобной, теперь будет вам мешать. Вам может понадобиться функция Object.getPrototypeOf, возвращающая прототип объекта. Также помните, что область видимости не наследуется от Object.prototype, поэтому если вам надо вызвать на них hasOwnProperty, придётся использовать такую неуклюжую конструкцию:

```
Object.prototype.hasOwnProperty.call(scope, name);
```

Это вызывает метод hasOwnProperty прототипа Object и затем вызывает его на объекте scope.

```
specialForms["set"] = function(args, env) {
  // Ваш код
};

run("do(define(x, 4),",
    "   define(setx, fun(val, set(x, val))),",
    "   setx(50),",
    "   print(x))");
// → 50
run("set(quux, true)");
// → Ошибка вида ReferenceError
```

JavaScript и браузер

Браузер – крайне враждебная программная среда

Дуглас Крокфорд, «Язык программирования JavaScript» (видеолекция)

Следующая часть книги расскажет о веб-браузерах. Без них не было бы JavaScript. А если бы и был, никто бы не обратил на него внимания.

Технологии веба с самого начала были децентрализованными – не только технически, но и с точки зрения их эволюции. Различные разработчики браузеров добавляли новую функциональность «по случаю», непродуманно, и часто эта функциональность обретала поддержку в других браузерах и становилась стандартом.

Это и благословление и проклятие. С одной стороны, здорово не иметь контролирующего центра, чтобы технология развивалась различными сторонами, иногда сотрудничающими, иногда конкурирующими. С другой – бессистемное развитие языка привело к тому, что

результат не является ярким примером внутренней согласованности. Некоторые части привносят путаницу и беспорядок.

Сети и интернет

Компьютерные сети появились в 1950-х. Если вы проложите кабель между двумя или несколькими компьютерами и разрешите им передавать данные, вы может делать много удивительных вещей. А если связь двух машин в одном здании позволяет делать много разного, то связь компьютеров по всей планете должна позволять ещё больше. Технология, позволяющая это сделать, была создана в 1980-х, и получившаяся сеть зовётся интернетом. И она оправдала ожидания.

Компьютер может использовать эту сеть, чтобы кидаться битами в другой компьютер. Чтобы общение вышло эффективным, оба компьютера должны знать, что эти биты означают. Значение любой заданной последовательности битов зависит от того, что пытаются ими выразить, и какой механизм кодирования используется.

Стиль общения по сети описывает сетевой протокол. Есть протоколы для отправки е-мейлов, для получения е-мейлов, для распространения файлов и даже для

контроля над компьютерами, заражёнными вредоносным софтом.

К примеру, простой протокол чата может состоять из одного компьютера, отправляющего биты, представляющие текст «ЧАТ?» на другой, а второго отвечающего текстом «ОК!», для подтверждения того, что он понял протокол. Дальше они могут перейти к отправке друг другу текстов, чтения полученных текстов и вывода их на экран.

Большинство протоколов построено на основе других протоколов. Наш протокол чата из примера рассматривает сеть как потоковое устройство, в которое можно вводить биты и заказывать их приход на конкретный адрес в правильном порядке. А обеспечение этого процесса – само по себе является сложной задачей. Transmission Control Protocol (TCP) – протокол, решающий эту задачу. Все устройства, подключённые к интернету, говорят на нём, и большинство общения в интернете построено на его основе.

Соединение по TCP работает так: один компьютер ждёт, или «слушает», пока другие не начнут с ним говорить. Чтобы можно было слушать разные виды общения в одно и то же время, для каждого из них назначается номер (называемый портом). Большинство протоколов устанавливают порт, используемый по умолчанию. К

примеру, если мы отправляем е-мейл по протоколу SMTP, компьютер, через который мы его шлём, должен слушать порт 25.

Тогда другой компьютер может установить соединение, связавшись с компьютером назначения по правильному порту. Если машина назначения доступна, и она слушает этот порт, соединение устанавливается. Слушающий компьютер зовётся сервером, а соединяющийся – клиентом.

Такое соединение работает как двусторонняя труба, по которой текут биты – обе машины могут помещать в неё данные. Когда биты переданы, другая машина может их прочесть. Это удобная модель. Можно сказать, что TCP обеспечивает абстракцию сети.

Веб

World Wide Web, всемирная паутина (это не то же самое, что весь интернет в целом) – набор протоколов и форматов, позволяющий нам посещать странички через браузер. Web (рус. «паутина») в названии обозначает, что страницы можно легко связать друг с другом, в результате чего образуется гигантская сеть-паутина, по которой движутся пользователи.

Чтобы добавить в Веб содержимое, вам нужно соединить машину с интернетом и заставить её слушать 80 порт, используя протокол передачи гипертекста, Hypertext Transfer Protocol (HTTP). Он позволяет другим компьютерам запрашивать документы по сети.

Каждый документ имеет имя в виде универсального локатора ресурсов, Universal Resource Locator (URL), который выглядит примерно так:

```
http://eloquentjavascript.net/12_browser.html
|     |                       |               |
протокол      сервер               путь
```

Первая часть говорит нам, что URL использует протокол HTTP (в отличие от, скажем, зашифрованного HTTP, который записывается как https://). Затем идёт часть, определяющая, с какого сервера мы запрашиваем документ. Последняя – строка пути, определяющая конкретный документ или ресурс.

У каждой машины, присоединённой к интернету, есть свой адрес IP, который выглядит как 37.187.37.82. Его иногда можно использовать вместо имени сервера в URL. Но цифры сложнее запоминать и печатать, чем имена – поэтому обычно вы регистрируете доменное имя, которое указывает на конкретную машину (или набор машин). Я зарегистрировал eloquentjavascript.net, указывающий на

IP-адрес машины, которую я контролирую, поэтому можно использовать этот адрес для предоставления веб-страниц.

Если вы введёте указанный URL в адресную строку браузера, он попробует запросить и показать документ, находящийся по этому URL. Во-первых, браузеру надо выяснить, куда ссылается домен eloquentjavascript.net. Затем, используя протокол HTTP, он соединяется с сервером по этому адресу, и спрашивает его ресурс по имени /12_browser.html

В главе 17 мы подробнее рассмотрим протокол HTTP.

HTML

HTML, или язык разметки гипертекста, Hypertext Markup Language – формат документа, использующийся для веб-страниц. HTML содержит текст и теги, придающие тексту структуру, описывающие такие вещи, как ссылки, параграфы и заголовки.

Простой HTML документ может выглядеть так:

```
<!doctype html>
<html>
  <head>
    <title>Моя домашняя страничка</title>
  </head>
  <body>
    <h1> Моя домашняя страничка </h1>
    <p>Привет, я Марийн и это моя домашняя страничка.</p>

    <p>А ещё я книжку написал! Читайте её
      <a href="http://eloquentjavascript.net">здесь</a>.
</p>
  </body>
</html>
```

Теги, окружённые угловыми скобками < и >, описывают информацию о структуре документа. Всё остальное – просто текст.

Документ начинается с <!doctype html>, и это говорит браузеру, что его надо интерпретировать как современный HTML, в отличие от разных диалектов прошлого.

У HTML документов есть заголовок и тело. Заголовок содержит информацию о документе, а тело – сам документ. В нашем случае мы объявили, что название страницы будет «Моя домашняя страничка», затем

описали документ, содержащий заголовок (`<h1>` , то есть heading 1, заголовок 1. Есть ещё `<h2>` – `<h6>` , заголовки разных размеров) и два параграфа.

У тегов может быть несколько форм. Элемент вроде тела, параграфа и ссылки начинается открывающим тегом `<p>` и заканчивается закрывающим `</p>` . Некоторые открывающие теги, типа ссылки `<a>` , содержат дополнительную информацию в виде имя=”значение”. Она называется «атрибутами». В нашем случае адрес ссылки задан как `href="http://eloquentjavascript.net"` , где href означает «гипертекстовая ссылка», “hypertext reference”.

Некоторые теги ничего не окружают, и их не надо закрывать. Пример – тег картинки

```
<img src="http://example.com/image.jpg">
```

Чтобы включать в текст документа угловые скобки, нужно пользоваться специальной записью, так как в HTML они имеют особое значение. Открывающая скобка (она же знак «меньше») записывается как `<` («less than», «меньше, чем»), закрывающая — `>` («greater that», «больше, чем»). В HTML амперсанд &, за которым идёт

слово и точка с запятой, зовётся сущностью и заменяется символом, который кодируется этой последовательностью.

Это похоже на обратные слэши, используемые в строках JavaScript. Из-за специального значения амперсанда его самого в текст можно включать в виде `&`. В атрибуте, заключаемом в двойные кавычки, символ кавычек записывается как `"`.

HTML разбирается парсером довольно либерально по отношению к возможным ошибкам. Если какие-то теги пропущены, браузер их воссоздаёт. Как именно это происходит, записано в стандартах, поэтому можно ожидать, что все современные браузеры будут делать это одинаково.

Следующий документ будет обработан так же, как и предыдущий.

```
<!doctype html>

<title>Моя домашняя страничка</title>

<h1> Моя домашняя страничка </h1>
<p>Привет, я Марийн и это моя домашняя страничка.
<p>А ещё я книжку написал! Читайте её
<a href=http://eloquentjavascript.net>here</a>.
```

Отсутствуют теги `<html>`, `<head>` и `<body>`. Браузер знает, что `<title>` должен быть в `<head>`, а `<h1>` — в `<body>`. Кроме того, параграфы не закрыты, поскольку открытие нового параграфа или конец документа означают их принудительное закрытие. Также адрес не заключён в кавычки.

В этой книге мы опустим теги `<html>`, `<head>` и `<body>` для краткости. Но я буду закрывать теги, и заключать атрибуты в кавычки.

Также обычно я буду опускать doctype. Я не советую делать это вам – браузеры иногда могут творить странные вещи, когда вы их опускаете. Считайте, что они присутствуют в примерах по умолчанию.

HTML и JavaScript

В контексте нашей книги самый главный тег HTML — `<script>`. Он позволяет включать в документ программу на JavaScript.

```
<h1>Внимание, тест.</h1>
<script>alert("Привет!");</script>
```

Такой скрипт запустится сразу, как только браузер встретит тег `<script>` при разборе HTML. На странице появится диалог-предупреждение.

Включать большие программы в HTML непрактично. У тега `<script>` есть атрибут src, чтобы запрашивать файл со скриптом (текст, содержащий программу на JavaScript) с адреса URL.

```
<h1>Внимание, тест.</h1>
<script src="code/hello.js"></script>
```

В файле code/hello.js содержится та же простая программа «alert('Привет!');». Когда страница ссылается на другой URL и включает его в себя, браузер подгружает этот файл и включает их в страницу.

Тег script всегда надо закрывать при помощи , даже если он не содержит кода и ссылается на файл скрипта. Если вы забудете это сделать, оставшаяся часть страницы будет обработана как скрипт.

Некоторые атрибуты тоже могут содержать программу JavaScript. У тега (на странице он выглядит как кнопка) есть атрибут onClick, и его содержимое будет запущено, когда по кнопке щёлкают мышкой.

```
<button onclick="alert('Бабах!');">НЕ ЖМИ</button>
```

Заметьте, что я использовал одинарные кавычки для строки в атрибуте onclick, поскольку двойные кавычки уже используются в самом атрибуте. Можно было бы использовать `"` , но это бы затруднило чтение.

Песочница

Запуск скачанных из интернета программ небезопасен. Вы не знаете ничего о тех людях, которые делали посещаемые вами сайты, и они не всегда доброжелательны. Запуская программы злых людей, вы можете заразить компьютер вирусами, потерять свои данные или дать доступ к своим аккаунтам третьим лицам.

Но привлекательность веба в том, что по нему можно сёрфить без обязательного доверия всем посещаемым страницам. Поэтому браузеры сильно ограничивают то, что может сделать программа JavaScript. Она не может открывать файлы на компьютере, или менять что-либо, не связанное со страницей, в которую она встроена.

Изолированное таким образом окружение называется песочницей – в том смысле, что программа безобидно играется в песочнице. Представляйте, однако, эту песочницу как клетку из толстых стальных прутьев.

Сложность в создании песочницы – позволять программам делать достаточно много, чтобы они были полезными, при этом ограничивая их от совершения опасных действий. Много из того, что делает пользователь, например общение с другими серверами или чтение содержимого буфера обмена, можно использовать для нарушения приватности.

Время от времени кто-то придумывает способ обойти ограничения браузера и сделать что-то вредное, от утечки некоей приватной информации до полного контроля над компьютером, где запущен скрипт. Разработчики исправляют эту дырку в браузере, и снова всё хорошо – до появления следующей проблемы, которая, можно надеяться, будет опубликована, и не тайно использоваться правительством или мафией.

Совместимость и браузерные войны

На ранних стадиях развития Веба браузер по имени Mosaic занимал большую часть рынка. Через несколько лет баланс сместился в сторону Netscape, который затем был сильно потеснён браузером Internet Explorer от Microsoft. В любой момент превосходства одного из браузеров его разработчики позволяли себе в

одностороннем порядке изобретать новые свойства веба. Так как большинство людей использовали один и тот же браузер, сайты просто начинали использовать эти свойства, не обращая внимания на остальные браузеры.

Это были тёмные века совместимости, которые иногда называли «войнами браузеров». Веб-разработчики сталкивались с двумя или тремя несовместимыми платформами. Кроме того, браузеры около 2003 года были полны ошибок, причём у каждого они были свои. Жизнь людей, создававших веб-страницы, была тяжёлой.

Mozilla Firefox, некоммерческое ответвление Netscape, бросил вызов гегемонии Internet Explorer в конце 2000-х. Так как Microsoft особо не стремилась к конкуренции, Firefox отобрал солидную часть рынка. Примерно в это время Google представил свой браузер Chrome, а Apple – Safari. Это привело к появлению четырёх основных игроков вместо одного.

У новых игроков были более серьёзные намерения по отношению к стандартам и больше инженерного опыта, что привело к лучшей совместимости и меньшему количеству багов. Microsoft, видя сжатие своей части рынка, приняла эти стандарты. Если вы начинаете изучать веб-разработку сегодня – вам повезло. Последние версии основных браузеров работают одинаково и в них мало ошибок.

Нельзя сказать, что ситуация уже идеальная. Некоторые люди в вебе по причинам инерционности или корпоративных правил используют очень старые браузеры. Пока они не отомрут совсем, написание веб-страниц для них потребует мистических знаний об их недостатках и причудах. Эта книга не про причуды – она представляет современный, разумный стиль веб-программирования.

Document Object Model

Когда вы открываете веб-страницу в браузере, он получает исходный текст HTML и разбирает (парсит) его примерно так, как наш парсер из главы 11 разбирал программу. Браузер строит модель структуры документа и использует её, чтобы нарисовать страницу на экране.

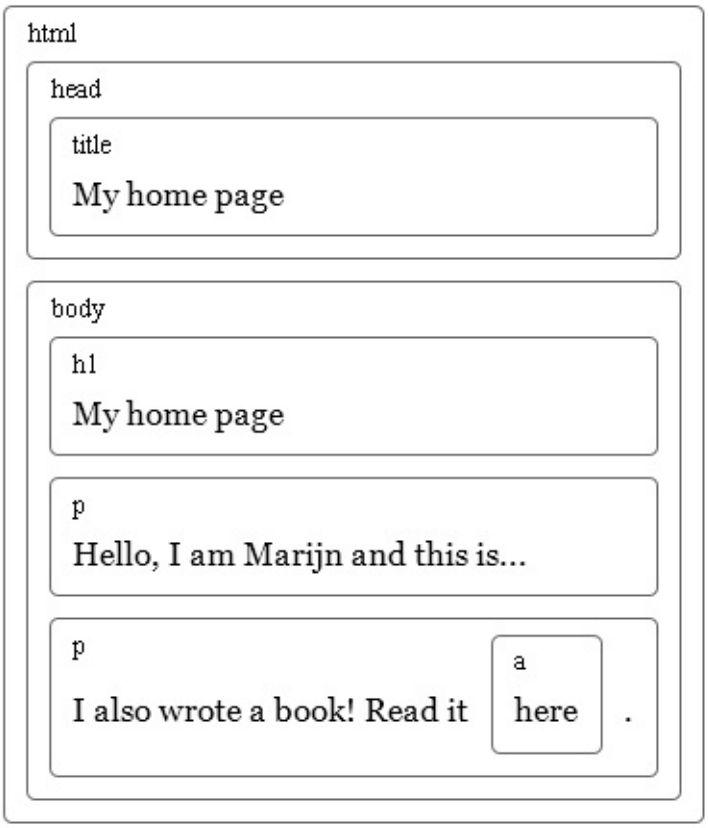
Это представление документа и есть одна из игрушек, доступных в песочнице JavaScript. Вы можете читать её и изменять. Она изменяется в реальном времени – как только вы её подправляете, страница на экране обновляется, отражая изменения.

Структура документа

Можно представить HTML как набор вложенных коробок. Теги вроде `<body>` и `</body>` включают в себя другие теги, которые в свою очередь включают теги, или текст. Вот вам пример документа из предыдущей главы:

```
<!doctype html>
<html>
  <head>
    <title>Моя домашняя страничка</title>
  </head>
  <body>
    <h1> Моя домашняя страничка </h1>
    <p>Привет, я Марийн и это моя домашняя страничка.</p>

    <p>А ещё я книжку написал! Читайте её
      <a href="http://eloquentjavascript.net">здесь</a>.
</p>
  </body>
</html>
```

У этой страницы следующая структура:



Структура данных, использующаяся браузером для представления документа, отражает его форму. Для каждой коробки есть объект, с которым мы можем взаимодействовать и узнавать про него разные данные – какой тег он представляет, какие коробки и текст содержит. Это представление называется Document Object Model (объектная модель документа), или сокращённо DOM.

Мы можем получить доступ к этим объектам через глобальную переменную document. Её свойство documentElement ссылается на объект, представляющий тег . Он также предоставляет свойства head и body, в которых содержатся объекты для соответствующих элементов.

Деревья

Вспомните синтаксические деревья из главы 11. Их структура удивительно похожа на структуру документа браузера. Каждый узел может ссылаться на другие узлы, у каждого из ответвлений может быть своё ответвление. Эта структура – типичный пример вложенных структур, где элементы содержат подэлементы, похожие на них самих.

Мы зовём структуру данных деревом, когда она разветвляется, не имеет циклов (узел не может содержать сам себя), и имеет единственный ярко выраженный «корень». В случае DOM в качестве корня выступает document.documentElement.

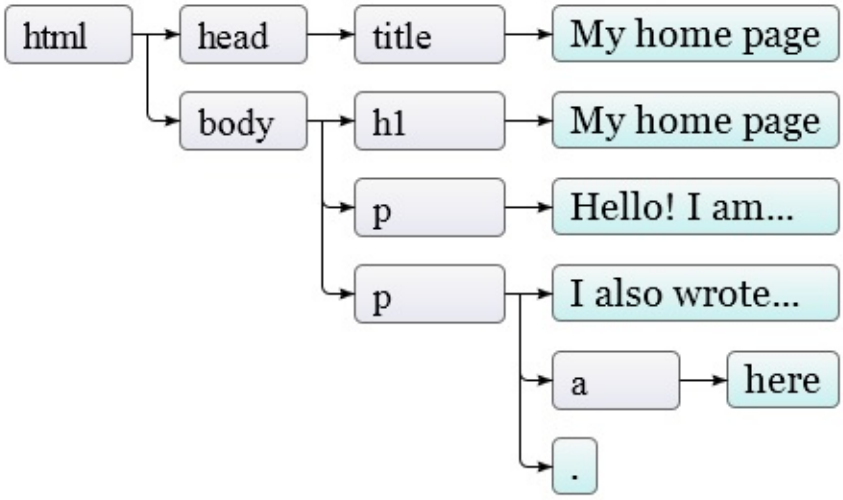
Деревья часто встречаются в вычислительной науке. В дополнение к представлению рекурсивных структур вроде документа HTML или программ, они часто используются для работы с сортированными наборами данных, потому что элементы обычно проще найти или вставлять в отсортированное дерево, чем в отсортированный одномерный массив.

У типичного дерева есть разные узлы. У синтаксического дерева языка Egg были переменные, значения и приложения. У приложений всегда были дочерние ветви, а переменные и значения были «листьями», то есть узлами без дочерних ответвлений.

То же и у DOM. Узлы для обычных элементов, представляющих теги HTML, определяют структуру документа. У них могут быть дочерние узлы. Пример такого узла — document.body. Некоторые из этих дочерних узлов могут оказаться листьями – например, текст или комментарии (в HTML комментарии записываются между символами `<!--` и `-->`).

У каждого узлового объекта DOM есть свойство nodeType, содержащее цифровой код, определяющий тип узла. У обычных элементов он равен 1, что также определено в виде свойства-константы document.ELEMENT_NODE. У текстовых узлов, представляющих отрывки текста, он равен 3 (document.TEXT_NODE). У комментариев — 8 (document.COMMENT_NODE).

То есть, вот ещё один способ графически представить дерево документа:



Листья – текстовые узлы, а стрелки показывают взаимоотношения отец-ребёнок между узлами.

Стандарт

Использовать загадочные цифры для представления типа узла – это подход не в стиле JavaScript. Позже мы встретимся с другими частями интерфейса DOM, которые тоже кажутся чуждыми и нескладными. Причина в том, что DOM разрабатывался не только для JavaScript. Он пытается определить интерфейс, не зависящий от языка, который можно использовать и в других системах – не только в HTML, но и в XML, который представляет из себя формат данных общего назначения с синтаксисом, напоминающим HTML.

Получается неудобно. Хотя стандарты – и весьма полезная штука, в нашем случае преимущество независимости от языка не такое уж и полезное. Лучше иметь интерфейс, хорошо приспособленный к языку, который вы используете, чем интерфейс, который будет знаком при использовании разных языков.

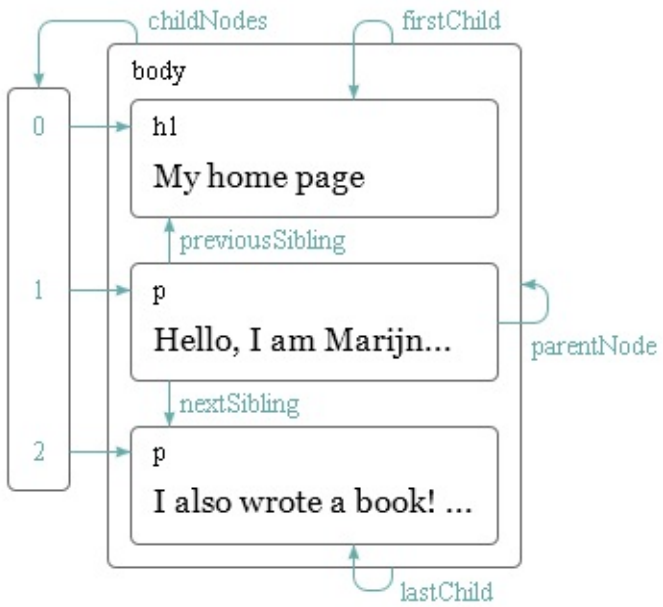
Чтобы показать неудобную интеграцию с языком, рассмотрим свойство childNodes, которое есть у узлов DOM. В нём содержится объект, похожий на массив, со свойством length, и пронумерованные свойства для доступа к дочерним узлам. Но это – экземпляр типа NodeList, не настоящий массив, поэтому у него нет методов вроде forEach.

Есть также проблемы, связанные с плохой продуманностью системы. К примеру, нельзя создать новый узел и сразу добавить к нему свойства или дочерние узлы. Сначала нужно его создать, затем добавить дочерние по одному, и в конце назначить свойства по одному, с использованием побочных эффектов. Код, плотно работающий с DOM, получается длинным, некрасивым и со множеством повторов.

Но эти проблемы не фатальные. JavaScript позволяет создавать абстракции. Легко написать вспомогательные функции, позволяющие выражать операции более понятно и коротко. Вообще, такого рода инструменты предоставляют много библиотек, направленных на программирование для браузера.

Обход дерева

Узлы DOM содержат много ссылок на соседние. Это показано на диаграмме:



Хотя тут показано только по одной ссылке каждого типа, у каждого узла есть свойство parentNode, указывающего на его родительский узел. Также у каждого узла-элемента (тип 1) есть свойство childNodes, указывающее на массивоподобный объект, содержащий его дочерние узлы.

В теории можно пройти в любую часть дерева, используя только эти ссылки. Но JavaScript предоставляет нам много дополнительных вспомогательных ссылок. Свойства firstChild и lastChild показывают на первый и последний дочерний элементы, или содержат null у тех узлов, у которых нет дочерних. previousSibling и nextSibling указывают на соседние узлы – узлы того же родителя, что и текущего узла, но находящиеся в списке сразу до или после текущей. У первого узла свойство previousSibling будет null, а у последнего nextSibling будет null.

При работе с такими вложенными структурами пригождаются рекурсивные функции. Следующая ищет в документе текстовые узлы, содержащие заданную строку, и возвращает true, когда находит:

```
function talksAbout(node, string) {
  if (node.nodeType == document.ELEMENT_NODE) {
    for (var i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string))
        return true;
    }
    return false;
  } else if (node.nodeType == document.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "книг"));
// → true
```

Свойства текстового узла nodeValue содержит строчку текста.

Поиск элементов

Часто бывает полезным ориентироваться по этим ссылкам между родителями, детьми и родственными узлами и проходить по всему документу. Однако если нам нужен конкретный узел в документе, очень неудобно идти

по нему, начиная с document.body и тупо перебирая жёстко заданный в коде путь. Поступая так, мы вносим в программу допущения о точной структуре документа – а её мы позже можем захотеть поменять. Другой усложняющий фактор – текстовые узлы создаются даже для пробелов между узлами. В документе из примера у тега body не три дочерних (h1 и два p), а целых семь: эти три плюс пробелы до, после и между ними.

Так что если нам нужен атрибут href из ссылки, мы не должны писать в программе что-то вроде: «второй ребёнок шестого ребёнка document.body». Лучше бы, если б мы могли сказать: «первая ссылка в документе». И так можно сделать:

```
var link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

У всех узлов-элементов есть метод getElementsByTagName, собирающий все элементы с данным тэгом, которые происходят (прямые или не прямые потомки) от этого узла, и возвращает его в виде массивоподобного объекта.

Чтобы найти конкретный узел, можно задать ему атрибут id и использовать метод document.getElementById.

```
<p>Мой страус Гертруда:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  var ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

Третий метод – getElementsByClassName, который, как и getElementsByTagName, ищет в содержимом узла-элемента и возвращает все элементы, содержащие в своём классе заданную строчку.

Меняем документ

Почти всё в структуре DOM можно менять. У узлов-элементов есть набор методов, которые используются для их изменения. Метод removeChild удаляет заданную дочерний узел. Для добавления узла можно использовать appendChild, который добавляет узел в конец списка, либо insertBefore, добавляющий узел, переданную первым аргументом, перед узлом, переданным вторым аргументом.

```
<p>Один</p>
<p>Два</p>
<p>Три</p>

<script>
  var paragraphs = document.body.getElementsByTagName("p"
);
  document.body.insertBefore(paragraphs[2], paragraphs[0
]);
</script>
```

Узел может существовать в документе только в одном месте. Поэтому вставляя параграф «Три» перед параграфом «Один» мы фактически удаляем его из конца списка и вставляем в начало, и получаем «Три/Один/Два». Все операции по вставке узла приведут к его исчезновению с текущей позиции (если у него таковая была).

Метод replaceChild используется для замены одного дочернего узла другим. Он принимает два узла: новый, и тот, который надо заменить. Заменяемый узел должен быть дочерним узлом того элемента, чей метод мы вызываем. Как replaceChild, так и insertBefore в качестве первого аргумента ожидают получить новый узел.

Создание узлов

В следующем примере нам надо сделать скрипт, заменяющий все картинки (тег `<img>`) в документе текстом, содержащимся в их атрибуте “alt”, который задаёт альтернативное текстовое представление картинки.

Для этого надо не только удалить картинки, но и добавить новые текстовые узлы им на замену. Для этого мы используем метод document.createTextNode.

```
<p>Это <img src="img/cat.png" alt="Кошка"> в
  <img src="img/hat.png" alt="сапожках">.</p>

<p><button onclick="replaceImages()">Заменить</button></p
>

<script>
  function replaceImages() {
    var images = document.body.getElementsByTagName("img"
);
    for (var i = images.length - 1; i >= 0; i--) {
      var image = images[i];
      if (image.alt) {
        var text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Получая строку, createTextNode даёт нам тип 3 узла DOM (текстовый), который мы можем вставить в документ, чтобы он был показан на экране.

Цикл по картинкам начинается в конце списка узлов. Это сделано потому, что список узлов, возвращаемый методом getElementsByTagName (или свойством childNodes) постоянно обновляется при изменениях документа. Если б мы начали с начала, удаление первой картинки привело бы к потере списком первого элемента, и во время второго прохода цикла, когда i равно 1, он бы остановился, потому что длина списка стала бы также равняться 1.

Если вам нужно работать с фиксированным списком узлов вместо «живого», можно преобразовать его в настоящий массив при помощи метода slice.

```
var arrayish = {0: "один", 1: "два", length: 2};
var real = Array.prototype.slice.call(arrayish, 0);
real.forEach(function(elt) { console.log(elt); });
// → один
//   два
```

Для создания узлов-элементов (тип 1) можно использовать document.createElement. Метод принимает имя тега и возвращает новый пустой узел заданного типа. Следующий пример определяет инструмент elt,

создающий узел-элемент и использующий остальные аргументы в качестве его детей. Эта функция потом используется для добавления дополнительной информации к цитате.

```
<blockquote id="quote">
Никакая книга не может быть закончена. Во время работы н
ад ней мы узнаём достаточно для того, чтобы найти её нез
релой сразу же после того, как мы отвлеклись от неё.
</blockquote>

<script>
  function elt(type) {
    var node = document.createElement(type);
    for (var i = 1; i < arguments.length; i++) {
      var child = arguments[i];
      if (typeof child == "string")
        child = document.createTextNode(child);
      node.appendChild(child);
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Карл Поппер"),
        ", предисловие ко второму изданию ",
        elt("em", "Открытое общество и его враги "),
        ", 1950"));
</script>
```

Атрибуты

К некоторым элементам атрибутов, типа href у ссылок, можно получить доступ через одноимённое свойство объекта. Это возможно для ограниченного числа часто используемых стандартных атрибутов.

Но HTML позволяет назначать узлам любые атрибуты. Это полезно, т.к. позволяет вам хранить дополнительную информацию в документе. Если вы придумаете свои названия атрибутов, их не будет среди свойств узла-элемента. Вместо этого вам надо будет использовать методы getAttribute и setAttribute для работы с ними.

```
<p data-classified="secret">Код запуска 00000000.</p>
<p data-classified="unclassified">У кошки четыре ноги.</p
>

<script>
  var paras = document.body.getElementsByTagName("p");
  Array.prototype.forEach.call(paras, function(para) {
    if (para.getAttribute("data-classified") == "secret"
)
      para.parentNode.removeChild(para);
  });
</script>
```

Рекомендую перед именами придуманных атрибутов ставить “data-“, чтобы быть уверенным, что они не конфликтуют с любыми другими. В качестве простого примера мы напишем подсветку синтаксиса, который ищет теги `<pre>` (“preformatted”, предварительно отформатированный – используется для кода и простого текста) с атрибутом data-language (язык) и довольно грубо пытается подсветить ключевые слова в языке.

```
function highlightCode(node, keywords) {
  var text = node.textContent;
  node.textContent = ""; // Очистим узел

  var match, pos = 0;
  while (match = keywords.exec(text)) {
    var before = text.slice(pos, match.index);
    node.appendChild(document.createTextNode(before));
    var strong = document.createElement("strong");
    strong.appendChild(document.createTextNode(match[0])
);
    node.appendChild(strong);
    pos = keywords.lastIndex;
  }
  var after = text.slice(pos);
  node.appendChild(document.createTextNode(after));
}
```

Функция highlightCode принимает узел `<pre>` и регулярку (с включённой настройкой global), совпадающую с ключевым словом языка программирования, которое

содержит элемент.

Свойство textContent используется для получения всего текста узла, а затем устанавливается в пустую строку, что приводит к очищению узла. Мы в цикле проходим по всем вхождениям выражения keyword, добавляем между ними текст в виде простых текстовых узлов, а совпавший текст (ключевые слова) добавляем, заключая их в элементы `<strong>` (жирный шрифт).

Мы можем автоматически подсветить весь код страницы, перебирая в цикле все элементы `<pre>`, у которых есть атрибут `data-language`, и вызывая на каждом `highlightCode` с правильной регуляркой.

```
var languages = {
  javascript: /\b(function|return|var)\b/g /* … etc */
};

function highlightAllCode() {
  var pres = document.body.getElementsByTagName("pre");
  for (var i = 0; i < pres.length; i++) {
    var pre = pres[i];
    var lang = pre.getAttribute("data-language");
    if (languages.hasOwnProperty(lang))
      highlightCode(pre, languages[lang]);
  }
}
```

Вот пример:

```
<source lang="html">
<p>А вот и она, функция идентификации:</p>
<pre data-language="javascript">
function id(x) { return x; }
```

Есть один часто используемый атрибут, `class`, имя которого является ключевым словом в JavaScript. По историческим причинам, когда старые реализации JavaScript не умели обращаться с именами свойств, совпадавшими с ключевыми словами, этот атрибут доступен через свойство под названием `className`. Вы также можете получить к нему доступ по его настоящему имени `class` через методы `getAttribute` и `setAttribute`.

Расположение элементов (layout)

Вы могли заметить, что разные типы элементов располагаются по-разному. Некоторые, типа параграфов `<p>` и заголовков `<h1>` растягиваются на всю ширину документа и появляются на отдельных строках. Такие элементы называют блочными. Другие, как ссылки `<a>` или жирный текст `<strong>` появляются на одной строчке с окружающим их текстом. Они называются встроенными (inline).

Для любого документа браузеры могут построить расположение элементов, расклад, в котором у каждого будет размер и положение на основе его типа и содержимого. Затем этот расклад используется для создания внешнего вида документа.

Размер и положение элемента можно узнать через JavaScript. Свойства offsetWidth и offsetHeight выдают размер в пикселях, занимаемый элементом. Пиксель – основная единица измерений в браузерах, и обычно соответствует размеру минимальной точки экрана. Сходным образом, clientWidth и clientHeight дают размер внутренней части элемента, не влючая ширину его границ (border).

```
<p style="border: 3px solid red">
  Я в коробочке
</p>

<script>
  var para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

Самый эффективный способ узнать точное расположение элемента на экране – метод getBoundingClientRect. Он возвращает объект со свойствами top, bottom, left, и right (сверху, снизу, слева и

справа), которые содержат положение элемента относительно левого верхнего угла экрана в пикселях. Если вам надо получить эти данные относительно всего документа, вам надо прибавить текущую позицию прокрутки, которая содержится в глобальных переменных pageXOffset и pageYOffset.

Разбор документа – задача сложная. В целях быстродействия браузерные движки не перестраивают документ каждый раз после его изменения, а ждут так долго, как это возможно. Когда программа JavaScript, изменившая документ, заканчивает работу, браузеру надо будет просчитать новую раскладку страницы, чтобы вывести изменённый документ на экран. Когда программа запрашивает позицию или размер чего-либо, читая свойства типа offsetHeight или вызывая getBoundingClientRect, для предоставления корректной информации тоже необходимо рассчитывать раскладку.

Программа, которая периодически считывает раскладку DOM и изменяет DOM, заставляет браузер много раз пересчитывать раскладку, и в связи с этим будет работать медленно. В следующем примере есть две разные программы, которые строят линию из символов X шириной в 2000 пикс, и измеряют время работы.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    var start = Date.now(); // Текущее время в миллисеку
ндах
    action();
    console.log(name, "заняло", Date.now() - start, "ms"
);
  }

  time("тупо", function() {
    var target = document.getElementById("one");
    while (target.offsetWidth < 2000)
      target.appendChild(document.createTextNode("X"));
  });
  // → тупо заняло 32 ms

  time("умно", function() {
    var target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"))
;
    var total = Math.ceil(2000 / (target.offsetWidth / 5
));
    for (var i = 5; i < total; i++)
      target.appendChild(document.createTextNode("X"));
  });
  // → умно заняло 1 ms
</script>
```

Стили

Мы видели, что разные элементы HTML ведут себя по-разному. Некоторые показываются в виде блоков, другие встроенные. Некоторые добавляют визуальный стиль – например, `<strong>` делает жирным текст и `<a>` делает текст подчёркнутым и синим.

Внешний вид картинки в теге `<img>` или то, что ссылка в теге `<a>` при клике открывает новую страницу, связано с типом элемента. Но основные стили, связанные с элементом, вроде цвета текста или подчёркивания, могут быть нами изменены. Вот пример использования свойства style (стиль):

```
<p><a href=".">Обычная ссылка</a></p>
<p><a href="." style="color: green">Зелёная ссылка</a></p
>
```

Атрибут style может содержать одно или несколько объявлений свойств (color), за которым следует двоеточие и значение. В случае нескольких объявлений они разделяются точкой с запятой: “color: red; border: none”.

Много всякого можно изменить при помощи стилей. Например, свойство display контролирует, показывается ли элемент в блочном или встроенном виде.

```
Текст показан <strong>встроенным</strong>,
<strong style="display: block">в виде блока</strong>, и
<strong style="display: none">вообще не виден</strong>.
```

Блочный элемент выводится отдельным блоком, а последний вообще не виден – display: none отключает показ элементов. Таким образом можно прятать элементы. Обычно это предпочтительно полному удалению их из документа, потому что их легче потом при необходимости снова показать.

Код JavaScript может напрямую действовать на стиль элемента через свойство узла style. В нём содержится объект, имеющий свойства для всех свойств стилей. Их значения – строки, в которые мы можем писать для смены какого-то аспекта стиля элемента.

```
<p id="para" style="color: purple">
  Красотень
</p>

<script>
  var para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Некоторые имена свойств стилей содержат дефисы, например font-family. Так как с ними неудобно было бы работать в JavaScript (пришлось бы писать style[«font-family»]), названия свойств в объекте стилей пишутся без дефиса, а вместо этого в них появляются прописные буквы: style.fontFamily

Каскадные стили

Система стилей в HTML называется CSS (Cascading Style Sheets, каскадные таблицы стилей). Таблица стилей – набор стилей в документе. Его можно писать внутри тега `<style>`:

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Теперь <strong>текст тега strong</strong> наклонный и
 серый.</p>
```

«Каскадные» означает, что несколько правил комбинируются для получения окончательного стиля документа. В примере на стиль по умолчанию для

`<strong>`, который делает текст жирным, накладывается правило из тега `<style>`, по которому добавляется font-style и цвет.

Когда значение свойства определяется несколькими правилами, приоритет остаётся у более поздних. Если бы стиль текста в `<style>` включал правило font-weight: normal, конфликтующее со стилем по умолчанию, то текст был бы обычный, а не жирный. Стили, которые применяются к узлу через атрибут style, имеют наивысший приоритет.

В CSS возможно задавать не только название тегов. Правило для .abc применяется ко всем элементам, у которых указан класс “abc”. Правило для #xyz применяется к элементу с атрибутом id равным “xyz” (атрибуты id необходимо делать уникальными для документа).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* Элементы p, у которых указаны классы a и b, а id зада
н как main */
p.a.b#main {
  margin-bottom: 20px;
}
```

Приоритет самых поздних правил работает, когда у правил одинаковая детализация. Это мера того, насколько точно оно описывает подходящие элементы, определяемая числом и видом необходимых аспектов элементов. К примеру, правило для p.a более детально, чем правила для p или просто .a, и будет иметь приоритет.

Запись `p > a {…}` применима ко всем тегам `<a>`, находящимся внутри тега `<p>` и являющимся его прямыми потомками. `p a {…}` применимо также ко всем тегам `<a>` внутри `<p>`, при этом неважно, является ли `<a>` прямым потомком или нет.

Селекторы запросов

В этой книге мы не будем часто использовать таблицы стилей. Понимание их работы критично для программирования в браузере, но подробное разъяснение всех их свойств заняло бы 2-3 книги. Главная причина знакомства с ними и с синтаксисом селекторов (записей, определяющих, к каким элементам относятся правила) – мы можем использовать тот же эффективный мини-язык для поиска элементов DOM.

Метод querySelectorAll, существующий и у объекта document, и у элементов-узлов, принимает строку селектора и возвращает массивоподобный объект, содержащий все элементы, подходящие под него.

```
<p>Люблю грозу в начале
  <span>мая</span></p>
<p>Когда весенний первый гром</p>
<p>Как бы <span>резвяся
  <span>и играя</span></span></p>
<p>Грохочет в небе голубом.</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // Все элементы <p>
  // → 4
  console.log(count(".animal"));     // Класс animal
  // → 2
  console.log(count("p .animal"));   // Класс animal вну
три <p>
  // → 2
  console.log(count("p > .animal")); // Прямой потомок <
p>
  // → 1
</script>
```

В отличие от методов вроде getElementsByTagName, возвращаемый querySelectorAll объект не интерактивный. Он не изменится, если вы измените документ.

Метод querySelector (без All) работает сходным образом. Он нужен, если вам необходим один конкретный элемент. Он вернёт только первое совпадение, или null, если совпадений нет.

Расположение и анимация

Свойство стилей position сильно влияет на расположение элементов. По умолчанию оно равно static, что означает, что элемент находится на своём обычном месте в документе. Когда оно равно relative, элемент всё ещё занимает место, но теперь свойства top и left можно использовать для сдвига относительно его обычного расположения. Когда оно равно absolute, элемент удаляется из нормального «потока» документа – то есть, он не занимает место и может накладываться на другие. Кроме того, его свойства left и top можно использовать для абсолютного позиционирования относительно левого верхнего угла ближайшего включающего его элемента, у которого position не равно static. А если такого элемента нет, тогда он позиционируется относительно документа.

Мы можем использовать это для создания анимации. Следующий документ показывает картинку с котом, которая двигается по эллипсу.

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  var cat = document.querySelector("img");
  var angle = 0, lastTime = null;
  function animate(time) {
    if (lastTime != null)
      angle += (time - lastTime) * 0.001;
    lastTime = time;
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(animate);
  }
  requestAnimationFrame(animate);
</script>
```

Картинка отцентрирована на странице и ей задана position: relative. Мы постоянно обновляем свойства top и left картинки, чтобы она двигалась.

Скрипт использует requestAnimationFrame для вызова функции animate каждый раз, когда браузер готов перерисовывать экран. Функция animate сама опять вызывает requestAnimationFrame, чтобы запланировать следующее обновление. Когда окно браузера (или закладка) активна, это приведёт к обновлениям со скорость примерно 60 раз в секунду, что позволяет добиться хорошо выглядящей анимации.

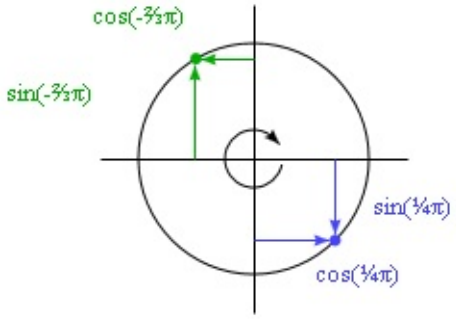
Если бы мы просто обновляли DOM в цикле, страница бы зависла и ничего не было бы видно. Браузеры не обновляют страницу во время работы JavaScript, и не допускают в это время работы со страницей. Поэтому нам нужна requestAnimationFrame – она сообщает браузеру, что мы пока закончили, и он может заниматься своими браузерными вещами, например обновлять экран и отвечать на запросы пользователя.

Нашей функции анимации передаётся текущее время в качестве аргумента, которое она сравнивает с предыдущим (переменная lastTime), чтобы движение кота было однородным, и анимация работала плавно. Если бы мы просто передвигали её на заданный промежуток на каждом шаге, движение бы запиналось если бы, например, другая задача загрузила бы компьютер.

Движение по кругу осуществляется с применением тригонометрических функций Math.cos и Math.sin. Я кратко опишу их для тех, кто с ними незнаком, так как они понадобятся нам в дальнейшем.

Math.cos и Math.sin полезны тогда, когда надо найти точки на круге с центром в точке (0, 0) и радиусом в единицу. Обе функции интерпретируют свой аргумент как позицию на круге, где 0 обозначает точку с правого края круга, затем нужно против часовой стрелки, пока путь длиной в 2π (около 6.28) не проведёт нас по кругу. Math.cos считает

координату по оси x той точки, которая является нашей текущей позицией на круге, а Math.sin выдаёт координату y. Позиции (или углы) больше, чем 2π или меньше чем 0, тоже допустимы – повороты повторяются так, что a+2π означает тот же самый угол, что и a.



Использование синуса и косинуса для вычисления координат

Анимация кота хранит счётчик angle для текущего угла поворота анимации, и увеличивает его пропорционально прошедшему времени каждый раз при вызове функции animation. Этот угол используется для подсчёта текущей позиции элемента image. Стиль top подсчитывается через Math.sin и умножается на 20 – это вертикальный радиус нашего эллипса. Стиль left считается через Math.cos и умножается на 200, так что ширина эллипса сильно больше высоты.

Стилям обычно требуются единицы измерения. В нашем случае приходится добавлять px к числу, чтобы объяснить браузеру, что мы считаем в пикселях (а не в сантиметрах, ems или других единицах). Это легко забыть. Использование чисел без единиц измерения приведёт к игнорированию стиля – если только число не равно 0, что не зависит от единиц измерения.

Итог

Программы JavaScript могут изучать и изменять текущий отображаемый браузером документ через структуру под названием DOM. Эта структура данных представляет модель документа браузера, а программа JavaScript может изменять её для изменения видимого документа. DOM организован в виде дерева, в котором элементы расположены иерархически в соответствии со структурой документа. У объектов элементов есть свойства типа parentNode и childNodes, которые используются для ориентирования на дереве.

Внешний вид документа можно изменять через стили, либо добавляя стили к узлам напрямую, либо определяя правила для каких-либо узлов. У стилей есть очень много

свойств, таких, как color или display. JavaScript может влиять на стиль элемента напрямую через его свойство style.

Упражнения

Строим таблицу

Мы строили таблицы из простого текста в главе 6. HTML упрощает построение таблиц. Таблица в HTML строится при помощи следующих тегов:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>country</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

Для каждой строки в теге `<table>` содержится тег `<tr>`. Внутри него мы можем размещать ячейки: либо ячейки заголовков `<th>`, либо обычные ячейки `<td>`.

Те же данные, что мы использовали в главе 6, снова доступны в переменной MOUNTAINS.

Напишите функцию buildTable, которая, принимая массив объектов с одинаковыми свойствами, строит структуру DOM, представляющую таблицу. У таблицы должна быть строка с заголовками, где имена свойств обёрнуты в элементы `<th>`, и должно быть по одной строчке на объект из массива, где его свойства обёрнуты в элементы `<td>`. Здесь пригодится функция Object.keys, возвращающая массив, содержащий имена свойств объекта.

Когда вы разберётесь с основами, выровняйте ячейки с числами по правому краю, изменив их свойство style.textAlign на «right».

```
<style>
  /* Определяет стили для красивых таблиц */
  table  { border-collapse: collapse; }
  td, th { border: 1px solid black; padding: 3px 8px; }
  th     { text-align: left; }
</style>

<script>
  function buildTable(data) {
    // Ваш код
  }

  document.body.appendChild(buildTable(MOUNTAINS));
</script>
```

Элементы по имени тегов

Метод getElementsByTagName возвращает все дочерние элементы с заданным именем тега. Сделайте свою версию этого метода в виде обычной функции, которая принимает узел и строчку (имя тега) и возвращает массив, содержащий все нисходящие узлы с заданным именем тега.

Чтобы выяснить имя тега элемента, используйте свойство tagName. Заметьте, что оно возвратит имя тега в верхнем регистре. Используйте методы строк toLowerCase или toUpperCase.

```
<h1>Заголовок с элементом <span>span</span> внутри.</h1>
<p>Параграф с <span>раз</span>, <span>два</span> элемент
ами spans.</p>

<script>
  function byTagName(node, tagName) {
    // Ваш код
  }

  console.log(byTagName(document.body, "h1").length);
  // → 1
  console.log(byTagName(document.body, "span").length);
  // → 3
  var para = document.querySelector("p");
  console.log(byTagName(para, "span").length);
  // → 2
</script>
```

Шляпа кота

Расширьте анимацию кота, чтобы и кот и его шляпа `<img src="img/hat.png">` летали по противоположным сторонам эллипса.

Или пусть шляпа летает вокруг кота. Или ещё что-нибудь интересное придумайте.

Чтобы упростить расположение множества объектов, неплохо будет переключиться на абсолютное позиционирование. Тогда top и left будут считаться

относительно левого верхнего угла документа. Чтобы не использовать отрицательные координаты, вы можете добавить заданное число пикселей к значениям position.

```
<img src="img/cat.png" id="cat" style="position: absolut
e">
<img src="img/hat.png" id="hat" style="position: absolut
e">

<script>
  var cat = document.querySelector("#cat");
  var hat = document.querySelector("#hat");
  // Your code here.
</script>
```

Обработка событий

Вы властны над своим разумом, но не над внешними событиями. Когда вы поймёте это, вы обретёте силу.

Марк Аврелий, «Медитации».

Некоторые программы работают с вводом пользователя, мышью и клавиатурой. Время возникновения такого ввода и последовательность данных нельзя предсказать заранее. Это требует иного подхода к контролю над порядком выполнения программы, чем уже привычный нам.

Обработчики событий

Представьте интерфейс, в котором единственным способом узнать, нажали ли на кнопку клавиатуры, было бы считывание текущего состояния кнопки. Чтобы реагировать на нажатия, вам пришлось бы постоянно считывать состояния кнопок, чтобы вы могли поймать это состояние, пока кнопка не отжалась. Было бы опасно

проводить другие подсчёты, отнимающие процессорное время, так как можно было бы пропустить момент нажатия.

Таким образом ввод обрабатывался на примитивных устройствах. Шагом вперёд было бы, если железо или операционка замечали бы нажатие кнопки и передавали бы его в очередь. Затем программа периодически могла бы проверять очередь на новые события и реагировать на то, что находится в очереди.

Разумеется, она должна помнить о проверке, и делать это достаточно часто, потому что наличие длительного промежутка времени между нажатием кнопки и тем, когда программа замечает и реагирует на это, ведёт к восприятию этой программы как медленно работающей. Такой подход используется достаточно редко.

Вариант получше – некая промежуточная система, которая позволяет коду реагировать на события в момент их возникновения. Браузеры позволяют это делать путём регистрации функций как обработчиков заданных событий.

```
<p>Щёлкните по документу для запуска обработчика.</p>
<script>
  addEventListener("click", function() {
    console.log("Щёлк!");
  });
</script>
```

Функция addEventListener регистрирует свой второй аргумент как функцию, которая вызывается, когда описанное в первом аргументе событие случается.

События и узлы DOM

Каждый обработчик событий браузера зарегистрирован в контексте. Когда вы вызываете addEventListener, вы вызываете её как метод целого окна, потому что в браузере глобальная область видимости – это объект window. У каждого элемента DOM есть свой метод addEventListener, позволяющий слушать события от этого элемента.

```
<button>Нажми меня нежно.</button>
<p>А здесь нет обработчиков.</p>
<script>
  var button = document.querySelector("button");
  button.addEventListener("click", function() {
    console.log("Кнопка нажата.");
  });
</script>
```

Пример назначает обработчик на DOM-узел кнопки. Нажатия на кнопку запускают обработчик, а нажатия на другие части документа – не запускают.

Присвоение узлу атрибута onclick работает похоже. Но у узла есть только один атрибут onclick, значит таким способом вы можете зарегистрировать только один обработчик. Метод addEventListener позволяет добавлять любое количество обработчиков, так что вы не замените случайно уже назначенный ранее обработчик.

Метод removeEventListener, вызванный с такими же аргументами, как addEventListener, удаляет обработчик.

```
<button>Act-once button</button>
<script>
  var button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

Чтобы это провернуть, мы даём функции имя (в данном случае, once), чтобы её можно было передать и в addEventListener, и в removeEventListener.

Объекты событий

В примерах мы проигнорировали тот факт, что функциям-обработчикам передаётся аргумент – объект события. В нём хранится дополнительная информация о событии. К примеру, если надо узнать, какая кнопка мыши была нажата, мы можем обратиться к свойству which этого объекта.

```
<button>Жми меня, чем хочешь!</button>
<script>
  var button = document.querySelector("button");
  button.addEventListener("mousedown", function(event) {
    if (event.which == 1)
      console.log("Левая");
    else if (event.which == 2)
      console.log("Средняя");
    else if (event.which == 3)
      console.log("Правая");
  });
</script>
```

Хранящаяся в объекте информация – разная для каждого типа событий. Мы обсудим эти типы позже. Свойство объекта type всегда содержит строку, описывающую событие (например, «click» или «mousedown»).

Распространение (propagation)

События, зарегистрированные на узлах, имеющих дочерние узлы, получат и некоторые события, случившиеся с их детьми. Если кликнуть на кнопку внутри параграфа, обработчики событий параграфа получат событие click.

Если и у параграфа и у кнопки есть обработчики, то первым запустится более конкретный – то есть, обработчик кнопки. Событие как бы распространяется наружу, от узла, где оно случилось, до его родительского и далее до корня документа. После отработки всех обработчиков всех промежуточных узлов, очередь среагировать на событие доходит и до самого окна.

В любой момент обработчик может вызвать метод stopPropagation объекта события, чтобы «высшие» узлы не получили его. Это может быть полезным, когда у вас есть кнопка внутри другого кликабельного элемента, и вы не хотите, чтобы клики по кнопке активировали поведение внешнего элемента.

Следующий пример регистрирует обработчики «mousedown» как на кнопке, так и на окружающем параграфе. При щелчке правой кнопкой обработчик кнопки вызывает stopPropagation, который предотвращает запуск обработчика параграфа. При клике другой кнопкой запускаются оба обработчика.

```
<p>Параграф с <button>кнопкой </button>.</p>
<script>
  var para = document.querySelector("p");
  var button = document.querySelector("button");
  para.addEventListener("mousedown", function() {
    console.log("Обработчик параграфа.");
  });
  button.addEventListener("mousedown", function(event) {
    console.log("Обработчик кнопки.");
    if (event.which == 3)
      event.stopPropagation();
  });
</script>
```

У большинства объектов событий есть свойство target, ссылающееся на узел, который запустил обработку. Его можно использовать для проверки того, что вы не обрабатываете что-то, пришедшее с ненужного вам узла.

Также возможно использовать свойство target, чтобы распространить обработку конкретного типа события. К примеру, если у вас есть узел, содержащий длинный список кнопок, было бы удобнее зарегистрировать один обработчик событий для узла, и в нём выяснять, нажали ли на кнопку – вместо того, чтобы регистрировать обработчики каждой кнопки по отдельности.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", function(event
) {
    if (event.target.nodeName == "BUTTON")
      console.log("Clicked", event.target.textContent);
  });
</script>
```

Действия по умолчанию

У многих событий есть действия по умолчанию. При клике на ссылку вы перейдёте по ней. При нажатии на стрелку вниз браузер прокрутит страницу вниз. По правому клику мыши вы увидите контекстное меню. И так далее.

Для большинства типов событий обработчики событий вызываются до того, как сработает действие по умолчанию. Если обработчик не хочет, чтобы это действие происходило (часто потому, что он уже обработал его), он может вызвать метод preventDefault объекта события.

Это можно использовать для создания своих горячих клавиш или контекстного меню. Также это можно использовать для слома привычного пользователю интерфейса. К примеру, вот ссылка, по которой нельзя пройти.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  var link = document.querySelector("a");
  link.addEventListener("click", function(event) {
    console.log("Фигушки.");
    event.preventDefault();
  });
</script>
```

Не делайте так – если у вас нет очень серьёзной причины! Пользователям вашей страницы будет очень неудобно, когда они столкнутся с неожиданными результатами своих действий. В зависимости от браузера, некоторые события перехватить нельзя. В Chrome нельзя обрабатывать горячие клавиши закрытия текущей закладки (Ctrl-W or Command-W).

События от кнопок клавиатуры

При нажатии кнопки на клавиатуре браузер запускает событие «keydown». Когда она отпускается, происходит событие «keyup».

```
<p>Страница по нажатию V офиолетивает.</p>
<script>
  addEventListener("keydown", function(event) {
    if (event.keyCode == 86)
      document.body.style.background = "violet";
  });
  addEventListener("keyup", function(event) {
    if (event.keyCode == 86)
      document.body.style.background = "";
  });
</script>
```

Несмотря на название, «keydown» происходит не только тогда, когда на кнопку нажимают. Если нажать и удерживать кнопку, событие будет происходить каждый раз по приходу повторного сигнала от клавиши (key repeat). Если вам, к примеру, надо увеличивать скорость игрового персонажа, когда нажата кнопка со стрелкой, и уменьшать её, когда она отпущена – надо быть осторожным, чтобы не увеличить скорость каждый раз при повторе сигнала от кнопки, иначе скорость возрастёт очень сильно.

В примере упомянуто свойство keyCode объекта события. Так вы можете узнать, какая именно кнопка нажата или отпущена. К сожалению, не всегда очевидно, как преобразовать числовые коды в нужную кнопку.

Для цифр и букв код будет кодом символа Unicode, связанного с прописным символом, изображённым на кнопке. Метод строки charCodeAt даёт нам этот код.

```
console.log("Violet".charCodeAt(0));
// → 86
console.log("1".charCodeAt(0));
// → 49
```

У других кнопок коды менее предсказуемы. Лучший способ их выяснить – экспериментальный. Зарегистрировать обработчик, который записывает коды клавиш, и нажать нужную кнопку.

Кнопки-модификаторы типа Shift, Ctrl, Alt, и Meta (Command на Mac) создают события, как и нормальные кнопки. Но при разборе комбинаций клавиш можно выяснить, были ли нажаты модификаторы, через свойства shiftKey, ctrlKey, altKey, и metaKey событий клавиатуры и мыши.

```
<p>Нажмите Ctrl-Space для продолжения.</p>
<script>
  addEventListener("keydown", function(event) {
    if (event.keyCode == 32 && event.ctrlKey)
      console.log("Продолжаем!");
  });
</script>
```

События «keydown» и «keyup» дают информацию о физическом нажатии кнопок. А если вам нужно узнать, какой текст вводит пользователь? Создавать его из нажатий кнопок – неудобно. Для этого существует событие «keypress», происходящее сразу после «keydown» (и повторяющееся вместе с «keydown», если клавишу продолжают удерживать), но только для тех кнопок, которые выдают символы. Свойство объекта события charCode содержит код, который можно интерпретировать как код Unicode. Мы можем использовать функцию String.fromCharCode для превращения кода в строку из одного символа.

```
<p>Переведите фокус на страницу и печатайте.</p>
<script>
  addEventListener("keypress", function(event) {
    console.log(String.fromCharCode(event.charCode));
  });
</script>
```

Источником события нажатия клавиши узел становится в зависимости от того, где находился фокус ввода во время нажатия. Обычные узлы не могут получить фокус ввода (если только вы не задали им атрибут tabindex), а такие, как ссылки, кнопки и поля форм – могут. Мы вернёмся к полям ввода в главе 18. Когда ни у чего нет фокуса, в качестве целевого узла событий работает document.body

Кнопки мыши

Нажатие кнопки мыши тоже запускает несколько событий. События «mousedown» и «mouseup» похожи на «keydown» и «keyup», и запускаются, когда кнопка нажата и когда отпущена. События происходят у тех узлов DOM, над которыми находился курсор мыши.

После события «mouseup» на узле, на который пришлись и нажатие, и отпускание кнопки, запускается событие “click”. Например, если я нажал кнопку над одним параграфом, потом передвинул мышь на другой параграф и отпустил кнопку, событие “click” случится у элемента, который содержал в себе оба эти параграфа.

Если два щелка происходят достаточно быстро друг за другом, запускается событие «dblclick» (double-click), сразу после второго запуска “click”.

Для получения точных координат места, где произошло событие мыши, обратитесь к свойствам pageX и pageY – они содержат координаты в пикселях относительно верхнего левого угла.

В примере создана примитивная программа для рисования. Каждый раз по клику на документе он добавляет точку под вашим курсором. В главе 19 будет представлена менее примитивная программа для рисования.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* скруглённые углы */
    background: blue;
    position: absolute;
  }
</style>
<script>
  addEventListener("click", function(event) {
    var dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

Свойства clientX и clientY похожи на pageX и pageY, но дают координаты относительно части документа, которая видна сейчас (если документ был прокручен). Это удобно при сравнении координат мыши с координатами, которые возвращает getBoundingClientRect – его возврат тоже связан с относительными координатами видимой части документа.

Движение мыши

Каждый раз при сдвиге курсора мыши запускается событие «mousemove». Его можно использовать для отслеживания позиции мыши. Обычно это нужно при создании некоей функциональности, связанной с перетаскиванием объектов мышью.

К примеру, следующая программа отображает полоску и устанавливает обработку событий так, что движение влево и вправо уменьшает или увеличивает её ширину.

```
<p>Переместите мышь для увеличения ширины:</p>
<div style="background: orange; width: 60px; height: 20p
x">
</div>
<script>
  var lastX; // Последняя позиция мыши
  var rect = document.querySelector("div");
  rect.addEventListener("mousedown", function(event) {
    if (event.which == 1) {
      lastX = event.pageX;
      addEventListener("mousemove", moved);
      event.preventDefault(); // Запретим выделение
    }
  });

  function moved(event) {
    if (event.which != 1) {
      removeEventListener("mousemove", moved);
    } else {
      var dist = event.pageX - lastX;
      var newWidth = Math.max(10, rect.offsetWidth + dis
t);
      rect.style.width = newWidth + "px";
      lastX = event.pageX;
    }
  }
</script>
```

Обратите внимание – обработчик «mousemove» зарегистрирован у всего окна. Даже если мышь уходит за пределы полоски, нам надо обновлять её размер и прекращать это, когда кнопку отпускают.

Когда курсор попадает на узел и уходит с него, происходят события «mouseover» or «mouseout». Их можно использовать, кроме прочего, для создания эффектов проведения мыши, показывая или меняя стиль чего-либо, когда курсор находится над этим элементом.

К сожалению, создание такого эффекта не ограничивается запуском его при событии «mouseover» и завершением при событии «mouseout». При движении мыши от узла к его дочерним узлам на родительском узле происходит событие «mouseout», хотя мышь, вообще говоря, его и не покидала. Что ещё хуже, эти события распространяются как и все другие, поэтому вы всё равно получаете «mouseout» при уходе курсора с одного их дочерних узлов того узла, где вы зарегистрировали обработчик.

Для обхода проблемы можно использовать свойство relatedTarget объекта событий. Он сообщает, на каком узле была до этого мышь при возникновении события «mouseover», и на какой элемент она переходит при событии «mouseout». Нам надо менять эффект, только когда relatedTarget находится вне нашего целевого узла. Только в этом случае событие на самом деле представляет собой переход на наш узел (или уход с узла).

```
<p>Наведите мышь на этот <strong>параграф </strong>.</p>
<script>
  var para = document.querySelector("p");
  function isInside(node, target) {
    for (; node != null; node = node.parentNode)
      if (node == target) return true;
  }
  para.addEventListener("mouseover", function(event) {
    if (!isInside(event.relatedTarget, para))
      para.style.color = "red";
  });
  para.addEventListener("mouseout", function(event) {
    if (!isInside(event.relatedTarget, para))
      para.style.color = "";
  });
</script>
```

Функция isInside перебирает всех предков узла, пока не доходит до верха документа (и тогда узел равен null), или же не находит заданного ей родителя.

Должен добавить, что такой эффект достижим гораздо проще через псевдоселектор CSS под названием :hover, как показано ниже. Но когда при наведении вам надо делать что-то более сложное, чем изменение стиля узла, придётся использовать трюк с событиями «mouseover» и «mouseout».

```
<style>
  p:hover { color: red }
</style>
<p>Наведите мышь на этот <strong>параграф </strong>.</p>
```

События прокрутки

Когда элемент прокручивается, запускается событие «scroll». Это используется во многих случаях, например чтобы узнать, на что сейчас пользователь смотрит (чтобы останавливать анимацию, не попавшую на экран, или отправлять секретные шпионские донесения в ваш злодейский штаб), или визуально демонстрировать прогресс (подсвечивая часть содержания или показывая номер страницы).

В примере в правом верхнем углу документа создаётся индикатор процесса, который заполняется по мере прокрутки элемента вниз.

```
<style>
  .progress {
    border: 1px solid blue;
    width: 100px;
    position: fixed;
    top: 10px; right: 10px;
  }
  .progress > div {
    height: 12px;
    background: blue;
    width: 0%;
  }
  body {
    height: 2000px;
  }
</style>
<div class="progress"><div></div></div>
<p>Scroll me...</p>
<script>
  var bar = document.querySelector(".progress div");
  addEventListener("scroll", function() {
    var max = document.body.scrollHeight - innerHeight;
    var percent = (pageYOffset / max) * 100;
    bar.style.width = percent + "%";
  });
</script>
```

Позиция элемента fixed означает почти то же, что absolute, но ещё и предотвращает прокручивание элемента вместе с остальным документом. Смысл в том, чтобы оставить наш индикатор в углу. Внутри него находится другой элемент, который изменяет размер,

отражая текущий прогресс. Мы используем проценты вместо px для задания ширины, чтобы размер элемента изменялся относительно размера всего индикатора.

Глобальная переменная innerHeight даёт высоту окна, которую надо вычесть из полной высоты прокручиваемого элемента – при достижении конца элемента прокрутка заканчивается. (Также в дополнение к innerHeight есть переменная innerWidth). Поделив текущую позицию прокрутки pageYOffset на максимальную позицию прокрутки, и умножив на 100, мы получили процент для индикатора.

Вызов preventDefault не предотвращает прокрутку. Обработчик события вызывается уже после того, как прокрутка случилась.

События, связанные с фокусом

При получении элементом фокуса браузер запускает событие “focus”. Когда он теряет фокус, запускается событие “blur”.

В отличие от предыдущих событий, эти два не распространяются. Обработчик родительского узла не уведомляется о получении или утрате фокуса дочерним

элементом.

Следующий пример демонстрирует текст подсказки для того текстового поля, у которого в данный момент фокус.

```
<p>Имя: <input type="text" data-help="Ваше полное имя"></
p>
<p>Возраст: <input type="text" data-help="Возраст в года
x"></p>
<p id="help"></p>

<script>
  var help = document.querySelector("#help");
  var fields = document.querySelectorAll("input");
  for (var i = 0; i < fields.length; i++) {
    fields[i].addEventListener("focus", function(event)
{
      var text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    fields[i].addEventListener("blur", function(event) {
      help.textContent = "";
    });
  }
</script>
```

Объект window получает события focus и blur, когда пользователь выделяет или убирает фокус с закладки браузера или окна браузера, в котором показан документ.

Событие загрузки

Когда заканчивается загрузка страницы, на объектах window и body запускается событие “load”. Это часто используется для планирования инициализирующих действий, которым необходим полностью построенный документ. Вспомните, что содержимое тегов `<script>` запускается сразу, как только тег встречается. Иногда это слишком рано – например, когда скрипту нужно что-то сделать с теми частями документа, которые находятся после тега `<script>`.

У элементов типа картинок или тегов скрипта, которые загружают внешний файл, тоже есть событие “load”, которое показывает, что файл загружен. Как и события фокуса, события загрузки не распространяются.

Когда страница закрывается или с неё уходят (например, по ссылке), запускается событие «beforeunload». Основная цель – защитить пользователя от случайной потери данных при закрытии документа. Предотвращение закрытия страницы не производится, как вы могли подумать, при помощи preventDefault. Вместо этого используется возврат строки из обработчика. Строка будет использована в диалоге, который спрашивает пользователя, хочет ли он остаться на странице или покинуть её. Этот механизм гарантирует, что

пользователь может покинуть страницу, даже если на ней работает зловредный скрипт, который бы хотел не отпускать пользователя, а вместо этого показывал бы ему мошенническую рекламу по снижению веса.

График выполнения скрипта

Несколько вещей могут привести к старту скрипта. Чтение тега `<script>` — одна из них. Запуск события – ещё одна. В главе 13 обсуждается функция requestAnimationFrame, которая планирует запуск функции перед следующей перерисовкой страницы. Это ещё один способ запустить скрипт.

Важно понять, что хотя события и запускаются в любой момент, два разных скрипта одновременно работать не могут. Если скрипт работает, обработчики событий и запланированные другим способом куски кода будут ждать своей очереди. Поэтому документ подвисает, когда скрипт работает слишком долго. Браузер не обрабатывает щелчки и другие события внутри документа потому, что он не может запустить обработчики событий, пока работает текущий скрипт.

В некоторых программных окружениях можно запускать несколько потоков одновременно. Можно сделать программу быстрее, если выполнять несколько вещей

одновременно. Но когда несколько действующих лиц трогают одни и те же части системы в одно и то же время, продумывать программу становится на порядок сложнее.

То, что программы JavaScript делают по одной вещи за раз, облегчает нашу жизнь. Если вам очень надо сделать в фоне что-то тяжёлое, не подвешивая при этом страницу, браузеры предоставляют штуку под названием «сетевые рабочие» (web worker) – изолированное окружение JavaScript, работающее вместе с главной программой над документом, которое может общаться с ней только посредством сообщений.

Предположим, у нас есть следующий код в файле `code/squareworker.js`:

```
addEventListener("message", function(event) {
  postMessage(event.data * event.data);
});
```

Представьте, что возведение в квадрат – очень тяжёлое, долго работающее вычисление, которое нам надо запустить фоновым потоком. Такой код порождает «рабочего», отправляет ему несколько сообщений, и выводит результаты.

```
var squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", function(event)
{
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

Функция postMessage отправляет сообщение, которое запускает событие “message” у принимающей стороны. Скрипт, создавший рабочего, отправляет и получает сообщения через объект Worker, тогда как рабочий общается со скриптом, создавшим его, отправляя и получая сообщения через его собственное глобальное окружение – которое является отдельным окружением, не связанным с оригинальным скриптом.

Установка таймеров

Функция setTimeout схожа с requestAnimationFrame. Она планирует запуск другой функции в будущем. Но вместо вызова функции при следующей перерисовке страницы, она ждёт заданное в миллисекундах время. Эта страница через две секунды превращается из синей в жёлтую:

```
<script>
  document.body.style.background = "blue";
  setTimeout(function() {
    document.body.style.background = "yellow";
  }, 2000);
</script>
```

Иногда вам надо отменить запланированную функцию. Это можно сделать, сохранив значение, возвращаемое setTimeout, и затем вызвав с ним clearTimeout.

```
var bombTimer = setTimeout(function() {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

Функция cancelAnimationFrame работает так же, как clearTimeout – вызов её со значением, возвращённым requestAnimationFrame, отменит этот кадр (если он уже не был вызван).

Похожий набор функций, setInterval и clearInterval используется для установки таймеров, которые будут повторяться каждые X миллисекунд.

```
var ticks = 0;
var clock = setInterval(function() {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Устранение помех (debouncing)

У некоторых событий есть возможность выполняться быстро и много раз подряд (например, «mousemove» и «scroll»). При обработке таких событий надо быть осторожным и не делать ничего «тяжёлого», или ваш обработчик займёт столько времени на выполнение, что взаимодействие с документом будет медленным и прерывистым.

Если в таком обработчике надо сделать что-то нетривиальное, можно использовать setTimeout, чтобы гарантировать, что вы делаете это не слишком часто. Это обычно называют «устранением помех» в событии. К этому существует несколько слегка различающихся подходов.

В первом примере надо сделать что-то, когда пользователь печатает, но не надо делать это сразу после запуска каждого события нажатия на клавиши. Когда они быстро печатают, нам надо подождать, когда возникнет пауза. Вместо немедленного выполнения действия в обработчике, мы устанавливаем таймаут. Также мы очищаем предыдущий таймаут, если он был, так что если события близко одно от другого (ближе, чем задержка таймера), предыдущее событие будет отменено.

```
<textarea>Напишите тут что-нибудь...</textarea>
<script>
  var textarea = document.querySelector("textarea");
  var timeout;
  textarea.addEventListener("keydown", function() {
    clearTimeout(timeout);
    timeout = setTimeout(function() {
      console.log("Вы остановились.");
    }, 500);
  });
</script>
```

Если задать undefined для clearTimeout, или вызвать его с таймаутом, который уже произошёл, то ничего не произойдёт. Таким образом, не надо осторожничать при его вызове, и мы просто поступаем так для каждого события.

Можно использовать немного другой подход, если нам надо разделить ответы минимальными промежутками времени, но при этом запускать их в то время, когда происходят события, а не после. К примеру, надо реагировать на события «mousemove», показывая текущие координаты мыши, но только каждые 250 миллисекунд.

```
<script>
  function displayCoords(event) {
    document.body.textContent =
      "Мышь на " + event.pageX + ", " + event.pageY;
  }

  var scheduled = false, lastEvent;
  addEventListener("mousemove", function(event) {
    lastEvent = event;
    if (!scheduled) {
      scheduled = true;
      setTimeout(function() {
        scheduled = false;
        displayCoords(lastEvent);
      }, 250);
    }
  });
</script>
```

Итог

Обработчики событий позволяют обнаруживать и реагировать на события, над которыми мы не властны. Для их регистрации используется метод addEventListener.

У событий есть определяющий их тип («keydown», «focus», и так далее). Большинство событий вызываются конкретными узлами DOM, и затем распространяются на их предков, позволяя связанными с ними обработчикам обрабатывать их.

При вызове обработчика ему передаётся объект события с дополнительной информацией о событии. У объекта также есть методы, позволяющие остановить дальнейшее распространение (stopPropagation) и предотвратить обработку события браузером по умолчанию (preventDefault).

Нажатия на клавиши запускают события «keydown», «keypress» и «keyup». Нажатия на кнопки мыши запускают события «mousedown», «mouseup» и «click». Движения мыши запускают события «mousemove» и, возможно, «mouseenter» или «mouseout».

Прокрутку можно обнаружить через событие “scroll”, а изменения фокуса через события «focus» и «blur». Когда заканчивается загрузка документа, у объекта window запускается событие “load”.

В одно и то же время может работать один участок программы. Поэтому обработчики событий и другие запланированные скрипты будут ждать окончания работы текущих.

Упражнения

Цензура клавиатуры

В промежутке с 1928 по 2013 год турецкие законы запрещали использование букв Q, W и X в официальных документах. Это являлось частью общей инициативы подавления курдской культуры – эти буквы используются в языке курдов, но не у турков.

В качестве упражнения на тему странного использования технологий, я прошу вас запрограммировать поле для ввода текста так, чтобы эти буквы нельзя было туда вписать. Насчет копирования и вставки и других подобных возможных обходов правила не беспокойтесь.

```
<input type="text">
<script>
  var field = document.querySelector("input");
  // Your code here.
</script>
```

След мыши

В ранние дни JavaScript, когда было время кричащих домашних страниц с обилием анимированных картинок, люди использовали язык очень вдохновляющими способами. Одним из них был «след мыши» — серия картинок, которые следовали за курсором при его движении по странице.

Я хочу, что бы вы в упражнении сделали такой след. Используйте с абсолютным позиционированием, фиксированным размером и цветом фона. Создайте кучку элементов и при движении мыши показывайте их следом за курсором.

К этому можно подойти многими способами. Можно сделать очень простое или очень сложное решение, как угодно. Простое – хранить фиксированное количество элементов и проходить по ним в цикле, двигая каждый следующий на текущее место курсора, каждый раз когда случается событие «mousemove».

```
<style>
  .trail { /* className для элементов, летящих за курсор
ом */
    position: absolute;
    height: 6px; width: 6px;
    border-radius: 3px;
    background: teal;
  }
  body {
    height: 300px;
  }
</style>

<script>
  // Ваш код.
</script>
```

Закладки

Интерфейс закладок встречается часто. Он позволяет вам выбирать панель интерфейса, выбирая одну из нескольких торчащих закладок над элементом.

В упражнении вам нужно сделать простой интерфейс закладок. Напишите функцию asTabs, которая принимает узел DOM, и создаёт закладочный интерфейс, показывая дочерние элементы этого узла. Ей нужно вставлять список элементов `<button>` вверху узла, по одному на каждый дочерний элемент, содержащих текст, полученный из атрибута data-tabname. Все, кроме одного

из дочерних элементов, должны быть спрятаны (при помощи display style none), а текущий видимый узел можно выбирать нажатием кнопки.

Когда оно заработает, расширьте функционал, чтобы у текущей активной кнопки был свой стиль.

```
<div id="wrapper">
  <div data-tabname="one">Закладка один</div>
  <div data-tabname="two">Закладка два</div>
  <div data-tabname="three">Закладка три</div>
</div>
<script>
  function asTabs(node) {
    // Ваш код.
  }
  asTabs(document.querySelector("#wrapper"));
</script>
```

Проект: игра-платформер

Вся наша жизнь – игра.

Ийен Бэнкс, «Игрок»

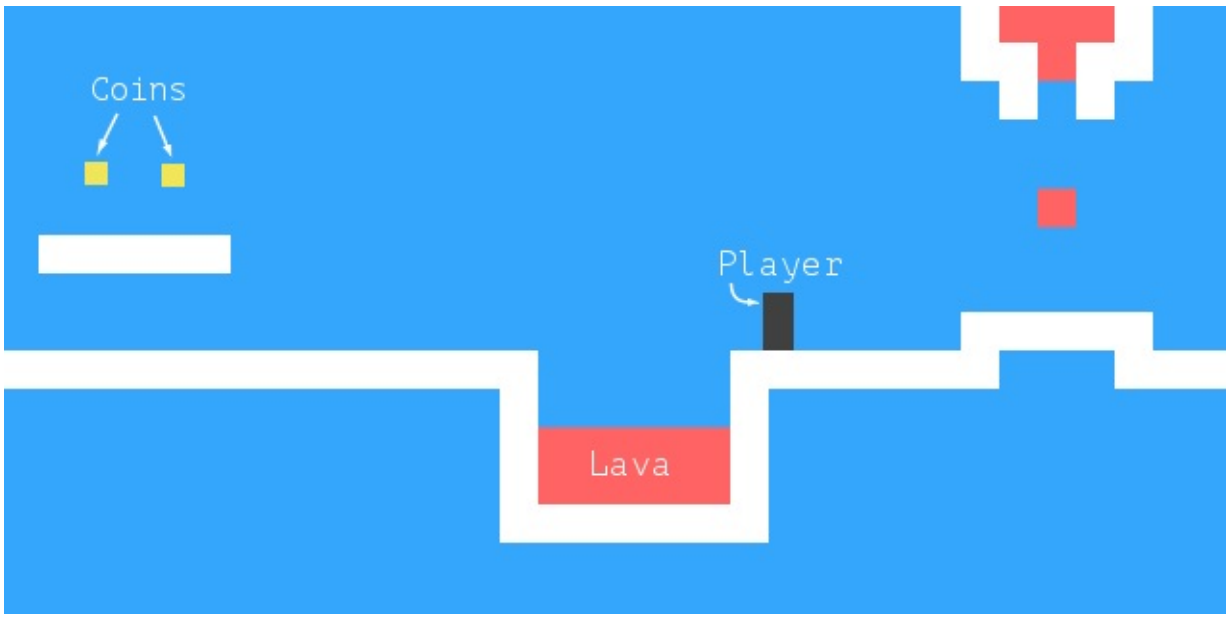
Впервые я увлёкся компьютерами, как и большинство детей, через компьютерные игры. Меня затянуло во вселенную симулированных миров, которыми можно было управлять, и в которых рассказывались истории – мне кажется больше потому, что в них был дан простор моему воображению, чем из-за реальных возможностей, которые они предоставляли.

Никому бы не пожелал карьеру игрового программиста. Как и в музыкальной индустрии, несоответствие между количеством молодых людей, желающих попасть туда и реальным спросом на них, создаёт нездоровую среду. Но написание игр для развлечения – это очень здорово.

В этой главе мы изучим реализацию простого платформера. В платформерах (или «прыгай и беги») от игрока требуется двигать фигурку по (обычно) двумерному миру, который мы видим сбоку, и часто перепрыгивать через разные штуки.

Игра

Наша игра будет примерно базироваться на игре Dark Blue от Томаса Палефа. Я выбрал её, потому что она как развлекательная, так и минималистичная, и её можно сделать минимумом кода. Выглядит она так:



Чёрный прямоугольник представляет игрока, чья задача – собирать жёлтые квадраты (монеты), избегая красных участков (лава?). Уровень заканчивается, когда игрок собрал все монеты.

Игрок может ходить клавишами влево и вправо, и прыгать клавишей вверх. Прыжки – это специальность нашего персонажа. Он может прыгать в несколько раз выше своего роста и менять направление движения в воздухе.

Это не очень-то реалистично, но помогает игроку почувствовать полный контроль над его экранным аватаром.

У игры фиксированный фон в виде решётки, где движущиеся элементы накладываются на фон. Каждая ячейка решётки либо пустая, либо является стеной, либо лавой. Движущиеся элементы – игрок, монеты и некоторые кусочки лавы. В отличие от симуляции жизни из главы 7, позиции этих элементов не привязаны к решётке. Их координаты могут быть дробными, обеспечивая плавное движение.

Технология

Мы используем DOM браузера для графики, и читаем ввод пользователя, обслуживая события клавиатуры.

Код, относящийся к экрану и клавиатуре – небольшая часть работы, которую нам надо проделать для создания игры. Так как всё состоит из цветных квадратиков, рисовать это просто: мы создаём элементы DOM и используем стили, чтобы задать им цвет фона, размер и расположение.

Мы представляем фон как таблицу, поскольку это – неизменная решётка из квадратов. Свободно двигающиеся элементы можно накладывать сверху, используя абсолютное позиционирование.

В играх и других интерактивных программах с графической анимацией, которые должны реагировать на действия пользователя без задержки, очень важна эффективность. Хотя DOM не был задуман для вывода высокоскоростной графики, он справляется с этим лучше, чем можно ожидать. В главе 13 вы видели немножко анимации. На современном компьютере такая простая игра идёт неплохо, даже если не сильно мучиться с оптимизацией.

В следующей главе мы изучим другую технологию браузера, тег `<canvas>`, который предоставляет более традиционный способ для рисования, и работает с формами и пикселями вместо элементов DOM.

Уровни

В главе 7 мы использовали массивы строк для описания двумерной решётки. Мы можем сделать то же и здесь. Это позволит нам разрабатывать уровни без того, чтобы сначала писать редактор уровней.

Простой уровень может выглядеть так:

```
var simpleLevelPlan = [
  "                       ",
  "                       ",
  "  x              = x  ",
  "  x         o o    x  ",
  "  x @      xxxxx   x  ",
  "  xxxxx            x  ",
  "      x!!!!!!!!!!!!x  ",
  "      xxxxxxxxxxxxxx  ",
  "                       "
];
```

Фиксированная решётка и движущиеся элементы включены. Символы x обозначают стены, пробелы – пустое место, а восклицательные знаки – фиксированная лава.

@ отмечает место, где игрок начинает. o – монетки, знак равенства = означает блок движущейся лавы, который двигается по горизонтали туда и сюда. Заметьте, что решётка на этих позициях будет содержать пустое пространство, и для отслеживания позиции этих подвижных элементов используется ещё одна структура данных.

Мы будем поддерживать ещё два вида лавы: вертикальная черта | — для кусочков, двигающихся по вертикали, и v для капающей лавы. Она будет двигаться

вниз, но не отскакивать обратно, а просто перепрыгивать на начальную позицию по достижению пола.

Игра состоит из нескольких уровней, которые надо закончить. Уровень закончен, когда собраны все монетки. Если игрок касается лавы, текущий уровень возвращается к исходному состоянию, и игрок начинает заново.

Чтение уровня

Следующий конструктор создаёт объект уровня. Аргументом должен быть массив строк, задающих уровень.

```
function Level(plan) {
  this.width = plan[0].length;
  this.height = plan.length;
  this.grid = [];
  this.actors = [];

  for (var y = 0; y < this.height; y++) {
    var line = plan[y], gridLine = [];
    for (var x = 0; x < this.width; x++) {
      var ch = line[x], fieldType = null;
      var Actor = actorChars[ch];
      if (Actor)
        this.actors.push(new Actor(new Vector(x, y), ch)
);
      else if (ch == "x")
        fieldType = "wall";
      else if (ch == "!")
        fieldType = "lava";
      gridLine.push(fieldType);
    }
    this.grid.push(gridLine);
  }

  this.player = this.actors.filter(function(actor) {
    return actor.type == "player";
  })[0];
  this.status = this.finishDelay = null;
}
```

Для краткости код не проверяет входящие данные. Он предполагает, что план уровня допустимый, что там есть стартовая позиция игрока и другие необходимые вещи.

Уровень сохраняет свои ширину и высоту и ещё два массива – один для решётки, и один для движущихся частей. Решётку представляет массив массивов, где каждый вложенный массив представляет горизонтальную линию, а каждый квадрат содержит либо null для пустых квадратов, либо строку, отражающую тип квадрата – “wall” или “lava”.

Массив actors содержит объекты, отслеживающие положения и состояния динамических элементов. У каждого из них должно быть свойство pos, содержащее позицию (координаты верхнего левого угла), свойство size с размером, и свойство type со строчкой, описывающей его тип («lava», «coin» или «player»).

После построения решётки мы используем метод filter, чтобы найти объект игрока, хранящийся в свойстве уровня. Свойство status отслеживает, выиграл игрок или проиграл. Когда это случается, используется finishDelay, которое держит уровень активным некоторое время для показа простой анимации. (Просто сразу восстанавливать состояние уровня или начинать следующий – это выглядит некрасиво). Этот метод можно использовать, чтобы узнать, закончен ли уровень:

```
Level.prototype.isFinished = function() {
  return this.status != null && this.finishDelay < 0;
};
```

Действующие лица (актёры)

Для хранения позиции и размера наших актёров мы вернёмся к нашему верному типу Vector, который группирует координаты x и y в объект.

```
function Vector(x, y) {
  this.x = x; this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
Vector.prototype.times = function(factor) {
  return new Vector(this.x * factor, this.y * factor);
};
```

Метод times масштабирует вектор, умножая на заданную величину. Это будет удобно, когда нам надо будет умножать вектор скорости на временной интервал, чтобы узнать пройденный путь за это время.

В предыдущей секции конструктором Level был использован объект actorChars, чтобы связать символы с функциями конструктора. Объект выглядит так:

```
var actorChars = {
  "@": Player,
  "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

Три символа ссылаются на Lava. Конструктор Level передаёт исходный символ актёра в качестве второго аргумента конструктора, и конструктор Lava использует его для корректировки своего поведения (прыгать по горизонтали, прыгать по вертикали, капать).

Тип player построен следующим конструктором. У него есть свойство speed, хранящее его текущую скорость, что поможет нам симулировать импульс и гравитацию.

```
function Player(pos) {
  this.pos = pos.plus(new Vector(0, -0.5));
  this.size = new Vector(0.8, 1.5);
  this.speed = new Vector(0, 0);
}
Player.prototype.type = "player";
```

Поскольку высотой игрок в полтора квадратика, его начальная позиция задаётся на полквадрата выше позиции, где расположен символ “@”. Таким образом его низ совпадает с низом квадрата, в котором он появляется.

При создании динамического объекта Lava, нам надо проинициализировать объект в зависимости от символа. Динамическая лава двигается с заданной скоростью, пока не встретит препятствие. Затем, если у неё есть свойство repeatPos, она отпрыгнет назад на стартовую позицию (капающая). Если нет, она инвертирует скорость и продолжает двигаться в обратном направлении (отскакивает). Конструктор задаёт только необходимые свойства. Позже мы напишем метод, который занимается самим движением.

```
function Lava(pos, ch) {
  this.pos = pos;
  this.size = new Vector(1, 1);
  if (ch == "=") {
    this.speed = new Vector(2, 0);
  } else if (ch == "|") {
    this.speed = new Vector(0, 2);
  } else if (ch == "v") {
    this.speed = new Vector(0, 3);
    this.repeatPos = pos;
  }
}
Lava.prototype.type = "lava";
```

Монеты просты в реализации. Они просто сидят на месте. Но для оживления игры они будут подрагивать, слегка двигаясь по вертикали туда-сюда. Для отслеживания этого, объект coin хранит основную

позицию вместе со свойством wobble, которое отслеживает фазу движения. Вместе они определяют положение монеты (хранящееся в свойстве pos).

```
function Coin(pos) {
  this.basePos = this.pos = pos.plus(new Vector(0.2, 0.1
));
  this.size = new Vector(0.6, 0.6);
  this.wobble = Math.random() * Math.PI * 2;
}
Coin.prototype.type = "coin";
```

В главе 13 мы видели, что Math.sin даёт координату y точки на круге. Она движется туда и обратно в виде плавной волны, пока мы движемся по кругу, что делает функцию синуса пригодной для моделирования волнового движения.

Чтобы избежать случая, когда все монетки двигаются синхронно, начальная фаза каждой будет случайной. Фаза волны Math.sin и ширина волны — 2π. Мы умножаем значение, возвращаемое Math.random, на этот номер, чтобы задать монете случайное начальное положение в волне.

Теперь мы написали всё, что необходимо для представления состояния уровня.

```
var simpleLevel = new Level(simpleLevelPlan);
console.log(simpleLevel.width, "by", simpleLevel.height)
;
// → 22 by 9
```

Нам предстоит выводить эти уровни на экран и моделировать время и движение внутри них.

Бремя инкапсуляции

В большинстве случаев код данной главы не заботится об инкапсуляции. Во-первых, инкапсуляция требует дополнительных усилий. Программы становятся больше, требуют больше концепций и интерфейсов. А так как от слишком большого объёма кода глаза читателя стекленеют, я постарался сохранить программу небольшой.

Во-вторых, различные элементы игры так связаны вместе, что если бы менялось поведение одного из них, вряд ли оставшийся код оставался бы неизменным. Создание интерфейсов между элементами привело бы к использованию слишком большого количества предположений по поводу того, как работает игра. И тогда они были бы неэффективными – меняя одну часть

системы, вам приходилось бы думать, как это влияет на другие части, потому что их интерфейсы не охватывали бы новую ситуацию.

Некоторые части системы хорошо поддаются разделению на кусочки со строго прописанными интерфейсами, а другие – нет. В попытках инкапсулировать нечто, не имеющее чётких границ, вы гарантированно потратите много сил. Совершив такую ошибку, вы увидите, что интерфейсы становятся чересчур большими и детальными, и что их надо часто менять в процессе эволюции программы.

Одну вещь мы всё-таки инкапсулируем – подсистему рисования. Это сделано специально для того, чтобы в следующей главе мы могли выводить на экран ту же игру другим способом. Спрятав рисование за интерфейс, мы можем просто загрузить ту же программу и подключить к ней новый модуль вывода на экран.

Рисование

Инкапсулировать код для рисования мы будем, введя объект display, который выводит уровень на экран. Тип экрана, который мы определяем, зовётся DOMDisplay, потому что он использует элементы DOM для показа уровня.

Мы используем таблицу стилей для задания цветов и других фиксированных свойств элементов, составляющих игру. Было бы возможно непосредственно назначать стиль элементу через свойство style при его создании, но программа в этом случае стала бы излишне многословной.

Следующая вспомогательная функция даёт простой способ создания элемента с назначением класса.

```
function elt(name, className) {
  var elt = document.createElement(name);
  if (className) elt.className = className;
  return elt;
}
```

Экран создаём, передавая ему родительский элемент, к которому необходимо подсоединиться, и объект уровня.

```
function DOMDisplay(parent, level) {
  this.wrap = parent.appendChild(elt("div", "game"));
  this.level = level;

  this.wrap.appendChild(this.drawBackground());
  this.actorLayer = null;
  this.drawFrame();
}
```

Используя тот факт, что appendChild возвращает добавленный элемент, мы создаём окружающий элемент wrapper и сохраняем его в свойстве wrap.

Неизменный фон уровня рисуется единожды. Актёры перерисовываются каждый раз при обновлении экрана. Свойство actorLayer используется в drawFrame для отслеживания элемента, содержащего актёра – чтобы их было легко удалять и заменять.

Координаты и размеры измеряются в единицах, относительных к размеру решётки так, что дистанция в единицу означает один элемент решётки. Когда мы задаём размеры в пикселях, нам нужно будет масштабировать координаты – игра была бы очень мелкой, если б один квадратик задавался одним пикселем. Переменная scale даёт количество пикселей, которое занимает один элемент решётки.

```
var scale = 20;

DOMDisplay.prototype.drawBackground = function() {
  var table = elt("table", "background");
  table.style.width = this.level.width * scale + "px";
  this.level.grid.forEach(function(row) {
    var rowElt = table.appendChild(elt("tr"));
    rowElt.style.height = scale + "px";
    row.forEach(function(type) {
      rowElt.appendChild(elt("td", type));
    });
  });
  return table;
};
```

Как мы уже упоминали, фон рисуется через элемент `<table>`. Это удобно соответствует тому факту, что уровень задан в виде решётки – каждый ряд решётки превращается в ряд таблицы (элемент `<tr>`). Строки решётки используются как имена классов ячеек таблицы (`<td>`). Следующий CSS приводит фон к необходимому нам внешнему виду:

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

Некоторые из настроек (table-layout, border-spacing и padding) используются для подавления нежелательного поведения по умолчанию. Не нужно, чтобы вид таблицы зависел от содержимого ячеек, и не нужны пробелы между ячейками или отступы внутри них.

Правило background задаёт цвет фона. CSS разрешает задавать цвета словами (white) и в формате rgb(R, G, B), где красная, зелёная и синяя компоненты разделены на три числа от 0 до 255. То есть, в записи rgb(52, 166, 251) красный компонент равен 52, зелёный 166 и синий 251. Поскольку синий компонент самый большой, результирующий цвет будет синеватым. Вы можете видеть, что самый большой компонент в правиле .lava – красный.

Каждый актёр рисуется созданием элемента DOM и заданием позиции и размера, основываясь на свойства актёра. Значения надо умножать на масштаб scale, чтобы переходить от единиц игры к пикселям.

```
DOMDisplay.prototype.drawActors = function() {
  var wrap = elt("div");
  this.level.actors.forEach(function(actor) {
    var rect = wrap.appendChild(elt("div",
                                    "actor " + actor.typ
e));
    rect.style.width = actor.size.x * scale + "px";
    rect.style.height = actor.size.y * scale + "px";
    rect.style.left = actor.pos.x * scale + "px";
    rect.style.top = actor.pos.y * scale + "px";
  });
  return wrap;
};
```

Чтобы задать элементу больше одного класса, мы разделяем их имена пробелами. В коде CSS класс actor задаёт позицию absolute. Имя типа используется в дополнительном классе для задания цвета. Нам не надо заново определять класс lava, потому что мы повторно используем класс для лавы из решётки, который мы определили ранее.

```
.actor  { position: absolute;            }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

При обновлении экрана метод drawFrame удаляет старое изображение актёра, если оно было, и затем перерисовывает его на новой позиции. Напрашивается

использование элементов DOM в качестве актёров, но для этого нам потребовалось бы передавать слишком много дополнительной информации между кодом дисплея и кодом симуляции. Надо было бы связать актёров с элементами DOM, и код рисования должен был бы удалять элементы при исчезновении актёров. Так как обычно в игре актёров совсем немного, их перерисовка отнимает немного ресурсов.

```
DOMDisplay.prototype.drawFrame = function() {
  if (this.actorLayer)
    this.wrap.removeChild(this.actorLayer);
  this.actorLayer = this.wrap.appendChild(this.drawActor
s());
  this.wrap.className = "game " + (this.level.status ||
"");
  this.scrollPlayerIntoView();
};
```

Добавив в обёртку wrapper текущий статус уровня в виде класса, мы можем стилизовать персонажа по-разному в зависимости от того, выиграна игра или проиграна. Мы добавим правило CSS, которое работает, только когда у игрока есть потомок с заданным классом.

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

После прикосновения к лаве цвета игрока становятся тёмно-красными, будто он сгорел. Когда последняя монетка собрана, мы используем размытые тени для создания эффекта сияния.

Нельзя предполагать, что уровни всегда вмещаются в окно просмотра. Поэтому нам нужен scrollPlayerIntoView – он нужен для гарантии того, что если уровень не влезает в окно, он будет прокручен, чтобы игрок всегда был близко к центру. Следующий CSS задаёт обёртке максимальный размер, и гарантирует, что всё вылезающее за него не видно. Также мы задаём элементу позицию relative, чтобы актёры внутри него располагались относительно его левого верхнего угла.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

В методе scrollPlayerIntoView мы находим положение игрока и обновляем позицию прокрутки обёртывающего элемента. Мы меняем позицию, работая со свойствами scrollLeft и scrollTop, когда игрок подходит близко к краю.

```
DOMDisplay.prototype.scrollPlayerIntoView = function() {
  var width = this.wrap.clientWidth;
  var height = this.wrap.clientHeight;
  var margin = width / 3;

  // The viewport
  var left = this.wrap.scrollLeft, right = left + width;
  var top = this.wrap.scrollTop, bottom = top + height;

  var player = this.level.player;
  var center = player.pos.plus(player.size.times(0.5))
                 .times(scale);

  if (center.x < left + margin)
    this.wrap.scrollLeft = center.x - margin;
  else if (center.x > right - margin)
    this.wrap.scrollLeft = center.x + margin - width;
  if (center.y < top + margin)
    this.wrap.scrollTop = center.y - margin;
  else if (center.y > bottom - margin)
    this.wrap.scrollTop = center.y + margin - height;
};
```

Метод нахождения центра игрока показывает, как методы наших типов Vector позволяют записывать расчёты, производимые с объектами, наглядно. Чтобы найти центр актёра, мы добавляем его позицию (его левый верхний

угол) и половину высоты. Это центр в координатах уровня, но нам он нужен в координатах пикселей, поэтому мы умножаем результирующий вектор на наш масштаб.

Затем серия проверок подтверждает, что игрок не находится вне доступного пространства. Иногда в результате будут заданы неправильные координаты прокрутки, ниже нуля или больше, чем размер прокручиваемого элемента. Но это не страшно – DOM автоматически ограничит их допустимыми значениями. Если назначить scrollLeft значение -10, он будет равен 0.

Было бы немного проще пробовать прокручивать позицию игрока в центр окна просмотра – но это создаёт неприятный дрожащий эффект. Во время прыжков вид будет постоянно двигаться вверх и вниз. Гораздо приятнее иметь «нейтральную» зону в середине экрана, где можно двигаться, не вызывая прокрутки.

Ещё нам необходимо очищать уровень, когда мы переходим на следующий или начинаем заново.

```
DOMDisplay.prototype.clear = function() {
  this.wrap.parentNode.removeChild(this.wrap);
};
```

Теперь мы можем показать наш уровень.

```
<link rel="stylesheet" href="css/game.css">

<script>
  var simpleLevel = new Level(simpleLevelPlan);
  var display = new DOMDisplay(document.body, simpleLeve
l);
</script>
```

Тэг `<link>` при использовании с `rel="stylesheet"` позволяет загружать файл с CSS. Файл game.css содержит необходимые для игры стили.

Движение и столкновение

Теперь нам надо добавить обработку движений – самое интересное в игре. Простой подход, который используют большинство игр – разделить время на небольшие отрезки, и на каждом шаге сдвигать актёров на дистанцию, соответствующую их скорости (расстояние в секунду), умноженное на длительность временного отрезка (в секундах).

Это просто. Сложность в том, что надо обрабатывать взаимодействие предметов. Когда игрок касается пола или стены, он не должен проходить насквозь. Игра должна замечать, когда движение одного объекта

приводит к столкновению с другим и реагировать соответственно. Стены останавливают движение, монеты собираются, и так далее.

В общем случае эта задача не такая простая. Можно найти библиотеки, обычно называющиеся «физическими движками», симулирующие взаимодействия между физическими объектами в двух или трёх измерениях. В этой главе мы поступим проще, так как нам нужно обрабатывать столкновения только прямоугольных объектов.

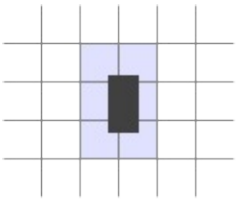
Перед тем, как сдвинуть игрока или блок лавы, мы проверяем, не приведёт ли нас движение внутрь непустой части фона. Если да – мы отменяем движение. Реакция на это будет зависеть от типа актёра – игрок останавливается, лава отскакивает.

Подход требует использования небольших отрезков времени, чтобы объекты останавливались до соприкосновения. Если взять слишком большие отрезки, игрок будет зависать над землёй. Можно было бы использовать более сложный вариант – вычислить место непосредственного соприкосновения и подвинуть актёра туда. Мы поступим проще, и скроем его проблемы, выбрав небольшие временные отрезки.

Метод сообщает, не пересекается ли прямоугольник (заданный позицией и размером) с каким-либо непустым пространством фоновой решётки.

```
Level.prototype.obstacleAt = function(pos, size) {
  var xStart = Math.floor(pos.x);
  var xEnd = Math.ceil(pos.x + size.x);
  var yStart = Math.floor(pos.y);
  var yEnd = Math.ceil(pos.y + size.y);

  if (xStart < 0 || xEnd > this.width || yStart < 0)
    return "wall";
  if (yEnd > this.height)
    return "lava";
  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      var fieldType = this.grid[y][x];
      if (fieldType) return fieldType;
    }
  }
};
```

Метод вычисляет занимаемые телом ячейки решётки, применяя Math.floor и Math.ceil на координатах тела. Помните, что размеры ячеек – 1x1 единиц. Округляя границы тела вверх и вниз, мы получаем промежуток из ячеек фона, которых касается тело.



Поиск столкновений на решётке

Если тело высовывается из уровня, мы всегда возвращаем “wall” для двух сторон и верха и “lava” для низа. Это обеспечит гибель игрока при выходе за пределы уровня. Когда тело внутри решётки, мы в цикле проходим блок квадратов решётки, найденный округлением координат, и возвращаем содержимое первого непустого квадратика.

Столкновения игрока с другими актёрами (монеты, движущаяся лава) обрабатываются после сдвига игрока. Когда движение приводит его к другому актёру, срабатывает соответствующий эффект (сбор монет или гибель).

Этот метод сканирует массив актёров, в поисках того, который накладывается на заданный аргумент:

```
Level.prototype.actorAt = function(actor) {
  for (var i = 0; i < this.actors.length; i++) {
    var other = this.actors[i];
    if (other != actor &&
        actor.pos.x + actor.size.x > other.pos.x &&
        actor.pos.x < other.pos.x + other.size.x &&
        actor.pos.y + actor.size.y > other.pos.y &&
        actor.pos.y < other.pos.y + other.size.y)
      return other;
  }
};
```

Актёры и действия

Метод animate типа Level даёт возможность всем актёрам уровня сдвинуться. Аргумент step задаёт временной промежуток. Объект keys содержит информацию про стрелки клавиатуры, нажатые игроком.

```
var maxStep = 0.05;

Level.prototype.animate = function(step, keys) {
  if (this.status != null)
    this.finishDelay -= step;

  while (step > 0) {
    var thisStep = Math.min(step, maxStep);
    this.actors.forEach(function(actor) {
      actor.act(thisStep, this, keys);
    }, this);
    step -= thisStep;
  }
};
```

Когда у свойства уровня status есть значение, отличное от null (а это бывает, когда игрок выиграл или проиграл), мы уменьшаем до нуля счётчик finishDelay, считающий время между моментом, когда произошёл выигрыш или проигрыш и моментом, когда надо заканчивать показ уровня.

Цикл while делит временной интервал на удобные мелкие куски. Он следит, чтобы промежутки были не больше maxStep. К примеру, шаг в 0.12 секунды будет нарезан на два шага по 0.05 и остаток в 0.02

У объектов актёров есть метод act, который принимает временной шаг, объект level и объект keys. Вот он для типа Lava, который игнорирует объект key:

```
Lava.prototype.act = function(step, level) {
  var newPos = this.pos.plus(this.speed.times(step));
  if (!level.obstacleAt(newPos, this.size))
    this.pos = newPos;
  else if (this.repeatPos)
    this.pos = this.repeatPos;
  else
    this.speed = this.speed.times(-1);
};
```

Он считает новую позицию, добавляя результат умножения временного промежутка и текущей скорости к старой позиции. Если новую позицию не занимает препятствие, происходит перемещение. Если препятствие существует, поведение зависит от типа блока лавы. У капающей лавы есть свойство repeatPos, и она при встрече с препятствием отражается в обратную сторону. Прыгающая лава просто инвертирует скорость (умножает на -1), чтобы продолжить движение в обратном направлении.

Монеты используют метод act, чтобы дрожать. Столкновения они игнорируют, поскольку они просто подрагивают внутри своего квадрата, а столкновения с игроком будут обрабатываться методом act игрока.

```
var wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.act = function(step) {
  this.wobble += step * wobbleSpeed;
  var wobblePos = Math.sin(this.wobble) * wobbleDist;
  this.pos = this.basePos.plus(new Vector(0, wobblePos))
;
};
```

Свойство wobble обновляется, чтобы следить за временем, и потом используется как аргумент Math.sin для создания волны, которая используется для подсчёта новой позиции.

Остаётся игрок. Движение игрока обрабатывается по разным осям отдельно, потому что встреча с полом не должна мешать горизонтальному перемещению, а встреча со стеной – падению или прыжку. Этот метод работает с горизонтальным перемещением.

```
var playerXSpeed = 7;

Player.prototype.moveX = function(step, level, keys) {
  this.speed.x = 0;
  if (keys.left) this.speed.x -= playerXSpeed;
  if (keys.right) this.speed.x += playerXSpeed;

  var motion = new Vector(this.speed.x * step, 0);
  var newPos = this.pos.plus(motion);
  var obstacle = level.obstacleAt(newPos, this.size);
  if (obstacle)
    level.playerTouched(obstacle);
  else
    this.pos = newPos;
};
```

Перемещение подсчитывается на основе состояния клавиш «направо» и «налево». Когда перемещение приводит к встрече с препятствием, вызывается метод уровня playerTouched, который обрабатывает гибель в лаве и сбор монеток. В ином случае объект обновляет свою позицию.

Движение по вертикали работает сходным образом, но симулирует прыжки и гравитацию.

```
var gravity = 30;
var jumpSpeed = 17;

Player.prototype.moveY = function(step, level, keys) {
  this.speed.y += step * gravity;
  var motion = new Vector(0, this.speed.y * step);
  var newPos = this.pos.plus(motion);
  var obstacle = level.obstacleAt(newPos, this.size);
  if (obstacle) {
    level.playerTouched(obstacle);
    if (keys.up && this.speed.y > 0)
      this.speed.y = -jumpSpeed;
    else
      this.speed.y = 0;
  } else {
    this.pos = newPos;
  }
};
```

В начале метода игрок ускоряется по вертикали, чтобы обеспечить гравитацию. Гравитация, скорость прыжка и все остальные константы в игре были подобраны методом проб и ошибок. Я проверял разные значения, пока меня не удовлетворил результат.

Затем мы снова проверяем препятствия. Если мы его встретили, возможны два варианта. Когда нажата клавиша «вверх», и мы двигаемся вниз (то есть, мы встретились с чем-то, что находится под нами), скорости

присваивается довольно большое отрицательное значение. В результате игрок прыгает. В ином случае, мы просто во что-то врезались и скорость обнуляется.

Сам метод act следующий:

```
Player.prototype.act = function(step, level, keys) {
  this.moveX(step, level, keys);
  this.moveY(step, level, keys);

  var otherActor = level.actorAt(this);
  if (otherActor)
    level.playerTouched(otherActor.type, otherActor);

  // Losing animation
  if (level.status == "lost") {
    this.pos.y += step;
    this.size.y -= step;
  }
};
```

После движения метод проверяет других актёров, с которыми игрок сталкивается, и опять вызывает playerTouched, если таковой нашёлся. В этот раз он передаёт вторым аргументом объект actor, так как если другим актёром была монетка, метод playerTouched должен знать, какую именно монетку мы собрали.

В финале, когда игрок погибает (дотронувшись до лавы), мы делаем небольшую анимацию, из-за которой персонаж сжимается (или тонет), уменьшая высоту

объекта player.

Вот метод, обрабатывающий столкновения между игроком и другими объектами:

```
Level.prototype.playerTouched = function(type, actor) {
  if (type == "lava" && this.status == null) {
    this.status = "lost";
    this.finishDelay = 1;
  } else if (type == "coin") {
    this.actors = this.actors.filter(function(other) {
      return other != actor;
    });
    if (!this.actors.some(function(actor) {
      return actor.type == "coin";
    })) {
      this.status = "won";
      this.finishDelay = 1;
    }
  }
};
```

Когда мы тронули лаву, статус игры устанавливается в “lost”. Когда собрана монетка, она удаляется из массива актёров, а если это была последняя – статус игры меняется на “won”. Всё это даёт нам уровень, пригодный для анимации. Не хватает только кода, её обрабатывающего.

Отслеживание клавиш

Для такой игры нам не нужны клавиши, эффект которых работает однократно после keypress. Нам нужен эффект, продолжающийся всё время, пока клавиша нажата (движущаяся фигурка)

Нам надо сделать обработчик клавиш, хранящий текущее состояние кнопок влево, вправо вверх и вниз. Также нам надо вызывать для них preventDefault, чтобы они не прокручивали страницу.

Следующая функция, когда ей дают объект с кодами клавиш в виде имён свойств и названиями клавиш в виде значений, возвращает другой объект, который отслеживает текущее состояние кнопок. Он регистрирует обработчики событий для событий «keydown» и «keyup», и когда код клавиши события совпадает с отслеживаемым кодом, обновляет объект.

```
var arrowCodes = {37: "left", 38: "up", 39: "right"};

function trackKeys(codes) {
  var pressed = Object.create(null);
  function handler(event) {
    if (codes.hasOwnProperty(event.keyCode)) {
      var down = event.type == "keydown";
      pressed[codes[event.keyCode]] = down;
      event.preventDefault();
    }
  }
  addEventListener("keydown", handler);
  addEventListener("keyup", handler);
  return pressed;
}
```

Обратите внимание, как одна функция обработчика используется для событий обоих типов. Она проверяет свойство type объекта события, определяя, надо ли обновлять состояние кнопки на true («keydown») или false («keyup»).

Запуск игры

Функция requestAnimationFrame, которую мы видели в главе 13, предоставляет хороший способ анимировать игру. Но интерфейс её примитивен – его использование

заставляет нас отслеживать момент времени, в который она была вызвана в прошлый раз, и вызывать requestAnimationFrame каждый раз после каждого кадра.

Давайте определим вспомогательную функцию, оборачивающую эти скучные операции в удобный интерфейс, и позволяющую нам просто вызвать runAnimation, задавая ей функцию, которая принимает разницу во времени и рисует один кадр. Когда функция frame возвращает false, анимация останавливается.

```
function runAnimation(frameFunc) {
  var lastTime = null;
  function frame(time) {
    var stop = false;
    if (lastTime != null) {
      var timeStep = Math.min(time - lastTime, 100) / 10
00;
      stop = frameFunc(timeStep) === false;
    }
    lastTime = time;
    if (!stop)
      requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

Я назначил максимальное время для кадра в 100 миллисекунд (1/10 секунды). Когда закладка или окно браузера спрятано, вызовы requestAnimationFrame

прекратятся, пока закладка или окно не станут снова активны. В этом случае, разница между lastTime и текущим временем будет равна тому времени, в течение которого страница была спрятана. Продвигать игру на всё это время было бы глупо и затратно (вспомните разделение времени в методе animate).

Эта функция также преобразовывает временные отрезки в секунды, которыми проще оперировать, чем миллисекундами.

Функция runLevel принимает объект Level, конструктор для display, и, необязательным параметром – функцию. Она выводит уровень в document.body и позволяет пользователю играть на нём. Когда уровень закончен (победа или поражение), runLevel очищает экран, останавливает анимацию, а если задана функция andThen, вызывает её со статусом уровня.

```
var arrows = trackKeys(arrowCodes);

function runLevel(level, Display, andThen) {
  var display = new Display(document.body, level);
  runAnimation(function(step) {
    level.animate(step, arrows);
    display.drawFrame(step);
    if (level.isFinished()) {
      display.clear();
      if (andThen)
        andThen(level.status);
      return false;
    }
  });
}
```

Игра – это последовательность уровней. Когда игрок погибает, уровень начинается заново. Когда уровень закончен, мы переходим на следующий. Это можно выразить следующей функцией, принимающей массив планов уровней (массив строк) и конструктор display

```
function runGame(plans, Display) {
  function startLevel(n) {
    runLevel(new Level(plans[n]), Display, function(stat
us) {
      if (status == "lost")
        startLevel(n);
      else if (n < plans.length - 1)
        startLevel(n + 1);
      else
        console.log("You win!");
    });
  }
  startLevel(0);
}
```

Эти функции демонстрируют необычный стиль программирования. Обе функции runAnimation и runLevel – функции высшего порядка, но не в том стиле, что мы видели в главе 5. Аргумент функций используется, чтобы подготовить вещи, которые произойдут когда-либо в будущем, и функции не возвращают ничего полезного. Их задача – запланировать действия. Оборачивая эти действия в функции, мы сохраняем их как значения, чтобы их можно было вызвать в нужный момент.

Такой стиль программирования обычно называют асинхронным. Обработка событий – тоже пример такого стиля, и мы с ним встретимся ещё не раз, когда будем

работать с задачами, которые могут занять произвольные промежутки времени – например, сетевые запросы в главе 17, или ввод и вывод общего назначения в главе 20.

В переменной GAME_LEVELS хранится набор планов уровней. Такая страница скармливает их в runGame, которая запускает саму игру.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Попробуйте выиграть. Я здорово повеселился, сочиняя их.

Упражнения

Конец игры

По традиции, платформеры дают игроку ограниченное количество жизней, и вычитают по одной каждый раз при гибели игрока. Когда жизни кончаются, игра начинается заново.

Подредактируйте runGame, чтобы она поддерживала жизни. Пусть игрок начинает с трёх.

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // Старая функция runGame – поменяйте её...
  function runGame(plans, Display) {
    function startLevel(n) {
      runLevel(new Level(plans[n]), Display, function(st
atus) {
        if (status == "lost")
          startLevel(n);
        else if (n < plans.length - 1)
          startLevel(n + 1);
        else
          console.log("You win!");
      });
    }
    startLevel(0);
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

Пауза

Сделайте возможным ставить и снимать игру с паузы по нажатию клавиши Esc.

Этого можно достичь, поменяв функцию runLevel, чтобы она использовала другой обработчик событий клавиатуры, и прерывала и возобновляла анимацию по нажатию Esc.

На первый взгляд может показаться, что интерфейс runAnimation не предназначен для этого – но если вы поменяете его вызов из runLevel, всё получится.

Когда получится, можете попробовать ещё кое-что. Мы регистрируем события с клавиатуры не самым лучшим способом. Объект arrows – глобальная переменная, и его обработчики событий находятся в памяти, даже если игра не запущена. Можно сказать, они утекают из системы. Расширьте trackKeys, чтоб можно было разрегистрировать обработчики и затем поменяйте runLevel, чтоб она регистрировала их на старте, и разрегистрировала на финише.

```
<link rel="stylesheet" href="css/game.css">

<body>
<script>
  // Старая функция runLevel – поменяйте её...
  function runLevel(level, Display, andThen) {
    var display = new Display(document.body, level);
    runAnimation(function(step) {
      level.animate(step, arrows);
      display.drawFrame(step);
      if (level.isFinished()) {
        display.clear();
        if (andThen)
          andThen(level.status);
        return false;
      }
    });
  }
  runGame(GAME_LEVELS, DOMDisplay);
</script>
</body>
```

Рисование на холсте

Рисование — это обман.

М.К.Эшер

Браузеры позволяют нам рисовать графику разными способами. Проще всего использовать стили для расположения и расцветки стандартных элементов DOM. Так можно добиться многого, как показал пример игры из предыдущей главы. Добавляя частично прозрачные картинки узлам, мы можем придать им любой нужный вид. Возможно даже поворачивать или искажать узлы через стиль transform.

Но такое использование DOM – не то, для чего он создавался. Некоторые задачи, типа рисования линии между двумя произвольными точками, крайне неудобно выполнять при помощи обычных элементов HTML.

Есть две альтернативы. Первая – SVG, масштабируемая векторная графика, также основанная на DOM, но без участия HTML. SVG – диалект для описания документов, который концентрируется на формах, а не тексте. SVG можно встроить в HTML, или включить через тег `<img>`.

Вторая альтернатива – холст (canvas). Холст – это один элемент DOM, в котором находится картинка. Он предоставляет API для рисования форм на том месте, которое занимает элемент. Разница между холстом и SVG в том, что в SVG хранится начальное описание форм – их можно в любой момент сдвигать или менять размер. Холст же преобразовывает формы в пиксели (цветные точки растра), как только нарисует их, и не запоминает, что эти пиксели из себя представляют. Единственным способом сдвинуть форму на холсте является очистить холст (или ту часть, которая окружает форму) и перерисовать её на другом месте.

SVG

Эта книга не углубляется детально в SVG, но кратко я поясню её работу. В конце главы я вернусь к сравнительным недостаткам методов, которые нужно принять во внимание, выбирая механизм рисования для конкретного применения.

Вот документ HTML, содержащий простую SVG-картинку:

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

Атрибут xmlns меняет пространство имён элемента по умолчанию. Это пространство задаётся через URL и обозначает диалект, на котором мы сейчас говорим. Тэги `<circle>` и `<rect>`, не существующие в HTML, имеют смысл в SVG – они рисуют формы, используя стиль и позицию, заданные их атрибутами.

Они создают элементы DOM так же, как тэги HTML. К примеру, такой код меняет цвет элемента на cyan:

```
var circle = document.querySelector(«circle»);
circle.setAttribute(«fill», «cyan»);
```

Элемент холста canvas

Графику холста можно рисовать на элементе `<canvas>`. Ему можно задать ширину и высоту, таким образом определяя его размер в пикселях.

Новый холст пуст, то есть он полностью прозрачен и показывает нам пустое пространство документа.

Тэг `<canvas>` поддерживает разные стили рисования. Чтобы получить доступ к интерфейсу рисования, сначала нужно создать context – объект, чьи методы предоставляют этот интерфейс. Сейчас есть два широко распространённых стиля рисования: “2d” для двумерной графики и “webgl” для трёхмерной графики при помощи интерфейса OpenGL.

WebGL мы обсуждать не будем, остановимся на двух измерениях. Если вам интересны три измерения, я советую вам окунуться в мир WebGL. Он предоставляет непосредственный доступ к современному графическому железу, поэтому с его помощью можно создавать довольно сложную и эффективную графику прямо из JavaScript.

Context создаётся методом getContext элемента `<canvas>`.

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  var canvas = document.querySelector("canvas");
  var context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

После создания объекта context пример рисует прямоугольник шириной в 100 пикселей и высотой в 50, с координатами левого верхнего угла (10, 10).

Точно как в HTML (и SVG), используемая холстом система координат помещает точку (0, 0) в левый верхний угол, и положительная часть оси Y идёт оттуда вниз. То есть, точка (10,10) на 10 пикселей ниже и правее верхнего левого угла.

Заливка и обводка

В интерфейсе холста форму можно залить, что означает, что занимаемая ею область будет закрашена нужным цветом или шаблоном, или же можно сделать stroke – обвести область линией по краю. Та же терминология используется в SVG.

Метод fillRect заливает прямоугольник. Он принимает координаты левого верхнего угла x,y, затем ширину и высоту. Схожий метод strokeRect рисует периметр прямоугольника.

Больше у методов параметров нет. Цвет заливки, толщина обводки и другие параметры определяются не аргументами метода (как можно было бы ожидать), а свойствами объекта context.

Задав fillStyle, вы меняете способ, которым заливаются формы. Его можно установить в строку, обозначающую цвет, и в любой цвет, который понимает CSS.

Свойство strokeStyle работает так же, но определяет цвет, которым будет нарисована обводка. Толщина линии определяется свойством lineWidth, которое может содержать любое положительное число.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

Когда не заданы атрибуты width или height, им назначаются значения по умолчанию – 300 для ширины и 150 для высоты.

Пути

Путь – последовательность линий. Двумерный холст имеет странный подход к описанию путей. Всё делается через побочные эффекты. Пути – не значения, которые можно хранить или передавать. Вместо этого, если вам что-то надо сделать с путём, вы создаёте последовательность вызовов метода для описания его формы.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.beginPath();
  for (var y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

Пример создаёт путь из нескольких горизонтальных отрезков, и затем обводит их методом stroke. Каждый сегмент, созданный через lineTo, начинается с текущей позиции пути. Эта позиция – обычно конец предыдущего сегмента, если только не было вызова moveTo. В последнем случае следующий сегмент начнётся с позиции, заданной в moveTo.

При заливке пути каждая из форм заливается отдельно. Путь может содержать несколько форм – каждое движение moveTo начинает новую. Но путь должен быть закрытым (начало и конец находятся на одном месте), прежде чем его можно будет закрасить. Если путь не закрыт, от его конца до начала добавляется линия, и заливается форма, очерченная закрытым путём.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

Пример рисует закрашенный треугольник. Заметьте, что непосредственно были нарисованы только две стороны. Третья, от правого нижнего угла обратно к вершине, подразумевается – она не будет закрашена вызовом stroke.

Также можно использовать метод closePath, чтобы принудительно закрыть путь, добавив реальный сегмент до начала пути. Этот сегмент будет закрашен вызовом stroke.

Кривые

Путь может состоять из кривых. Их рисовать посложнее, нежели прямые.

Метод quadraticCurveTo рисует кривую до нужной точки. Для определения кривизны методу даётся контрольная точка вместе с точкой назначения. Представьте, что контрольная точка как бы притягивает линию, задавая кривой кривизну. Линия не проходит через контрольную точку. Вместо этого направления линии в её начальной и конечной точках будут стремиться к контрольной точке. Следующий пример иллюстрирует это:

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) goal=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

Рисуем слева направо квадратичную кривую, у которой контрольная точка задана как (60,10), а затем рисуем два сегмента, проходящие обратно через контрольную точку и начало линии. Результат напоминает эмблему Звёздного пути. Можно увидеть действие контрольной точки: линия, выходящая из начальной и конечной точек, начинается по направлению к контрольной точке, а затем загибается.

Метод bezierCurve рисует схожую кривую. Вместо одной контрольной точки у неё есть две – по одной на каждый из концов кривой. Вот похожий рисунок для иллюстрации поведения такой кривой:

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10,10) control2=(90,10) goal=(50,90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

Две контрольные точки задают направления обоих концов кривой. Чем они дальше от начала или конца, тем сильнее кривая будет выпучиваться в их направлении.

С этими кривыми сложновато работать – не всегда понятно, как искать контрольные точки, которые приведут к нужной вам форме. Иногда их можно вычислить, иногда приходится подбирать методом проб и ошибок.

Дуги, фрагменты кругов, легче в обращении. Метод arcTo принимает целых пять аргументов. Первые четыре – похожи на аргументы quadraticCurveTo. Первая пара задаёт что-то вроде контрольной точки, вторая – место назначения кривой. Пятый задаёт радиус дуги. Метод создаёт скруглённый угол – линию, идущую к контрольной точке, а затем к точке назначения – и скругляет угол

заданным радиусом. Метод arcTo рисует круглую часть, а также линию от точки старта до начала закруглённой части.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.beginPath();
  cx.moveTo(10, 10);
  // control=(90,10) goal=(90,90) radius=20
  cx.arcTo(90, 10, 90, 90, 20);
  cx.moveTo(10, 10);
  // control=(90,10) goal=(90,90) radius=80
  cx.arcTo(90, 10, 90, 90, 80);
  cx.stroke();
</script>
```

arcTo не рисует линию от конца закруглённой части до точки назначения, несмотря на своё название. Её можно закончить через lineTo с такими же координатами.

Чтобы нарисовать круг, можно сделать четыре вызова arcTo, где каждый повёрнут относительно другого на 90 градусов. Но метод arc предоставляет способ проще. Он принимает пару координат центра арки, радиус и начальный и конечный углы.

Два последних параметра могут помочь в рисовании части круга. Углы измеряются в радианах, а не градусах. Это значит, что полный круг имеет угол в 2π, или 2 * Math.PI, что примерно равно 6.28. Угол начинает отсчёт от точки справа от центра, и идёт против часовой стрелки. Чтобы нарисовать полный круг, можно задать начало в 0, а конец больше 2π (к примеру, 7).

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to ½π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

На картинке в результате будет линия слева от круга (первый вызов arc), до левой части четверти круга (второй вызов). Как и другие методы рисования путей, линия дуги соединена с предыдущим сегментом пути. Для начала рисования нового пути надо вызвать moveTo.

Рисуем круговую диаграмму

Представьте, что вы получили работу в ООО «Экономика для всех», и вашим первым заданием будет нарисовать круговую диаграмму удовлетворённости клиентов согласно результатам опроса.

Переменная result содержит массив объектов, представляющих результаты.

```
var results = [
  {name: "Удовлетворён", count: 1043, color: "lightblue"
},
  {name: "Нейтральное", count: 563, color: "lightgreen"}
,
  {name: "Не удовлетворён", count: 510, color: "pink"},
  {name: "Без комментариев", count: 175, color: "silver"
}
];
```

Чтобы нарисовать диаграмму, мы рисуем несколько секторов, каждый из которых делается из арки и пары линий от центра. Угол мы вычисляем, деля полный круг (2π) на общее количество отзывов, и умножая на количество людей, выбравших данный вариант ответа.

```
<canvas width="200" height="200"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  var total = results.reduce(function(sum, choice) {
    return sum + choice.count;
  }, 0);
  // Start at the top
  var currentAngle = -0.5 * Math.PI;
  results.forEach(function(result) {
    var sliceAngle = (result.count / total) * 2 * Math.P
I;
    cx.beginPath();
    // center=100,100, radius=100
    // from current angle, clockwise by slice's angle
    cx.arc(100, 100, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
    cx.fill();
  });
</script>
```

Но диаграмма не расшифровывает значения секторов – это неудобно. Нам надо как-то нарисовать на холсте текст.

Текст

У контекста двумерного холста есть методы fillText и strokeText. Последний можно использовать для обведённых букв, но обычно используется fillText. Он заполняет заданный текст цветом fillColor.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("Я и текст могу рисовать!", 10, 50);
</script>
```

Можно задать размер, стиль и шрифт текста через свойство font. В примере задаётся только размер и шрифт. Можно добавить наклон и жирность в начале строки.

Два последних аргумента fillText (и strokeText) задают позицию, с которой начинается текст. По умолчанию это начало линии, на которой «стоят» буквы – не считая свисающих частей букв типа p и y. Можно менять позицию по горизонтали, задавая свойству textAlign значения «end» или «center», а по вертикали – задавая textBaseline «top», «middle», или «bottom».

В конце главы мы вернёмся к нашей диаграмме.

Изображения

В компьютерной графике проводится различие между векторной и растровой графикой. Первая – то, чем мы занимались в этой главе, рисование при помощи логических описаний форм. Вторая – не задаёт формы, а работает на уровне пикселей.

Метод drawImage позволяет выводить на холст пиксельные данные. Они могут быть взяты из элемента `<img>` или с другого холста, которые не обязательно видны в самом документе. Следующий пример создаёт элемент `<img>` и загружает в него файл изображения. Но он не может сразу начать рисовать при помощи этой картинки, потому что браузер мог не успеть её подгрузить. Для этого мы регистрируем обработчик события “load” и рисуем после загрузки.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  var img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", function() {
    for (var x = 10; x < 200; x += 30)
      cx.drawImage(img, x, 10);
  });
</script>
```

По умолчанию drawImage нарисует картинку оригинального размера. Ему можно задать два дополнительных параметра для изменения ширины и высоты.

Когда drawImage задано девять аргументов, её можно использовать для рисования части изображения. Со второго по пятый аргументы обозначают прямоугольник (x, y, ширина и высота) в исходной картинке, который надо скопировать. С шестого по девятый – прямоугольник на холсте, куда его надо скопировать.

Это можно использовать, чтобы упаковывать несколько спрайтов (элементов картинки или кадров анимации) в один файл изображения, и рисовать только нужные его части. К примеру, есть у нас картинка игрового персонажа в разных позах:

Перебирая позы, мы можем вывести анимацию идущего персонажа.

Для анимации на холсте пригодится метод clearRect. Он напоминает fillRect, но вместо окраски прямоугольника он делает его прозрачным, удаляя предыдущие пиксели.

Мы знаем, что каждый спрайт шириной 24 и высотой 30 пикселей. Следующий код загружает картинку и задаёт интервал для рисования следующих кадров:

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  var img = document.createElement("img");
  img.src = "img/player.png";
  var spriteW = 24, spriteH = 30;
  img.addEventListener("load", function() {
    var cycle = 0;
    setInterval(function() {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // source rectangle
                   cycle * spriteW, 0, spriteW, spriteH,
                   // destination rectangle
                   0,               0, spriteW, spriteH)
;
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

Переменная cycle отслеживает позицию в анимации. Каждый кадр она увеличивается и по достижению 7 начинает сначала, используя оператор деления с

остатком. Она используется для подсчёта координаты x, на которой в изображении находится спрайт с нужной позой.

Преобразования

А что, если нам надо, чтобы персонаж шёл влево, а не вправо? Мы могли бы добавить ещё один набор спрайтов. Но мы также можем сказать холсту, чтоб он рисовал картинку зеркально.

Вызов метода scale приведёт к тому, что все последующие рисунки будут масштабированы. Он принимает два параметра – масштаб по горизонтали и по вертикали.

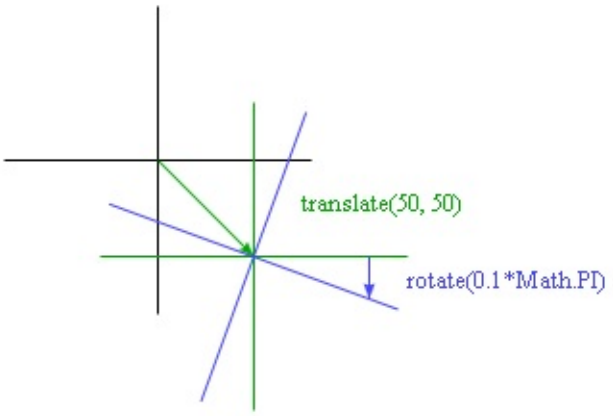
```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

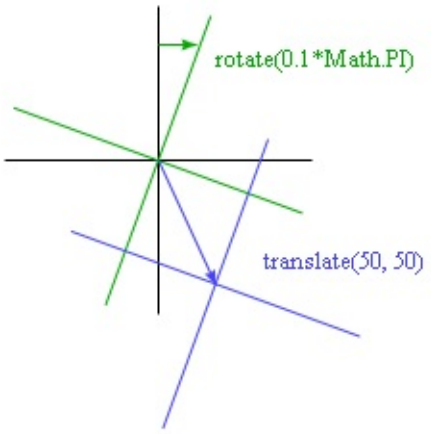
Масштабирование растягивает или сжимает все параметры картинки, включая ширину линии по заданным параметрам. Масштабирование с отрицательным параметром переворачивает картинку зеркально. Переворот происходит вокруг точки (0, 0), что означает, что направление системы координат тоже поменяется. При применении горизонтального масштаба -1, форма, нарисованная на позиции x = 100, будет нарисована там, где раньше была позиция -100.

Значит, для отзеркаливания картинки мы не можем просто добавить cx.scale(-1, 1) перед вызовом drawImage – наша картинка уедет с холста и не будет видна. Можно было бы подправить координаты, передаваемые в drawImage, чтобы компенсировать этот сдвиг. Другой вариант действий, когда код рисования ничего не знает про масштабирование, заключается в изменении направления оси.

Есть несколько других методов кроме масштабирования, влияющих на координатную систему холста. Нарисованные формы можно поворачивать методом rotate и сдвигать методом translate. Интересно, что все трансформации накапливаются, то есть каждая последующая происходит относительно предыдущих.

Значит, если мы дважды сдвинем изображение на 10 пикселей по горизонтали, то всё будет нарисовано на 20 пикселей правее. Если мы сначала сдвинем начало отсчёта на (50, 50), а затем повернём всё на 20 градусов (0.1π радиан), поворот произойдёт вокруг точки (50, 50).



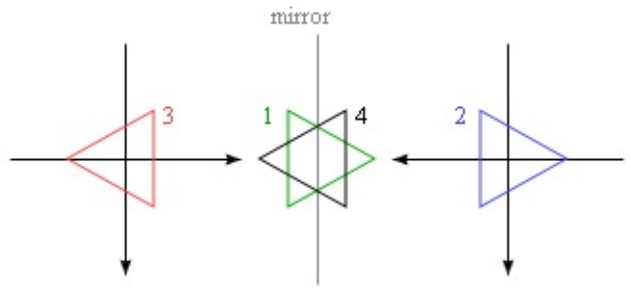


А если мы сначала повернём всё на 20 градусов, а уже затем сдвинем на (50, 50), то преобразование случится в повёрнутой системе координат, что приведёт к иному результату. Порядок преобразований имеет значение.

Чтобы отзеркалить картинку относительно вертикали на заданной позиции x, мы делаем следующее:

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

Мы сдвигаем ось Y туда, где нам нужно расположить наше зеркало, проводим отзеркаливание, и сдвигаем ось Y обратно на полагающееся место в зеркальной вселенной. Следующий рисунок объясняет, как это работает:



Тут показаны системы координат до и после отзеркаливания относительно центральной линии. Если мы нарисуем треугольник в положительной полуплоскости относительно Y, он будет находиться на месте треугольника 1. Вызов flipHorizontally сначала сдвигает его вправо, на место треугольника 2. Затем происходит масштабирование, и треугольник оказывается на месте 3. Он должен быть не там, если нам надо отзеркалить его относительно заданной линии. Второй вызов translate исправляет это – он «отменяет» изначальный сдвиг и помещает треугольник на позицию 4.

Теперь можно нарисовать отзеркаленного персонажа на позиции (100, 0), перевернув мир относительно вертикали изображения персонажа.

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  var img = document.createElement("img");
  img.src = "img/player.png";
  var spriteW = 24, spriteH = 30;
  img.addEventListener("load", function() {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

Хранение и очистка преобразований

Преобразования накапливаются. Всё, что мы рисуем после рисования отзеркаленного персонажа, также будет зеркальным. Это может стать проблемой.

Возможно сохранить текущее преобразование, порисовать что-то, а затем вернуть старое состояние. Так должна поступать функция, делающая временное преобразование системы координат. Сначала мы сохраняем то преобразование, которое использовал код, вызвавший эту функцию. Затем функция отрабатывает на

основе преобразований, проведённых на этот момент, и, возможно, добавляет новые. И в конце мы возвращаем преобразования к началу.

Этим занимаются методы save и restore двумерного холста. По сути, они хранят стек состояний преобразований. При вызове save в стек добавляется текущее состояние, а при restore берётся состояние сверху стека и применяется в качестве текущего контекста всех преобразований.

Функция branch в примере показывает, что можно сделать с функцией, которая выполняет преобразования и вызывает другую функцию (в данном случае, саму себя), которая продолжает рисовать с заданными преобразованиями.

Функция рисует древовидную структуру, рисуя линию, потом передвигая центр координат на конец линии, и вызывая себя затем дважды – сначала, повернув влево, а затем вправо. Каждый вызов уменьшает длину ветви, и рекурсия останавливается, когда длина падает меньше 8.

```
<canvas width="600" height="300"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

Если бы не было вызовов save и restore, второй рекурсивный вызов branch начинал бы с позиции и поворота, созданных первым. Он был бы соединён не с текущей веткой, а внутренней правой веткой, нарисованной первым вызовом. В результате получается тоже интересная форма, но уже не древовидная.

Назад к игре

Теперь мы знаем о холсте достаточно, чтобы начать разработку графической системы для игры из предыдущей главы. Новая система не будет показывать только цветные квадратики. Мы будем использовать drawImage для рисования картинок, представляющих элементы игры.

Мы определим тип объекта CanvasDisplay, который будет поддерживать тот же интерфейс, что и DOMDisplay из главы 15, а именно, методы drawFrame and clear.

Объект хранит больше информации, чем DOMDisplay. Вместо использования позиции прокрутки элемента DOM, он отслеживает окно просмотра, которое сообщает, какую часть уровня мы сейчас видим. Также он отслеживает время и использует это, чтобы решить, какой кадр анимации показывать. И ещё он хранит свойство flipPlayer, чтобы даже когда игрок стоял на месте, он был повёрнут в ту сторону, в которую шёл в последний раз.

```
function CanvasDisplay(parent, level) {
  this.canvas = document.createElement("canvas");
  this.canvas.width = Math.min(600, level.width * scale)
;
  this.canvas.height = Math.min(450, level.height * scal
e);
  parent.appendChild(this.canvas);
  this.cx = this.canvas.getContext("2d");

  this.level = level;
  this.animationTime = 0;
  this.flipPlayer = false;

  this.viewport = {
    left: 0,
    top: 0,
    width: this.canvas.width / scale,
    height: this.canvas.height / scale
  };

  this.drawFrame(0);
}

CanvasDisplay.prototype.clear = function() {
  this.canvas.parentNode.removeChild(this.canvas);
};
```

В 15 главе мы передавали размер шага в drawFrame из-за счётчика animationTime, несмотря на то, что DOMDisplay его не использовал. Наша новая функция

drawFrame использует его для отсчёта времени, чтобы переключаться между кадрами анимации в зависимости от текущего времени.

```
CanvasDisplay.prototype.drawFrame = function(step) {
  this.animationTime += step;

  this.updateViewport();
  this.clearDisplay();
  this.drawBackground();
  this.drawActors();
};
```

Кроме отслеживания времени, метод обновляет окно просмотра текущей позиции игрока, заполняет холст цветом фона, и рисует фон и актёров. Заметьте, что всё происходит не так, как в главе 15, где мы рисовали фон один раз, а затем прокручивали оборачивающий элемент DOM для перемещения по нему.

Так как формы на холсте – всего лишь пиксели, после их отрисовки их нельзя сдвинуть (или убрать). Единственным способом обновить холст будет очистить его и перерисовать сцену.

Метод updateViewport похож на метод scrollPlayerIntoView из DOMDisplay. Он проверяет, не находится ли игрок слишком близко к краю экрана и двигает окно просмотра, если это случается.

```
CanvasDisplay.prototype.updateViewport = function() {
  var view = this.viewport, margin = view.width / 3;
  var player = this.level.player;
  var center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin)
    view.left = Math.max(center.x - margin, 0);
  else if (center.x > view.left + view.width - margin)
    view.left = Math.min(center.x + margin - view.width,
                         this.level.width - view.width);
  if (center.y < view.top + margin)
    view.top = Math.max(center.y - margin, 0);
  else if (center.y > view.top + view.height - margin)
    view.top = Math.min(center.y + margin - view.height,
                        this.level.height - view.height)
;
};
```

Вызовы Math.max и Math.min гарантируют, что окно просмотра не будет показывать пространство за пределами уровня. Math.max(x, 0) гарантирует, что итоговое число не меньше нуля. Сходным образом Math.min гарантирует, что значение не превысит заданную границу.

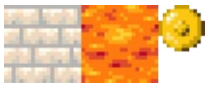
При очистке дисплея мы используем другой цвет, в зависимости от того, выиграна игра или проиграна.

```
CanvasDisplay.prototype.clearDisplay = function() {
  if (this.level.status == "won")
    this.cx.fillStyle = "rgb(68, 191, 255)";
  else if (this.level.status == "lost")
    this.cx.fillStyle = "rgb(44, 136, 214)";
  else
    this.cx.fillStyle = "rgb(52, 166, 251)";
  this.cx.fillRect(0, 0,
                   this.canvas.width, this.canvas.height
);
};
```

Для рисования фона мы пробегаемся по клеткам, видимым в текущем окне просмотра, используя тот же фокус, что и в obstacleAt в предыдущей главе.

```
var otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function() {
  var view = this.viewport;
  var xStart = Math.floor(view.left);
  var xEnd = Math.ceil(view.left + view.width);
  var yStart = Math.floor(view.top);
  var yEnd = Math.ceil(view.top + view.height);

  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      var tile = this.level.grid[y][x];
      if (tile == null) continue;
      var screenX = (x - view.left) * scale;
      var screenY = (y - view.top) * scale;
      var tileX = tile == "lava" ? scale : 0;
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
                        screenX, screenY, scale, scale);
    }
  }
};
```

Непустые клетки (null) рисуются через drawImage. Изображение otherSprites содержит картинки для элементов, не относящихся к игроку. Слева направо — это стена, лава и монетка.



Спрайты для нашей игры

Клетки фона 20x20 пикселей, так как мы используем ту же шкалу, что была в DOMDisplay. Значит, сдвиг клеток лавы 20 (значение переменной scale), а сдвиг стен 0.

Мы не ждём загрузки спрайта. Вызов drawImage с незагруженной пока картинкой ничего не сделает. Поэтому, на нескольких первых кадрах игра может быть отрисована неверно, но это не так уж критично. Так как мы обновляем экран, правильная сцена появится сразу после окончания загрузки.

Наш персонаж будет использован в качестве игрока. Код его отрисовки должен выбирать правильный спрайт и направление, зависящее от текущего движения игрока. Первые восемь спрайтов содержат анимацию ходьбы. Когда игрок передвигается по полу, мы перебираем их в зависимости от свойства animationTime объекта display. Оно измеряется в секундах, а нам надо менять кадры 12 раз в секунду, поэтому мы умножаем время на 12. Когда игрок стоит, мы рисуем девятый спрайт. В прыжках, которые мы распознаём по тому, что вертикальная скорость отлична от нуля, мы рисуем десятый, самый правый спрайт.

Поскольку спрайты чуть шире ширины объекта игрока – 24 пикселя вместо 16, чтобы было место для рук и ног, метод должен подправлять координату x и ширину на заданное число (playerXOverlap).

```
var playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
var playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(x, y, widt
h,
                                                height) {
  var sprite = 8, player = this.level.player;
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0)
    this.flipPlayer = player.speed.x < 0;

  if (player.speed.y != 0)
    sprite = 9;
  else if (player.speed.x != 0)
    sprite = Math.floor(this.animationTime * 12) % 8;

  this.cx.save();
  if (this.flipPlayer)
    flipHorizontally(this.cx, x + width / 2);

  this.cx.drawImage(playerSprites,
                    sprite * width, 0, width, height,
                    x,              y, width, height);

  this.cx.restore();
};
```

Метод drawPlayer вызывается через drawActors, который рисует всех актёров в игре.

```
CanvasDisplay.prototype.drawActors = function() {
  this.level.actors.forEach(function(actor) {
    var width = actor.size.x * scale;
    var height = actor.size.y * scale;
    var x = (actor.pos.x - this.viewport.left) * scale;
    var y = (actor.pos.y - this.viewport.top) * scale;
    if (actor.type == "player") {
      this.drawPlayer(x, y, width, height);
    } else {
      var tileX = (actor.type == "coin" ? 2 : 1) * scale
;
      this.cx.drawImage(otherSprites,
                        tileX, 0, width, height,
                        x,     y, width, height);
    }
  }, this);
};
```

При отрисовке чего-либо кроме игрока мы смотрим на его тип, чтобы найти смещение для нужного спрайта. Лава находится по смещению 20, монета – 40.

Нужно вычитать позицию окна просмотра при подсчёте позиции актёра, так как точка (0, 0) нашего холста соответствует левой верхней точке окна просмотра, а не левой верхней точке уровня. Ещё мы могли бы использовать для этой цели translate.

Следующий маленький документ подключает новый display в runGame:

```
<body>
  <script>
    runGame(GAME_LEVELS, CanvasDisplay);
  </script>
</body>
```

Выбор графического интерфейса

Когда вам нужно создавать графику в браузере, у вас есть выбор – HTML, SVG и холст. Не существует идеального подхода для всех ситуаций. У каждого варианта есть плюсы и минусы.

Чистый HTML прост. Он хорошо сочетается с текстом. SVG и холст позволяют рисовать текст, но не помогают в его расположении и не делают переносов, когда он занимает более одной линии. В HTML просто включать блоки текста.

SVG можно использовать для создания чёткой графики, которая выглядит хорошо при любом увеличении. Он сложнее обычного HTML, но и гораздо мощнее.

SVG и HTML строят структуру данных (DOM), которая представляет картинку. Это позволяет изменять элементы после того, как они нарисованы. Если вам надо периодически менять небольшую часть большой картинки в ответ на действия пользователя или в качестве анимации, на холсте это будет делать очень затратно. DOM позволяет регистрировать обработчики событий мыши на любом элементе картинки (даже на формах, нарисованных через SVG). С холстом это не пройдёт.

Но пиксельный подход холста имеет преимущество при рисовании большого количества небольших элементов. Он не строит структуру данных, а просто рисует на той же самой поверхности пиксели, что снижает затратность в пересчёте на формы.

Есть ещё факторы, типа создания сцены попиксельно (например, при использовании трассировки лучей) или постобработка картинки в JavaScript (размытие или искажение), которые можно сделать только при помощи попиксельного рисования.

В некоторых случаях можно комбинировать эти техники. Например, можно нарисовать граф через SVG или холст, а текстовую информацию показывать, позиционируя элементы HTML поверх картинки.

Для непривередливых приложений неважно, какой вы используете интерфейс. Дисплей, построенный нами для нашей игры, можно сделать любым из трёх графических способов, так как он не выводит текст и не обрабатывает нажатия мыши, и не обслуживает огромное количество элементов.

Итог

В этой главе мы обсудили техники рисования графики в браузере, сконцентрировавшись на элементе `<canvas>`.

Узел холста представляет область документа, где программа может рисовать. Это делается через объект context, создаваемый методом getContext. Интерфейс двумерного рисования позволяет закрашивать и обводить разные формы. Свойство fillStyle задаёт заливку форм. Свойства strokeStyle и lineWidth управляют тем, как рисуются линии.

Прямоугольники и куски текста можно рисовать одним вызовом метода. Методы fillRect и strokeRect рисуют прямоугольники, а fillText и strokeText выводят текст. Для создания произвольных форм нам нужно строить пути.

Вызов beginPath начинает путь. Несколько методов добавляют линии и кривые к текущему пути. Например, lineTo добавляет прямую. Когда путь закончен, его можно заполить методом fill или обвести методом stroke.

Перемещение пикселей с картинки или другого холста на наш делается методом drawImage. По умолчанию, он рисует всю исходную картинку, но с большим количеством параметров вы можете скопировать нужный участок изображения. В нашей игре мы использовали эту возможность, копируя разные позы игрового персонажа из частей картинки, содержавшей много поз.

Перемещения позволяют рисовать форму, ориентированную по-разному. Двумерный контекст хранит текущее преобразование, которое можно менять через методы translate, scale и rotate. Это повлияет на все остальные операции рисования. Текущее состояние преобразований можно сохранить методом save и восстановить методом restore.

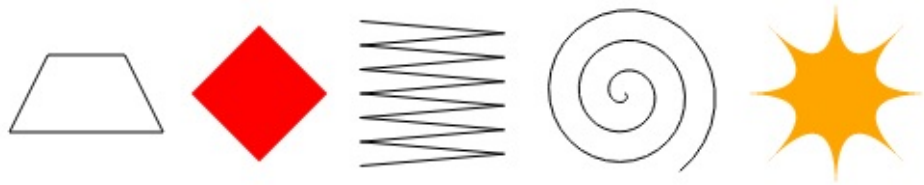
При рисовании анимаций на холсте можно использовать метод clearRect для очистки частей холста перед перерисовкой.

Упражнения

Формы

Напишите программу, рисующую следующие фигуры:

1. трапецию
2. красный ромб
3. зигзаг
4. спираль из 100 отрезков
5. жёлтую звезду



Рисуя две последних, консультируйтесь с описаниями функций Math.cos и Math.sin из главы 13, которая описывает получение координат на круге с их использованием.

Рекомендую для каждой формы сделать функцию. Передавайте позицию и другие свойства, типа размера, количества точек. Вариант со вписыванием нужных чисел прямо в код обычно труднее читать и изменять.

```
<canvas width="600" height="200"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");

  // Ваш код
</script>
```

Круговая диаграмма

Ранее мы видели пример программы для рисования круговой диаграммы. Поменяйте её, чтобы имя каждой категории было показано рядом с куском, который её представляет. Попробуйте отыскать симпатичный вариант автоматического позиционирования текста, который бы работал и на других наборах данных. Можно предположить, что нет категории меньше 5% (чтобы текст не громоздился друг на друга).

Вам снова могут понадобиться Math.sin и Math.cos.

```
<canvas width="600" height="300"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");
  var total = results.reduce(function(sum, choice) {
    return sum + choice.count;
  }, 0);

  var currentAngle = -0.5 * Math.PI;
  var centerX = 300, centerY = 150;
  // Добавьте код для вывода меток
  results.forEach(function(result) {
    var sliceAngle = (result.count / total) * 2 * Math.P
I;
    cx.beginPath();
    cx.arc(centerX, centerY, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(centerX, centerY);
    cx.fillStyle = result.color;
    cx.fill();
  });
</script>
```

Прыгающий мячик

Используйте технику requestAnimationFrame из глав 13 и 15 для рисования прямоугольника с прыгающим внутри мячом. Мяч двигается с постоянной скоростью и отскакивает от сторон прямоугольника при соударении.

```
<canvas width="400" height="400"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("
2d");

  var lastTime = null;
  function frame(time) {
    if (lastTime != null)
      updateAnimation(Math.min(100, time - lastTime) / 1
000);
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);

  function updateAnimation(step) {
    // Ваш код
  }
</script>
```

Предварительно рассчитанное отзеркаливание

Преобразования, к сожалению, замедляют рисование растровых изображений. Для векторной графики эффект не так заметен, потому что преобразованиям подвергаются всего лишь несколько точек, после чего рисование продолжается как обычно. Для растра позиция каждого пикселя должна быть преобразована, и хотя

возможно, что браузеры в будущем будут делать это по-умному, это приводит к ненужному увеличению времени на отрисовку растра.

В нашей игре, где есть всего один преобразуемый спрайт, это не проблема. Но представьте, что вам надо рисовать сотни персонажей или тысячи вращающихся частиц от взрыва.

Подумайте, как можно было бы рисовать инвертированного персонажа без подгрузок дополнительных файлов и без постоянных преобразований вызовов drawImage.

HTTP

Мечта, ради которой создавалась Сеть – это общее информационное пространство, в котором мы общаемся, делясь информацией. Его универсальность является его неотъемлемой частью: ссылка в гипертексте может вести куда угодно, будь то персональная, локальная или глобальная информация, черновик или выверенный текст.

Тим Бернес-Ли, Всемирная паутина: Очень короткая личная история

Протокол

Если в адресной строке браузера набрать eloquentjavascript.net/17_http.html, браузер сначала распознает адрес сервера, связанный с именем eloquentjavascript.net и попробует открыть TCP соединение по 80 порту – порт для HTTP по умолчанию. Если сервер существует и принимает соединение, браузер отправляет что-то вроде:

```
GET /17_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Название браузера
```

Сервер отвечает по тому же соединению:

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT

<!doctype html>
… остаток документа
```

Браузер берёт ту часть, что идёт за ответом после пустой строки и показывает её в виде HTML-документа.

Информация, отправленная клиентом, называется запросом. Он начинается со строки:

```
GET /17_http.html HTTP/1.1
```

Первое слово – метод запроса. GET означает, что нам нужно получить определённый ресурс. Другие распространённые методы – DELETE для удаления, PUT для замещения и POST для отправки информации. Заметьте, что сервер не обязан выполнять каждый

полученный запрос. Если вы выберете случайный сайт и скажете ему DELETE главную страницу – он, скорее всего, откажется.

Часть после названия метода – путь к ресурсу, к которому отправлен запрос. В простейшем случае, ресурс – просто файл на сервере, но протокол не ограничивается этой возможностью. Ресурс может быть чем угодно, что можно передать в качестве файла. Многие серверы создают ответы на лету. К примеру, если вы откроете twitter.com/marijnjh, сервер посмотрит в базе данных пользователя marijnjh, и если найдёт – создаст страницу профиля этого пользователя.

После пути к ресурсу первая строка запроса упоминает HTTP/1.1, чтобы сообщить о версии HTTP – протокола, которую она использует.

Ответ сервера также начинается с версии протокола, за которой идёт статус ответа – сначала код из трёх цифр, затем строчка.

```
HTTP/1.1 200 OK
```

Коды статуса, начинающиеся с 2, обозначают успешные запросы. Коды, начинающиеся с 4, означают, что что-то пошло не так. 404 – самый знаменитый статус HTTP,

обозначающий, что запрошенный ресурс не найден. Коды, начинающиеся с 5, обозначают, что на сервере произошла ошибка, но не по вине запроса.

За первой строкой запроса или ответа может идти любое число строк заголовка. Это строки в виде “имя: значение”, которые обозначают дополнительную информацию о запросе или ответе. Эти заголовки были включены в пример:

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT
```

Тут определяется размер и тип документа, полученного в ответ. В данном случае это HTML-документ размером 65’585 байт. Также тут указано, когда документ был изменён последний раз.

По большей части клиент или сервер определяют, какие заголовки необходимо включать в запрос или ответ, хотя некоторые заголовки обязательны. К примеру, Host, обозначающий имя хоста, должен быть включён в запрос, потому что один сервер может обслуживать много имён хостов на одном ip-адресе, и без этого заголовка сервер не узнает, с каким хостом клиент пытается общаться.

После заголовков, как запрос, так и ответ могут указать пустую строку, за которой следует тело, содержащее передаваемые данные. Запросы GET и DELETE не пересылают дополнительных данных, а PUT и POST пересылают. Некоторые ответы, например, сообщения об ошибке, не требуют наличия тела.

Браузер и HTTP

Как мы видели в примере, браузер отправляет запрос, когда мы вводим URL в адресную строку. Когда в полученном HTML документе содержатся упоминания других файлов, такие, как картинки или файлы JavaScript, они тоже запрашиваются с сервера.

Веб-сайт средней руки легко может содержать от 10 до 200 ресурсов. Чтобы иметь возможность запросить их побыстрее, браузеры делают несколько запросов одновременно, а не ждут окончания запросов одного за другим. Такие документы всегда запрашиваются через запросы GET.

На страницах HTML могут быть формы, которые позволяют пользователям вписывать информацию и отправлять её на сервер. Вот пример формы:

```
<form method="GET" action="example/message.html">
  <p>Имя: <input type="text" name="name"></p>
  <p>Сообщение:<br><textarea name="message"></textarea></
p>
  <p><button type="submit">Отправить </button></p>
</form>
```

Код описывает форму с двумя полями: маленькое запрашивает имя, а большое – сообщение. При нажатии кнопки «Отправить» информация из этих полей будет закодирована в строку запроса (query string). Когда атрибут method элемента равен GET, или когда он вообще не указан, строка запроса помещается в URL из поля action, и браузер делает GET-запрос с этим URL.

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/
1.1
```

Начало строки запроса обозначено знаком вопроса. После этого идут пары имён и значений, соответствующие атрибуту name полей формы и содержимому этих полей. Амперсанд (&) используется для их разделения.

Сообщение, отправляемое в примере, содержит строку “Yes?”, хотя знак вопроса и заменён каким-то странным кодом. Некоторые символы в строке запроса нужно

экранировать (escape). Знак вопроса в том числе, и он представляется кодом %3F. Есть какое-то неписаное правило, по которому у каждого формата должен быть способ экранировать символы. Это правило под названием кодирование URL использует процент, за которым идут две шестнадцатеричные цифры, которые представляют код символа. 3F в десятичной системе будет 63, и это код знака вопроса. У JavaScript есть функции encodeURIComponent и decodeURIComponent для кодирования и раскодирования.

```
console.log(encodeURIComponent("Hello & goodbye"));
// → Hello%20%26%20goodbye
console.log(decodeURIComponent("Hello%20%26%20goodbye"))
;
// → Hello & goodbye
```

Если мы поменяем атрибут method в форме в предыдущем примере на POST, запрос HTTP с отправкой формы пройдёт при помощи метода POST, который отправит строку запроса в теле запроса, вместо добавления её к URL.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

По соглашению метод GET используется для запросов, не имеющих побочных эффектов, таких, как поиск. Запросы, которые что-то меняют на сервере – создают новый аккаунт или размещают сообщение, должны отправляться методом POST. Клиентские программы типа браузера знают, что просто так делать запросы формата POST не нужно, и иногда незаметно для пользователя делают запросы GET – к примеру, чтобы загрузить заранее контент, который может вскоре понадобиться пользователю.

В следующей главе мы вернёмся к формам и поговорим про то, как мы можем делать их при помощи JavaScript.

XMLHttpRequest

Интерфейс, через который JavaScript в браузере может делать HTTP-запросы, называется XMLHttpRequest (заметьте, как прыгает размер букв). Он был разработан в Microsoft для браузера Internet Explorer в конце 1990-х. В это время формат XML был очень популярным в мире бизнес-программ – а в этом мире Microsoft всегда чувствовал себя, как дома. Он был настолько популярным, что аббревиатура XML была пришпилена перед названием интерфейса для работы с HTTP, хотя последний с XML вообще не связан.

И всё же имя не полностью бессмысленное. Интерфейс позволяет разбирать вам ответы, как если бы это были документы XML. Смешивать две разные вещи (запрос и разбор ответа) в одну – это, конечно, отвратительный дизайн, но что поделаешь.

Когда интерфейс XMLHttpRequest был добавлен в Internet Explorer, стало можно делать вещи, которые раньше было делать очень сложно. К примеру, сайты стали показывать списки из подсказок, пока пользователь вводит что-либо в текстовое поле. Скрипт отправляет текст на сервер через HTTP одновременно с набором текста пользователем. Сервер, у которого есть база данных для возможных вариантов ввода, ищет среди записей подходящие и возвращает их назад для показа. Это выглядело очень круто – люди до этого привыкли ждать перезагрузки всей страницы после каждого взаимодействия с сайтом.

Другой важный браузер того времени Mozilla (позже Firefox), не хотел отставать. Чтобы разрешить делать сходные вещи, Mozilla скопировал интерфейс вместе с названием. Следующее поколение браузеров последовало этому примеру, и сегодня XMLHttpRequest является стандартом de facto.

Отправка запроса

Чтобы отправить простой запрос, мы создаём объект запроса с конструктором XMLHttpRequest и вызываем методы open и send.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.responseText);
// → This is the content of data.txt
```

Метод open настраивает запрос. В нашем случае мы решили сделать GET запрос на файл example/data.txt. URL, не начинающиеся с названия протокола (например, http:) называются относительными, то есть они интерпретируются относительно текущего документа. Когда они начинаются со слеша (/), они заменяют текущий путь – часть после названия сервера. В ином случае часть текущего пути вплоть до последнего слеша помещается перед относительным URL.

После открытия запроса мы можем отправить его методом send. Аргументом служит тело запроса. Для запросов GET используется null. Если третий аргумент для open был false, то send вернётся только после того, как был получен ответ на наш запрос. Для получения тела ответа мы можем прочесть свойство responseText объекта request.

Можно получить из объекта response и другую информацию. Код статуса доступен в свойстве status, а текст статуса – в statusText. Заголовки можно прочесть из getResponseHeader.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.status, req.statusText);
// → 200 OK
console.log(req.getResponseHeader("content-type"));
// → text/plain
```

Названия заголовков не чувствительны к регистру. Они обычно пишутся с заглавной буквы в начале каждого слова, например “Content-Type”, но “content-type” или “cOnTeNt-TyPe” будут описывать один и тот же заголовок.

Браузер сам добавит некоторые заголовки, такие, как “Host” и другие, которые нужны серверу, чтобы вычислить размер тела. Но вы можете добавлять свои собственные заголовки методом setRequestHeader. Это нужно для особых случаев и требует содействия сервера, к которому вы обращаетесь – он волен игнорировать заголовки, которые он не умеет обрабатывать.

Асинхронные запросы

В примере запрос был окончен, когда заканчивается вызов send. Это удобно потому, что свойства вроде responseText становятся доступными сразу. Но это значит, что программа наша будет ожидать, пока браузер и сервер общаются меж собой. При плохой связи, слабом сервере или большом файле это может занять длительное время. Это плохо ещё и потому, что никакие обработчики событий не сработают, пока программа находится в режиме ожидания – документ перестанет реагировать на действия пользователя.

Если третьим аргументом open мы передадим true, запрос будет асинхронным. Это значит, что при вызове send запрос ставится в очередь на отправку. Программа продолжает работать, а браузер позаботиться об отправке и получении данных в фоне.

Но пока запрос обрабатывается, мы не получим ответ. Нам нужен механизм оповещения о том, что данные поступили и готовы. Для этого нам нужно будет слушать событие “load”.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
  console.log("Done:", req.status);
});
req.send(null);
```

Так же, как вызов requestAnimationFrame в главе 15, этот код вынуждает нас использовать асинхронный стиль программирования, оборачивая в функцию тот код, который должен быть выполнен после запроса, и устраивая вызов этой функции в нужное время. Мы вернёмся к этому позже.

Получение данных XML

Когда ресурс, возвращённый объектом XMLHttpRequest, является документом XML, свойство responseXML будет содержать разобранное представление о документе. Оно работает схожим с DOM образом, за исключением того, что у него нет присущей HTML функциональности навроде свойства style. Объект, содержащийся в responseXML, соответствует объекту document. Его свойство documentElement ссылается на внешний тег документа XML. В следующем документе (example/fruit.xml) таким тегом будет :

```
<fruits>
  <fruit name="banana" color="yellow"/>
  <fruit name="lemon" color="yellow"/>
  <fruit name="cherry" color="red"/>
</fruits>
```

Мы можем получить такой файл следующим образом:

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.xml", false);
req.send(null);
console.log(req.responseXML.querySelectorAll("fruit").le
ngth);
// → 3
```

Документы XML можно использовать для обмена с сервером структурированной информацией. Их форма – вложенные теги – хорошо подходит для хранения большинства данных, ну или по крайней мере лучше, чем текстовые файлы. Интерфейс DOM неуклюж в плане извлечения информации, и XML документы получаются довольно многословными. Обычно лучше общаться при помощи данных в формате JSON, которые проще читать и писать – как программам, так и людям.

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.json", false);
req.send(null);
console.log(JSON.parse(req.responseText));
// → {banana: "yellow", lemon: "yellow", cherry: "red"}
```

Песочница для HTTP

HTTP-запросы из веб-страницы вызывают вопросы касаемо безопасности. Человек, контролирующий скрипт, может иметь интересы отличные от интересов пользователя, на чьём компьютере он запущен. Конкретно, если я зашёл на сайт themafia.org, я не хочу, чтобы их скрипты могли делать запросы к mybank.com, используя информацию моего браузера в качестве идентификатора, и давая команду отправить все мои деньги на какой-нибудь счёт мафии.

Вебсайты могут защитить себя от подобных атак, но для этого требуются определённые усилия, и многие сайты с этим не справляются. Из-за этого браузеры защищают их, запрещая скриптам делать запросы к другим доменам (именам вроде themafia.org и mybank.com).

Это может мешать разработке систем, которым надо иметь доступ к разным доменам по уважительной причине. К счастью, сервер может включать в ответ следующий заголовок, поясняя браузерам, что запрос может прийти с других доменов:

Access-Control-Allow-Origin: *

Абстрагируем запросы

В главе 10 в нашей реализации модульной системы AMD мы использовали гипотетическую функцию backgroundReadFile. Она принимала имя файла и функцию, и вызывала эту функцию после прочтения содержимого файла. Вот простая реализация этой функции:

```
function backgroundReadFile(url, callback) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(req.responseText);
  });
  req.send(null);
}
```

Простая абстракция упрощает использование XMLHttpRequest для простых GET-запросов. Если вы пишете программу, которая делает HTTP-запросы, будет неплохо использовать вспомогательную функцию, чтобы вам не приходилось всё время повторять уродливый шаблон XMLHttpRequest.

Аргумент callback (обратный вызов) – термин, часто использующийся для описания подобных функций. Функция обратного вызова передаётся в другой код, чтобы он мог позвать нас обратно позже.

Несложно написать свою вспомогательную функцию HTTP, специально скроенную под вашу программу. Предыдущая делает только GET-запросы, и не даёт нам контроля над заголовками или телом запроса. Можно написать ещё один вариант для запроса POST, или более общий, поддерживающий разные запросы. Многие библиотеки JavaScript предлагают обёртки для XMLHttpRequest.

Основная проблема с приведённой обёрткой – обработка ошибок. Когда запрос возвращает код статуса, обозначающий ошибку (от 400 и выше), он ничего не делает. В некоторых случаях это нормально, но представьте, что мы поставили индикатор загрузки на странице, показывающий, что мы получаем информацию. Если запрос не удался, потому что сервер упал или соединение прервано, страница будет делать вид, что она чем-то занята. Пользователь подождёт немного, потом ему надоест и он решит, что сайт какой-то дурацкий.

Нам нужен вариант, в котором мы получаем предупреждение о неудачном запросе, чтобы мы могли принять меры. Например, мы можем убрать сообщение о загрузке и сообщить пользователю, что что-то пошло не так.

Обработка ошибок в асинхронном коде ещё сложнее, чем в синхронном. Поскольку нам часто приходится отделять часть работы и размещать её в функции обратного вызова, область видимости блока try теряет смысл. В следующем коде исключение не будет поймано, потому что вызов backgroundReadFile возвращается сразу же. Затем управление уходит из блока try, и функция из него не будет вызвана.

```
try {
  backgroundReadFile("example/data.txt", function(text)
{
    if (text != "expected")
      throw new Error("That was unexpected");
  });
} catch (e) {
  console.log("Hello from the catch block");
}
```

Чтобы обрабатывать неудачные запросы, придётся передавать дополнительную функцию в нашу обёртку, и вызывать её в случае проблем. Другой вариант – использовать соглашение, что если запрос не удался, то в функцию обратного вызова передаётся дополнительный аргумент с описанием проблемы. Пример:

```
function getURL(url, callback) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(req.responseText);
    else
      callback(null, new Error("Request failed: " +
                               req.statusText));
  });
  req.addEventListener("error", function() {
    callback(null, new Error("Network error"));
  });
  req.send(null);
}
```

Мы добавили обработчик события error, который сработает при проблеме с вызовом. Также мы вызываем функцию обратного вызова с аргументом error, когда запрос завершается со статусом, говорящим об ошибке.

Код, использующий getURL, должен проверять не возвращена ли ошибка, и обрабатывать её, если она есть.

```
getURL("data/nonsense.txt", function(content, error) {
  if (error != null)
    console.log("Failed to fetch nonsense.txt: " + error
);
  else
    console.log("nonsense.txt: " + content);
});
```

С исключениями это не помогает. Когда мы совершаем последовательно несколько асинхронных действий, исключение в любой точке цепочки в любом случае (если только вы не обернули каждый обработчик в свой блок try/catch) вывалится на верхнем уровне и прервёт всю цепочку.

Обещания

Тяжело писать асинхронный код для сложных проектов в виде простых обратных вызовов. Очень легко забыть проверку на ошибку или позволить неожиданному исключению резко прервать программу. Кроме того, организация правильной обработки ошибок и проход ошибки через несколько последовательных обратных вызовов очень утомительна.

Предпринималось множество попыток решить эту проблему дополнительными абстракциями. Одна из наиболее удачных попыток называется обещаниями (promises). Обещания оборачивают асинхронное действие в объект, который может передаваться и которому нужно сделать какие-то вещи, когда действие завершается или не удаётся. Такой интерфейс уже стал частью текущей версии JavaScript, а для старых версий его можно использовать в виде библиотеки.

Интерфейс обещаний не особенно интуитивно понятный, но мощный. В этой главе мы лишь частично опишем его. Больше информации можно найти на www.promisejs.org

Для создания объекта promises мы вызываем конструктор Promise, задавая ему функцию инициализации асинхронного действия. Конструктор вызывает эту функцию и передаёт ей два аргумента, которые сами также являются функциями. Первая должна вызываться в удачном случае, другая – в неудачном.

И вот наша обёртка для запросов GET, которая на этот раз возвращает обещание. Теперь мы просто назовём его get.

```
function get(url) {
  return new Promise(function(succeed, fail) {
    var req = new XMLHttpRequest();
    req.open("GET", url, true);
    req.addEventListener("load", function() {
      if (req.status < 400)
        succeed(req.responseText);
      else
        fail(new Error("Request failed: " + req.statusTe
xt));
    });
    req.addEventListener("error", function() {
      fail(new Error("Network error"));
    });
    req.send(null);
  });
}
```

Заметьте, что интерфейс к самой функции упростился. Мы передаём ей URL, а она возвращает обещание. Оно работает как обработчик для выходных данных запроса. У него есть метод then, который вызывается с двумя функциями: одной для обработки успеха, другой – для неудачи.

```
get("example/data.txt").then(function(text) {
  console.log("data.txt: " + text);
}, function(error) {
  console.log("Failed to fetch data.txt: " + error);
});
```

Пока это всё ещё один из способов выразить то же, что мы уже сделали. Только когда у вас появляется цепь событий, становится видна заметная разница.

Вызов then производит новое обещание, чей результат (значение, передающееся в обработчики успешных результатов) зависит от значения, возвращаемого первой переданной нами в then функцией. Эта функция может вернуть ещё одно обещание, обозначая что проводится дополнительная асинхронная работа. В этом случае обещание, возвращаемое then само по себе будет ждать обещания, возвращённого функцией-обработчиком, и успех или неудача произойдут с таким же значением. Когда функция-обработчик возвращает значение, не являющееся обещанием, обещание, возвращаемое then, становится успешным, в качестве результата используя это значение.

Значит, вы можете использовать then для изменения результата обещания. К примеру, следующая функция возвращает обещание, чей результат – содержимое с данного URL, разобранное как JSON:

```
function getJSON(url) {
  return get(url).then(JSON.parse);
}
```

Последний вызов then не обозначил обработчик неудач. Это допустимо. Ошибка будет передана в обещание, возвращаемое через then, а ведь это нам и надо – getJSON не знает, что делать, когда что-то идёт не так, но есть надежда, что вызывающий её код это знает.

В качестве примера, показывающего использование обещаний, мы напишем программу, получающую число JSON-файлов с сервера, и показывающую во время исполнения запроса слово «загрузка». Файлы содержат информацию о людях и ссылки на другие файлы с информацией о других людях в свойствах типа отец, мать, супруг.

Нам нужно получить имя матери супруга из example/bert.json. В случае проблем нам нужно убрать текст «загрузка» и показать сообщение об ошибке. Вот как это можно делать при помощи обещаний:

```
<script>
  function showMessage(msg) {
    var elt = document.createElement("div");
    elt.textContent = msg;
    return document.body.appendChild(elt);
  }

  var loading = showMessage("Загрузка...");
  getJSON("example/bert.json").then(function(bert) {
    return getJSON(bert.spouse);
  }).then(function(spouse) {
    return getJSON(spouse.mother);
  }).then(function(mother) {
    showMessage("Имя - " + mother.name);
  }).catch(function(error) {
    showMessage(String(error));
  }).then(function() {
    document.body.removeChild(loading);
  });
</script>
```

Итоговая программа относительно компактна и читаема. Метод catch схож с then, но он ожидает только обработчик неудачного результата и в случае успеха передаёт дальше неизменённый результат. Исполнение программы будет продолжено обычным путём после отлова исключения – так же, как в случае с try/catch. Таким образом, последний then, удаляющий сообщение о загрузке, выполняется в любом случае, даже в случае неудачи.

Можно представлять себе, что интерфейс обещаний – это отдельный язык для асинхронной обработки исполнения программы. Дополнительные вызовы методов и функций, которые нужны для его работы, придают коду несколько странный вид, но не настолько неудобный, как обработка всех ошибок вручную.

Цените HTTP

При создании системы, в которой программа на JavaScript в браузере (клиентская) общается с серверной программой, можно использовать несколько вариантов моделирования такого общения.

Общепринятый метод – удалённые вызовы процедур. В этой модели общение идёт по шаблону обычных вызовов функций, только функции эти выполняются на другом компьютере. Вызов заключается в создании запроса на сервер, в который входят имя функции и аргументы. Ответ на запрос включает возвращаемое значение.

При использовании удалённых вызовов процедур HTTP служит лишь транспортом для общения, и вы, скорее всего, напишете слой абстракции, который спрячет его полностью.

Другой подход – построить свою систему общения на концепции ресурсов и методов HTTP. Вместо удалённого вызова процедуры по имени addUser вы делаете запрос PUT к /users/larry. Вместо кодирования свойств пользователя в аргументах функции вы определяете формат документа или используете существующий формат, который будет представлять пользователя. Тело PUT-запроса, создающего новый ресурс, будет просто документом этого формата. Ресурс получается через запрос GET к его URL (/user/larry), который возвращает представляющий этот ресурс документ.

Второй подход упрощает использование некоторых возможностей HTTP, например поддержки кеширования ресурсов (копия ресурса хранится на стороне клиента). Также он способствует созданию согласованного интерфейса, потому что думать в терминах ресурсов проще, чем в терминах функций.

Безопасность и HTTPS

Данные путешествуют по интернету по длинному и опасному пути. Чтобы добраться до пункта назначения, им надо попрыгать через всякие места, начиная от Wi-Fi

сети кофейни до сетей, контролируемых разными организациями и государствами. В любой точке пути их могут прочитать или даже поменять.

Если нужно хранить что-либо в секрете, например пароли к емейлу, или данным необходимо прийти в пункт назначения в неизменном виде — таким, например, как номер банковского счёта, на который вы переводите деньги,- простого HTTP недостаточно.

Безопасный протокол HTTP, URL которого начинаются с https://, оборачивает HTTP-трафик так, чтобы его было сложнее прочитать и поменять. Сначала клиент подтверждает, что сервер – тот, за кого себя выдаёт, требуя с сервера представить криптографический сертификат, выданный авторитетной стороной, которую признаёт браузер. Потом, все данные, проходящие через соединение, шифруются так, чтобы предотвратить прослушку и изменение.

Таким образом, когда всё работает правильно, HTTPS предотвращает как случаи, когда кто-то притворяется другим веб-сайтом, с которым вы общаетесь, так и случаи прослушки вашего общения. Он не идеален, и уже были случаи, когда HTTPS не справлялся с работой из-за поддельных или краденых сертификатов или сломанных программ. Тем не менее, с HTTP очень легко сделать что-то плохое, а взлом HTTPS требует таких усилий, которые

могут прикладывать только государственные структуры или очень серьёзные криминальные организации (а между этими организациями иногда совсем нет различий).

Итог

В этой главе мы увидели, что HTTP – это протокол доступа к ресурсам в интернете. Клиент отправляет запрос, содержащий метод (обычно GET), и путь, который определяет ресурс. Сервер решает, что ему делать с запросом и отвечает с кодом статуса и телом ответа. Запросы и ответы могут содержать заголовки, в которых передаётся дополнительная информация.

Браузеры делают GET-запросы для получения ресурсов, необходимых для показа страницы. Страница может содержать формы, которые позволяют информации, введённой пользователем, быть отправленной в запросе, который создаётся после отправки формы. Вы узнаете об этом больше в следующей главе.

Интерфейс, через который JavaScript делает HTTP-запросы из браузера, называется XMLHttpRequest. Можно игнорировать приставку “XML” (но писать её всё равно нужно). Использовать его можно двумя способами: синхронным, который блокирует всю работу до окончания

выполнения запроса, и асинхронным, который требует установки обработчика событий, отслеживающего окончание запроса. Почти во всех случаях предпочтительным является асинхронный способ. Создание запроса выглядит так:

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
  console.log(req.statusCode);
});
req.send(null);
```

Асинхронное программирование – непростая вещь. Обещания – интерфейс, который делает её проще, помогая направлять сообщения об ошибках и исключения к нужному обработчику, и абстрагируя некоторые повторяющиеся элементы, подверженные ошибкам.

Упражнения

Согласование содержания (content negotiation)

Одна из вещей, которые HTTP умеет делать, но которую мы не обсуждали, называется согласованием содержания. Заголовок Accept в запросе можно

использовать для сообщения серверу того, какие типы документов клиент желает получить. Многие серверы его игнорируют, но когда сервер знает о разных способах кодирования ресурса, он может взглянуть на заголовок и отправить тот, который предпочитает клиент.

URL eloquentjavascript.net/author настроен на ответ как прямым текстом, так и HTML или JSON, в зависимости от запроса клиента. Эти форматы определяются стандартизированными типами содержимого text/plain, text/html, и application/json.

Отправьте запрос для получения всех трёх форматов этого ресурса. Используйте метод setRequestHeader объекта XMLHttpRequest для установки заголовка Accept в один из нужных типов содержимого. Убедитесь, что вы устанавливаете заголовок после open, но перед send.

Наконец, попробуйте запросить содержимое типа application/rainbows+unicorns и посмотрите, что произойдёт.

Ожидание нескольких обещаний

У конструктора Promise есть метод all, который, получая массив обещаний, возвращает обещание, которое ждёт завершения всех указанных в массиве обещаний. Затем он выдаёт успешный результат и возвращает массив с

результатами. Если какие-то из обещаний в массиве завершились неудачно, общее обещание также возвращает неудачу (со значением неудавшегося обещания из массива).

Попробуйте сделать что-либо подобное, написав функцию all.

Заметьте, что после завершения обещания (когда оно либо завершилось успешно, либо с ошибкой), оно не может заново выдать ошибку или успех, и дальнейшие вызовы функции игнорируются. Это может упростить обработку ошибок в вашем обещании.

```
function all(promises) {
  return new Promise(function(success, fail) {
    // Ваш код.
  });
}

// Проверочный код.
all([]).then(function(array) {
  console.log("Это должен быть []:", array);
});
function soon(val) {
  return new Promise(function(success) {
    setTimeout(function() { success(val); },
               Math.random() * 500);
  });
}
all([soon(1), soon(2), soon(3)]).then(function(array) {
  console.log("Это должен быть [1, 2, 3]:", array);
});
function fail() {
  return new Promise(function(success, fail) {
    fail(new Error("бабах"));
  });
}
all([soon(1), fail(), soon(3)]).then(function(array) {
  console.log("Сюда мы попасть не должны ");
}, function(error) {
  if (error.message != "бабах")
    console.log("Неожиданный облом:", error);
});
```

Формы и поля форм

Я нынче ж на ученом кутеже
Твое доверье службой завоюю,
Ты ж мне черкни расписку долговую,
Чтоб мне не сомневаться в платеже.

Мефистофель, в «Фаусте» Гёте

Формы были кратко представлены в предыдущей главе в качестве способа передачи информации, введённой пользователем, через HTTP. Они были разработаны в вебе до появления JavaScript, с тем расчётом, что взаимодействие с сервером происходит при переходе на другую страницу.

Но их элементы являются частями DOM, как и остальные части страницы, а элементы DOM, представляющие поля формы, поддерживают несколько свойств и событий, которых нет у других элементов. Это делает возможным просматривать и управлять полями ввода из программ JavaScript и добавлять функциональности к классическим формам или использовать формы и поля как основу для построения приложения.

Поля

Веб-форма состоит из любого числа полей ввода, окружённых тегом `<form>`. HTML предлагает много разных полей, от простых галочек со значениями вкл/выкл до выпадающих списков и полей для ввода текста. В этой книге не будут подробно обсуждаться все виды полей, но мы сделаем небольшой их обзор.

Много типов полей ввода используют тег `<input>`. Его атрибут type используется для выбора стиля поля. Вот несколько распространённых типов:

text текстовое поле на одну строку **password** то же, что текст, но прячет ввод **checkbox** переключатель вкл/выкл **radio** часть поля с возможностью выбора из нескольких вариантов **file** позволяет пользователю выбрать файл на его компьютере

Поля форм не обязательно должны появляться внутри тега `<form>`. Их можно разместить в любом месте страницы. Информацию из таких полей нельзя передавать на сервер (это возможно только для всей формы целиком), но когда мы делаем поля, которые обрабатывает JavaScript, нам обычно и не нужно передавать информацию из полей через submit.

```
<p><input type="text" value="abc"> (text)</p>
<p><input type="password" value="abc"> (password)</p>
<p><input type="checkbox" checked> (checkbox)</p>
<p><input type="radio" value="A" name="choice">
   <input type="radio" value="B" name="choice" checked>
   <input type="radio" value="C" name="choice"> (radio)</
p>
<p><input type="file" checked> (file)</p>
```

Интерфейс JavaScript для таких элементов разнится в зависимости от типа. Мы рассмотрим каждый из них чуть позже.

У текстовых полей на несколько строк есть свой тег `<textarea>`. У тега должен быть закрывающий тег `</textarea>`, и он использует текст внутри этих тегов вместо использования атрибута value.

```
<textarea>
один
два
три
</textarea>
```

А тег `<select>` используется для создания поля, которое позволяет пользователю выбрать один из заданных вариантов.

```
<select>
  <option>Блины</option>
  <option>Запеканка</option>
  <option>Мороженка </option>
</select>
```

Когда значение поля изменяется, запускается событие “change”.

Фокус

В отличие от большинства элементов документа HTML, поля форм могут получать фокус ввода клавиатуры. При щелчке или выборе их другим способом они становятся активными, т.е. главными приёмниками клавиатурного ввода.

Если в документе есть текстовое поле, то набираемый текст появится в нём, только если поле имеет фокус ввода. Другие поля по-разному реагируют на клавиатуру. К примеру, `<select>` пытается перейти на вариант, содержащий текст, который вводит пользователь, а также отвечает на нажатия стрелок, передвигая выбор варианта вверх и вниз.

Управлять фокусом из JavaScript можно методами focus и blur. Первый перемещает фокус на элемент DOM, из которого он вызван, а второй убирает фокус. Значение document.activeElement соответствует текущему элементу, получившему фокус.

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
  // → BODY
</script>
```

На некоторых страницах нужно, чтобы пользователь сразу начинал работу с какого-то из полей формы. При помощи JavaScript можно передать этому полю фокус при загрузке документа, но в HTML также есть атрибут autofocus, который приводит к тому же результату, но сообщает браузеру о наших намерениях. В этом случае браузер может отменить это поведение в подходящих случаях, например когда пользователь перевёл фокус куда-то ещё.

```
<input type="text" autofocus>
```

Браузеры по традиции позволяют пользователю перемещать фокус клавишей Tab. Мы можем влиять на порядок перемещения через атрибут tabindex. В примере документ будет переносить фокус с текстового поля на кнопку OK, вместо того, чтобы сначала пройти через ссылку help.

```
<input type="text" tabindex=1> <a href=".">(help)</a>
<button onclick="console.log('ok')" tabindex=2>OK</button
>
```

По умолчанию, большинство типов элементов HTML не получают фокус. Но добавив tabindex к элементу, вы сделаете возможным получение им фокуса.

Отключённые поля

Все поля можно отключить атрибутом disabled, который существует и в виде свойства элемента объекта DOM.

```
<button>У меня всё хорошо</button>
<button disabled>Я в отключке</button>
```

Отключённые поля не принимают фокус и не изменяются, и в отличие от активных, обычно выглядят серыми и выцветшими.

Когда программа находится в процессе обработки нажатия на кнопку или другой элемент, которое может потребовать общение с сервером и занять длительное время, неплохо отключать элемент до завершения действия. В этом случае, когда пользователь потеряет терпение и нажмёт на элемент ещё раз, действие не будет повторено лишний раз.

Форма в целом

Когда поле, содержится в элементе `<form>`, у его элемента DOM будет свойство form, которое будет ссылаться на форму. Элемент `<form>`, в свою очередь, имеет свойство elements, содержащее массивоподобную коллекцию полей.

Атрибут name поля задаёт, как будет определено значение этого поля при передаче на сервер. Его также можно использовать как имя свойства при доступе к свойству формы elements, который работает и как объект, похожий на массив (с доступом по номерам), так и map (с доступом по имени).

```
<form action="example/submit.html">
  Имя: <input type="text" name="name"><br>
  Пароль: <input type="password" name="password"><br>
  <button type="submit">Войти</button>
</form>
<script>
  var form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

Кнопка с атрибутом type равным submit при нажатии отправляет форму. Нажатие клавиши Enter внутри поля формы имеет тот же эффект.

Отправка формы обычно означает, что браузер переходит на страницу, обозначенную в атрибуте формы action, используя либо GET либо POST запрос. Но перед этим запускается свойство “submit”. Его можно обработать в JavaScript, и обработчик может предотвратить поведение по умолчанию, вызвав на объекте event preventDefault.

```
<form action="example/submit.html">
  Значение: <input type="text" name="value">
  <button type="submit">Сохранить </button>
</form>
<script>
  var form = document.querySelector("form");
  form.addEventListener("submit", function(event) {
    console.log("Saving value", form.elements.value.valu
e);
    event.preventDefault();
  });
</script>
```

Перехват событий “submit” полезен в нескольких случаях. Мы можем написать код, проверяющий допустимость введённых значений и сразу же показать ошибку вместо передачи данных формы. Или мы можем отключить отправку формы по умолчанию и дать программе возможность самой обработать ввод, например используя XMLHttpRequest для отправки данных на сервер без перезагрузки страницы.

Текстовые поля

Поля с тегами `<input>` и типами text и password, а также теги , имеют общий интерфейс. У их элементов DOM есть свойство value, в котором содержится их текущее

содержимое в виде строки текста. Присваивание этому свойству значения меняет содержимое поля.

Свойства текстовых полей selectionStart и selectionEnd содержат данные о положении курсора и выделения текста. Когда ничего не выделено, их значение одинаковое, и равно положению курсора. Например, 0 обозначает начало текста, 10 обозначает, что курсор находится на 10-м символе. Когда выделена часть поля, свойства имеют разные значения, а именно начало и конец выделенного текста. В эти поля также можно записывать значение.

К примеру, представьте, что вы пишете статью про Khasekhemwy, но затрудняетесь писать его имя правильно. Следующий код назначает тегу `<textarea>` обработчик событий, который при нажатии F2 вставляет строку “ Khasekhemwy”.

```
<textarea></textarea>
<script>
  var textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", function(event) {
    // The key code for F2 happens to be 113
    if (event.keyCode == 113) {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    var from = field.selectionStart, to = field.selectio
nEnd;
    field.value = field.value.slice(0, from) + word +
                  field.value.slice(to);
    // Put the cursor after the word
    field.selectionStart = field.selectionEnd =
      from + word.length;
  };
</script>
```

Функция replaceSelection заменяет текущий выделенный текст заданным словом, и перемещает курсор на позицию после этого слова, чтобы можно было продолжать печатать.

Событие “change” для текстового поля не срабатывает каждый раз при вводе одного символа. Оно срабатывает после потери полем фокуса, когда его значение было изменено. Чтобы мгновенно реагировать на изменение текстового поля нужно зарегистрировать событие “input”,

которое срабатывает каждый раз при вводе символа, удалении текста или других манипуляциях с содержимым поля.

В следующем примере мы видим текстовое поле и счётчик, показывающий текущую длину введённого текста.

```
<input type="text"> length: <span id="length">0</span>
<script>
  var text = document.querySelector("input");
  var output = document.querySelector("#length");
  text.addEventListener("input", function() {
    output.textContent = text.value.length;
  });
</script>
```

Галочки и радиокнопки

Поле галочки – простой бинарный переключатель. Его значение можно извлечь или поменять через свойство checked, содержащее булевскую величину.

```
<input type="checkbox" id="purple">
<label for="purple">Сделать страницу фиолетовой</label>
<script>
  var checkbox = document.querySelector("#purple");
  checkbox.addEventListener("change", function() {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

Тег `<label>` используется для связи куска текста с полем ввода. Атрибут for должен совпадать с id поля. Щелчок по метке label включает поле ввода, оно получает фокус и меняет значение – если это галочка или радиокнопка.

Радиокнопка схожа с галочкой, но она связана с другими радиокнопками с тем же именем, так что только одна из них может быть выбрана.

```
Цвет:
<input type="radio" name="color" value="mediumpurple"> Ф
иолетовый
<input type="radio" name="color" value="lightgreen"> Зел
ёныйы
<input type="radio" name="color" value="lightblue"> Голу
бой
<script>
  var buttons = document.getElementsByName("color");
  function setColor(event) {
    document.body.style.background = event.target.value;
  }
  for (var i = 0; i < buttons.length; i++)
    buttons[i].addEventListener("change", setColor);
</script>
```

Метод document.getElementsByName выдаёт все элементы с заданным атрибутом name. Пример перебирает их (посредством обычного цикла for, а не forEach, потому что возвращаемая коллекция – не настоящий массив) и регистрирует обработчик событий для каждого элемента. Помните, что у объектов событий есть свойство target, относящееся к элементу, который запустил событие. Это полезно для создания обработчиков событий – наш обработчик может быть вызван разными элементами, и у него должен быть способ получить доступ к текущему элементу, который его вызвал.

Поля select

Поля select похожи на радиокнопки – они также позволяют выбрать из нескольких вариантов. Но если радиокнопки позволяют нам контролировать раскладку вариантов, то вид поля `<select>` определяет браузер.

У полей select есть вариант, больше похожий на список галочек, чем на радиокнопки. При наличии атрибута multiple тег `<select>` позволит выбирать любое количество вариантов, а не один.

```
<select multiple>
  <option>Блины</option>
  <option>Запеканка</option>
  <option>Мороженка </option>
</select>
```

В большинстве браузеров внешний вид поля будет отличаться от поля с единственным вариантом выбора, который обычно выглядит как выпадающее меню.

Атрибут size тега `<select>` используется для задания количества вариантов, которые видны одновременно – так вы можете влиять на внешний вид выпадушки. К примеру, назначив size 3, вы увидите три строки одновременно, безотносительно того, присутствует ли опция multiple.

У каждого тега `<option>` есть значение. Его можно определить атрибутом value, но если он не задан, то значение тега определяет текст, находящийся внутри тега `<option>..</option>`. Свойство value элемента отражает текущий выбранный вариант. Для поля с возможностью выбора нескольких вариантов это свойство не особо нужно, т.к. в нём будет содержаться только один из нескольких выбранных вариантов.

К тегу `<option>` поля `<select>` можно получить доступ как к массивоподобному объекту через свойство options. У каждого варианта есть свойство selected, показывающее, выбран ли сейчас этот вариант. Свойство также можно менять, чтобы вариант становился выбранным или не выбранным.

Следующий пример извлекает выбранные значения из поля select и использует их для создания двоичного числа из битов. Нажмите Ctrl (или Command на Маке), чтобы выбрать несколько значений сразу.

```
<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  var select = document.querySelector("select");
  var output = document.querySelector("#output");
  select.addEventListener("change", function() {
    var number = 0;
    for (var i = 0; i < select.options.length; i++) {
      var option = select.options[i];
      if (option.selected)
        number += Number(option.value);
    }
    output.textContent = number;
  });
</script>
```

Файловое поле

Файловое поле изначально было предназначено для закачивания файлов с компьютера через форму. В современных браузерах они также позволяют читать файлы из JavaScript. Поле работает как охранник для файлов. Скрипт не может просто взять и открыть файл с

компьютера пользователя, но если тот выбрал файл в этом поле, браузер разрешает скрипту начать чтение файла.

Файловое поле обычно выглядит как кнопка с надписью вроде «Выберите файл», с информацией про выбранный файл рядом с ней.

```
<input type="file">
<script>
  var input = document.querySelector("input");
  input.addEventListener("change", function() {
    if (input.files.length > 0) {
      var file = input.files[0];
      console.log("You chose", file.name);
      if (file.type)
        console.log("It has type", file.type);
    }
  });
</script>
```

Свойство files элемента – массивоподобный объект (не настоящий массив), содержащий список выбранных файлов. Изначально он пуст. У элемента нет простого свойства file, потому что пользователь может выбрать несколько файлов за раз при включённом атрибуте multiple.

У объектов в свойстве files есть свойства имя (имя файла), размер (размер файла в байтах), и тип (тип файла в смысле media type — text/plain или image/jpeg).

Чего у него нет, так это свойства, содержащего содержимое файла. Чтобы получить содержимое, приходиться постараться. Так как чтение файла с диска занимает длительное время, интерфейс должен быть асинхронным, чтобы документ не замирал. Конструктор FileReader можно представлять себе, как конструктор XMLHttpRequest, только для файлов.

```
<input type="file" multiple>
<script>
  var input = document.querySelector("input");
  input.addEventListener("change", function() {
    Array.prototype.forEach.call(input.files, function(f
ile) {
      var reader = new FileReader();
      reader.addEventListener("load", function() {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    });
  });
</script>
```

Чтение файла происходит при помощи создания объекта FileReader, регистрации события “load” для него, и вызова его метода readAsText с передачей тому файла. По окончанию загрузки в свойстве result сохраняется содержимое файла.

Пример использует Array.prototype.forEach для прохода по массиву, так как в обычном цикле было бы неудобно получать нужные объекты file и reader от обработчика событий. Переменные были бы общими для всех итераций цикла.

У FileReaders также есть событие “error”, когда чтение файла не получается. Объект error будет сохранён в свойстве error. Если вы не хотите забивать голову ещё одной неудобной асинхронной схемой, вы можете обернуть её в обещание (см. главу 17):

```
function readFile(file) {
  return new Promise(function(succeed, fail) {
    var reader = new FileReader();
    reader.addEventListener("load", function() {
      succeed(reader.result);
    });
    reader.addEventListener("error", function() {
      fail(reader.error);
    });
    reader.readAsText(file);
  });
}
```

Возможно читать только часть файла, вызывая slice и передавая результат (т.н. объект blob) объекту reader.

Хранение данных на стороне клиента

Простые HTML-странички с добавкой JavaScript могут выступать отличной основой для мини-приложений – небольших вспомогательных программ, автоматизирующих ежедневные дела. Присоединив к полям формы обработчики событий вы можете делать всё – от конвертации фаренгейтов в цельсии до генерации паролей из основного пароля и имени веб-сайта.

Когда такому приложению нужно сохранять информацию между сессиями, переменные JavaScript использовать не получится – их значения выбрасываются каждый раз при закрытии страницы. Можно было бы настроить сервер, подсоединить его к интернету и тогда приложение хранило бы ваши данные там. Это мы разберём в главе 20. Но это добавляет вам работы и сложности. Иногда достаточно хранить данные в своём браузере. Но как?

Можно хранить строковые данные так, что они переживут перезагрузку страниц — для этого надо положить их в объект localStorage. Он разрешает хранить строковые данные под именами (которые тоже являются строками), как в этом примере:

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

Переменная в localStorage хранится, пока её не перезапишут, удаляется при помощи removeItem или очисткой локального хранилища пользователем.

У сайтов с разных доменов – разные отделения в этом хранилище. То есть, данные, сохранённые с вебсайта в localStorage, могут быть прочтены или перезаписаны только скриптами с этого же сайта.

Также браузеры ограничивают объём хранимых данных, обычно в несколько мегабайт. Это ограничение, вкупе с тем фактом, что забивание жёстких дисков у людей не приносит прибыли, предотвращает отъедание места на диске.

Следующий код реализует простую программу для ведения заметок. Она хранит заметки в виде объекта, ассоциируя заголовки с содержимым. Он кодируется в JSON и хранится в localStorage. Пользователь может выбрать записку через поле `<select>` и поменять её текст в `<textarea>`. Добавляется запись по нажатию на кнопку.

```
Заметки: <select id="list"></select>
<button onclick="addNote()">новая</button><br>
<textarea id="currentnote" style="width: 100%; height: 1
0em">
</textarea>

<script>
  var list = document.querySelector("#list");
  function addToList(name) {
    var option = document.createElement("option");
    option.textContent = name;
    list.appendChild(option);
  }

  // Берём список из локального хранилища
  var notes = JSON.parse(localStorage.getItem("notes"))
||
              {"что купить": ""};
  for (var name in notes)
    if (notes.hasOwnProperty(name))
      addToList(name);

  function saveToStorage() {
    localStorage.setItem("notes", JSON.stringify(notes))
;
```

```
  }

  var current = document.querySelector("#currentnote");
  current.value = notes[list.value];

  list.addEventListener("change", function() {
    current.value = notes[list.value];
  });
  current.addEventListener("change", function() {
    notes[list.value] = current.value;
    saveToStorage();
  });

  function addNote() {
    var name = prompt("Имя записи", "");
    if (!name) return;
    if (!notes.hasOwnProperty(name)) {
      notes[name] = "";
      addToList(name);
      saveToStorage();
    }
    list.value = name;
    current.value = notes[name];
  }
</script>
```

Скрипт инициализирует переменную notes значением из localStorage, а если его там нет – простым объектом с одной записью «что купить». Попытка прочесть отсутствующее поле из localStorage вернёт null. Передав null в JSON.parse, мы получим null обратно. Поэтому для значения по умолчанию можно использовать оператор ||.

Когда данные в note меняются (добавляется новая запись или меняется текущая), для обновления хранимого поля вызывается функция saveToStorage. Если б мы рассчитывали, что у нас будут храниться тысячи записей, это было бы слишком накладно, и нам пришлось бы придумать более сложную процедуру для хранения – например, своё поле для каждой записи.

Когда пользователь добавляет запись, код должен обновить текстовое поле, хотя у поля и есть обработчик “change”. Это нужно потому, что событие “change” происходит, только когда пользователь меняет значение поля, а не когда это делает скрипт.

Есть ещё один похожий на localStorage объект под названием sessionStorage. Разница между ними в том, что содержимое sessionStorage забывается по окончанию сессии, что для большинства браузеров означает момент закрытия.

Итог

HTML предоставляет множество различных типов полей формы – текстовые, галочки, множественного выбора, выбора файла.

Из JavaScript можно получать значение и манипулировать этими полями. По изменению они запускают событие “change”, по вводу с клавиатуры – “input”, и ещё много разных клавиатурных событий. Они помогают нам отловить момент, когда пользователь взаимодействует с полем ввода. Свойства вроде value (для текстовых полей и select) или checked (для галочек и радиокнопок) используются для чтения и записи содержимого полей.

При передаче формы происходит событие “submit”. Обработчик JavaScript затем может вызвать preventDefault этого события, чтобы остановить передачу данных. Элементы формы не обязаны быть заключены в теги `<form>`.

Когда пользователь выбрал файл с жёсткого диска через поле выбора файла, интерфейс FileReader позволит нам добраться до содержимого файла из программы JavaScript.

Объекты localStorage и sessionStorage можно использовать для хранения информации таким способом, который переживёт перезагрузку страницы. Первый сохраняет данные навсегда (ну или пока пользователь специально не сотрёт их), а второй – до закрытия браузера.

Упражнения

Верстак JavaScript

Сделайте интерфейс, позволяющий писать и исполнять кусочки кода JavaScript.

Сделайте кнопку рядом с `<textarea>`, по нажатию которой конструктор Function из главы 10 будет обёртывать введённый текст в функцию и вызывать его. Преобразуйте значение, возвращаемое функцией, или любую её ошибку, в строку, и выведите её после текстового поля.

```
<textarea id="code">return "hi";</textarea>
<button id="button">Поехали</button>
<pre id="output"></pre>

<script>
  // Ваш код.
</script>
```

Автодополнение

Дополните текстовое поле так, что при вводе текста под ним появлялся бы список вариантов. У вас есть массив возможных вариантов, и показывать нужно те из них,

которые начинаются с вводимого текста. Когда пользователь щёлкает по предложенному варианту, он меняет содержимое поля на него.

```
<input type="text" id="field">
<div id="suggestions" style="cursor: pointer"></div>

<script>
  // Строит массив из имён глобальных перменных,
  // типа  'alert', 'document', и 'scrollTo'
  var terms = [];
  for (var name in window)
    terms.push(name);

  // Ваш код.
</script>
```

Игра «Жизнь» Конвея

Это простая симуляция жизни на прямоугольной решётке, каждый элемент которой живой или нет. Каждое поколение (шаг игры) применяются следующие правила:

— каждая живая клетка, количество соседей которой меньше двух или больше трёх, погибает — каждая живая клетка, у которой от двух до трёх соседей, живёт до следующего хода — каждая мёртвая клетка, у которой есть ровно три соседа, оживает

Соседи клетки – это все соседние с ней клетки по горизонтали, вертикали и диагонали, всего 8 штук.

Обратите внимание, что правила применяются ко всей решётке одновременно, а не к каждой из клеток по очереди. То есть, подсчёт количества соседей происходит в один момент перед следующим шагом, и изменения, происходящие на соседних клетках, не влияют на новое состояние клетки.

Реализуйте игру, используя любые подходящие структуры. Используйте Math.random для создания случайных начальных популяций. Выводите поле как решётку из галочек с кнопкой «перейти на следующий шаг». Когда пользователь включает или выключает галочки, эти изменения нужно учитывать при подсчёте следующего поколения.

```
<div id="grid"></div>
<button id="next">Следующее поколение</button>

<script>
  // Ваш код.
</script>
```

Проект: Paint

Я смотрю на многообразие цветов. Я смотрю на пустой холст. Затем я пытаюсь нанести цвета как слова, из которых возникают поэмы, как ноты, из которых возникает музыка.

Жоан Миро

Материал предыдущих глав даёт вам всё необходимое для создания простого веб-приложения. Именно этим мы и займёмся.

Наше приложение будет программой для рисования в браузере, схожей с Microsoft Paint. С его помощью можно будет открывать файлы с изображениями, малевать на них мышкой и сохранять обратно. Вот, как это будет выглядеть:

Простая программа рисования

Рисовать на компьютере клёво. Не надо волноваться насчёт материалов, умения, таланта. Просто берёшь, и начинаешь калякать.

Реализация

Интерфейс программы выводит вверху большой элемент `<canvas>`, под которым есть несколько полей ввода. Пользователь рисует на картинке, выбирая инструмент из

поля `<select>`, а затем нажимая на холсте мышь. Есть инструменты для рисования линий, стирания кусочков картинки, добавления текста и т.п.

Щелчок на холсте передаёт событие «mousedown» текущему инструменту, который обрабатывает его, как считает нужным. Рисование линий, например, будет слушать события «mousemove», пока кнопка мыши не будет отпущена, и нарисует линию по пути мыши текущим цветом и размером кисти.

Цвет и размер кисти выбираются в дополнительных полях ввода. Они выполняют обновление свойств контекста рисования на холсте fillStyle, strokeStyle, и lineWidth каждый раз при их изменении.

Загрузить картинку в программу можно двумя способами. Первый использует поле file, где пользователь выбирает файл со своего диска. Вторая запрашивает URL и скачивает картинку из интернета.

Картинки хранятся нестандартным способом. Ссылка save с правой стороны ведёт на текущую картинку. По ней можно проходить, делиться ей или сохранять файл через неё. Я скоро объясню, как это работает.

Строим DOM

Интерфейс программы состоит из более чем 30 элементов DOM. Нужно их как-то собрать вместе.

Очевидным форматом для сложных структур DOM является HTML. Но разделять программу на HTML и скрипт неудобно – для элементов DOM понадобится множество обработчиков событий или других необходимых вещей, которые надо будет как-то обрабатывать из скрипта. Для этого придётся делать много вызовов querySelector и им подобных, чтобы найти нужный элемент DOM для работы.

Было бы удобно определять части DOM рядом с теми частями кода JavaScript, которые ими управляют. Поэтому я решил создавать всю конструкцию DOM прямо в JavaScript. Как мы видели в главе 13, встроенный интерфейс для создания структур DOM ужасно многословен. Поскольку нам придётся создать много конструкций, нам понадобится вспомогательная функция.

Эта функция – расширенная версия функции elt из главы 13. Она создаёт элемент с заданным именем и атрибутами, и добавляет все остальные аргументы, которые получает, в качестве дочерних узлов, автоматически преобразовывая строки в текстовые узлы.

```
function elt(name, attributes) {
  var node = document.createElement(name);
  if (attributes) {
    for (var attr in attributes)
      if (attributes.hasOwnProperty(attr))
        node.setAttribute(attr, attributes[attr]);
  }
  for (var i = 2; i < arguments.length; i++) {
    var child = arguments[i];
    if (typeof child == "string")
      child = document.createTextNode(child);
    node.appendChild(child);
  }
  return node;
}
```

Так мы легко и просто создаём элементы, не раздувая код до размеров лицензионного соглашения.

Основание

Ядро нашей программы – функция createPaint, добавляющая интерфейс рисования к элементу DOM, который передаётся в качестве аргумента. Так как мы создаём программу последовательно, мы определяем объект controls, который будет содержать функции для инициализации разных элементов управления под картинкой.

```
var controls = Object.create(null);

function createPaint(parent) {
  var canvas = elt("canvas", {width: 500, height: 300});
  var cx = canvas.getContext("2d");
  var toolbar = elt("div", {class: "toolbar"});
  for (var name in controls)
    toolbar.appendChild(controls[name](cx));

  var panel = elt("div", {class: "picturepanel"}, canvas
);
  parent.appendChild(elt("div", null, panel, toolbar));
}
```

У каждого элемента управления есть доступ к контексту рисования на холсте, а через него – к элементу `<canvas>`. Основное состояние программы хранится в этом холсте – он содержит текущую картинку, выбранный цвет (в свойстве fillStyle) и размер кисти (в свойстве lineWidth).

Мы обернём холст и элементы управления в элементы `<div>` с классами, чтобы можно было добавить им стили, например серую рамку вокруг картинки.

Выбор инструмента

Первый элемент управления, который мы добавим – элемент `<select>`, позволяющий выбирать инструмент рисования. Как и в случае с controls, мы будем

использовать объект для сбора необходимых инструментов, чтобы не надо было описывать их работу в коде по отдельности, и чтобы можно было легко добавлять новые. Этот объект связывает названия инструментов с функцией, которая вызывается при их выборе и при клике на холсте.

```
var tools = Object.create(null);

controls.tool = function(cx) {
  var select = elt("select");
  for (var name in tools)
    select.appendChild(elt("option", null, name));

  cx.canvas.addEventListener("mousedown", function(event
) {
    if (event.which == 1) {
      tools[select.value](event, cx);
      event.preventDefault();
    }
  });

  return elt("span", null, "Tool: ", select);
};
```

В поле tool есть элементы `<option>` для всех определённых нами инструментов, а обработчик события «mousedown» на холсте берёт на себя обязанность вызывать функцию текущего инструмента, передавая ей

объекты event и context. Также он вызывает preventDefault, чтобы зажатие и перетаскивание мыши не вызывало выделения участков страницы.

Самый простой инструмент – линия, который рисует линии за мышью. Чтобы рисовать линию, нам надо сопоставить координаты курсора мыши с координатами точек на холсте. Вскользь упомянутый в 13 главе метод getBoundingClientRect может нам в этом помочь. Он говорит, где показывается элемент, относительно левого верхнего угла экрана. Свойства события мыши clientX и clientY также содержат координаты относительно этого угла, поэтому мы можем вычесть верхний левый угол холста из них и получить позицию относительно этого угла.

```
function relativePos(event, element) {
  var rect = element.getBoundingClientRect();
  return {x: Math.floor(event.clientX - rect.left),
          y: Math.floor(event.clientY - rect.top)};
}
```

Несколько инструментов рисования должны слушать событие «mousemove», пока кнопка мыши нажата. Функция trackDrag регистрирует и убирает событие для данных ситуаций.

```
function trackDrag(onMove, onEnd) {
  function end(event) {
    removeEventListener("mousemove", onMove);
    removeEventListener("mouseup", end);
    if (onEnd)
      onEnd(event);
  }
  addEventListener("mousemove", onMove);
  addEventListener("mouseup", end);
}
```

У неё два аргумента. Один – функция, которая вызывается при каждом событии «mousemove», а другая – функция, которая вызывается при отпускании кнопки. Каждый аргумент может быть не задан.

Инструмент для рисования линий использует две вспомогательные функции для самого рисования.

```
tools.Line = function(event, cx, onEnd) {
  cx.lineCap = "round";

  var pos = relativePos(event, cx.canvas);
  trackDrag(function(event) {
    cx.beginPath();
    cx.moveTo(pos.x, pos.y);
    pos = relativePos(event, cx.canvas);
    cx.lineTo(pos.x, pos.y);
    cx.stroke();
  }, onEnd);
};
```

Функция сначала устанавливает свойство контекста lineCap в “round”, из-за чего концы нарисованного пути становятся закруглёнными, а не квадратными, как это происходит по умолчанию. Этот трюк обеспечивает непрерывность линий, когда они нарисованы в несколько приёмов. Если рисовать линии большой ширины, вы увидите разрывы в углах линий, если будете использовать установку lineCap по умолчанию.

Затем, по каждому событию «mousemove», которое случается, пока кнопка нажата, рисуется простая линия между старой и новой позициями мыши, с использованием тех значений параметров strokeStyle и lineWidth, которые заданы в данный момент.

Аргумент onEnd просто передаётся дальше, в trackDrag. При обычном вызове третий аргумент передаваться не будет, и при использовании функции он будет содержать undefined, поэтому в конце перетаскивания ничего не произойдёт. Но он поможет нам организовать ещё один инструмент, ластик erase, используя очень небольшое дополнение к коду.

```
tools.Erase = function(event, cx) {
  cx.globalCompositeOperation = "destination-out";
  tools.Line(event, cx, function() {
    cx.globalCompositeOperation = "source-over";
  });
};
```

Свойство globalCompositeOperation влияет на то, как операции рисования на холсте меняют цвет пикселей. По умолчанию, значение свойства «source-over», что означает, что цвет, которым рисуют, накладывается поверх существующего. Если цвет непрозрачный, он просто заменит существующий, но если он частично прозрачный, они будут смешаны.

Инструмент “erase” устанавливает globalCompositeOperation в «destination-out», что имеет эффект ластика, и делает пиксели снова прозрачными.

Вот у нас уже есть два инструмента для рисования. Мы можем рисовать чёрные линии в один пиксель шириной (это задано значениями свойств холста strokeStyle и lineWidth по умолчанию), и стирать их. Работающий, хотя и примитивный, прототип программы.

Цвет и размер кисти

Предполагая, что пользователи захотят рисовать не только чёрным цветом и не только одним размером кисти, добавим элементы управления для этих настроек.

В главе 18 я обсуждал разные варианты полей формы. Среди них не было полей для выбора цвета. По традиции у браузеров нет встроенных полей для выбора цвета, но за последнее время в стандарт включили несколько новых типов полей форм. Один из них — `<input type="color">`. Среди других — «date», «email», «url» и «number». Пока ещё их поддерживают не все. Для тега `<input>` тип по умолчанию – “text”, и при использовании нового тега, который ещё не поддерживается браузером, браузеры будут обрабатывать его как текстовое поле. Значит, пользователям с браузерами, которые не поддерживают инструмент для выбора цвета, необходимо будет вписывать название цвета вместо того, чтобы выбирать его через удобный элемент управления.

```
controls.color = function(cx) {
  var input = elt("input", {type: "color"});
  input.addEventListener("change", function() {
    cx.fillStyle = input.value;
    cx.strokeStyle = input.value;
  });
  return elt("span", null, "Color: ", input);
};
```

При смене значения поля color значения свойств контекста холста fillStyle и strokeStyle заменяются на новое значение.

Настройка размера кисти работает сходным образом.

```
controls.brushSize = function(cx) {
  var select = elt("select");
  var sizes = [1, 2, 3, 5, 8, 12, 25, 35, 50, 75, 100];
  sizes.forEach(function(size) {
    select.appendChild(elt("option", {value: size},
                           size + " pixels"));
  });
  select.addEventListener("change", function() {
    cx.lineWidth = select.value;
  });
  return elt("span", null, "Brush size: ", select);
};
```

Код создаёт варианты размеров кистей из массива, и убеждается в том, что свойство холста lineWidth обновлено при выборе кисти.

Сохранение

Чтобы объяснить, как работает ссылка на сохранение, сначала мне нужно рассказать про URL с данными. В отличие от обычных http: и https:, URL с данными не

указывают на ресурс, а содержат весь ресурс в себе. Это URL с данными, содержащий простой HTML документ:

```
data:text/html,<h1 style="color:red">Hello!</h1>
```

Такие URL полезны для разных вещей, как, например, включение небольших картинок прямо в файл стилей. Они также позволяют нам ссылаться на создаваемые файлы на стороне клиента, в браузере, не перемещая их сперва на какой-либо сервер.

У элемента холста есть удобный метод toDataURL, который возвращает URL с данными, содержащий картинку на холсте в виде графического файла. Но нам не следует обновлять ссылку для сохранения при каждом изменении картинки. В случае больших картинок перемещение данных в URL занимает много времени. Вместо этого мы подключаем обновление к ссылке, чтоб она обновляла свой атрибут href каждый раз, когда она получает фокус с клавиатуры или над ней появляется курсор мыши.

```
controls.save = function(cx) {
  var link = elt("a", {href: "/"}, "Save");
  function update() {
    try {
      link.href = cx.canvas.toDataURL();
    } catch (e) {
      if (e instanceof SecurityError)
        link.href = "javascript:alert(" +
          JSON.stringify("Can't save: " + e.toString())
+ ")";
      else
        throw e;
    }
  }
  link.addEventListener("mouseover", update);
  link.addEventListener("focus", update);
  return link;
};
```

Таким образом, линк просто сидит себе тихонечко и указывает на неправильные данные, но как только пользователь приблизится к нему, он волшебным образом обновляет себя так, чтобы указывать на текущую картинку.

Если вы загрузите большую картинку, некоторые браузеры поперхнутся слишком большим URL с данными, который получится в результате. Для маленьких картинок система работает без проблем.

Но здесь мы опять сталкиваемся с деталями реализации песочницы в браузере. Когда картинка грузится с URL с другого домена, если ответ сервера не содержит заголовок, разрешающий использование ресурса с других доменов (см. главу 17), холст будет содержать информацию, которая будет видна пользователю, но не видна скрипту.

Мы могли запросить картинку с приватной информацией (график изменений банковского счёта). Если бы скрипт мог получить к ней доступ, он мог бы шпионить за пользователем.

Для предотвращения таких утечек информации, когда картинка, невидимая скрипту, будет загружена на холст, браузеры пометят его как «испорчен». Пиксельные данные, включая URL с данными, нельзя будет получить с «испорченного» холста. На него можно писать, но с него нельзя читать.

Поэтому нам нужна обработка try/catch в функции update для ссылки сохранения. Когда холст «портится», вызов toDataURL выбросит исключение, являющееся экземпляром SecurityError. В этом случае мы перенаправляем ссылку на ещё один вид URL с протоколом javascript:. Такие ссылки просто выполняют скрипт, стоящий после двоеточия, и наша ссылка покажет предупреждение, сообщающее о проблеме.

Загрузка картинок

Последние два элемента управления используются для загрузки картинок с локального диска и с URL. Нам потребуется вспомогательная функция, которая пробует загрузить картинку с URL и заменить ею содержимое холста.

```
function loadImageURL(cx, url) {
  var image = document.createElement("img");
  image.addEventListener("load", function() {
    var color = cx.fillStyle, size = cx.lineWidth;
    cx.canvas.width = image.width;
    cx.canvas.height = image.height;
    cx.drawImage(image, 0, 0);
    cx.fillStyle = color;
    cx.strokeStyle = color;
    cx.lineWidth = size;
  });
  image.src = url;
}
```

Нам надо поменять размер холста, чтобы он соответствовал картинке. Почему-то при смене размера холста его контекст забывает все настройки (fillStyle и lineWidth), в связи с чем функция сохраняет их и загружает обратно после обновления размера холста.

Элемент управления для загрузки локального файла использует технику FileReader из главы 18. Кроме используемого здесь метода readAsText у таких объектов есть метод под названием readAsDataURL – а это то, что нам нужно. Мы загружаем файл, который пользователь выбирает, как URL с данными, и передаём его в loadImageURL для вывода на холст.

```
controls.openFile = function(cx) {
  var input = elt("input", {type: "file"});
  input.addEventListener("change", function() {
    if (input.files.length == 0) return;
    var reader = new FileReader();
    reader.addEventListener("load", function() {
      loadImageURL(cx, reader.result);
    });
    reader.readAsDataURL(input.files[0]);
  });
  return elt("div", null, "Open file: ", input);
};
```

Загружать файл с URL ещё проще. Но с текстовым полем мы не знаем, закончил ли пользователь набирать в нём URL, поэтому мы не можем просто слушать события “change”. Вместо этого мы обернём поле в форму и среагируем, когда она будет отправлена – либо по нажатию Enter, либо по нажатию кнопку load.

```
controls.openURL = function(cx) {
  var input = elt("input", {type: "text"});
  var form = elt("form", null,
                 "Open URL: ", input,
                 elt("button", {type: "submit"}, "load")
);
  form.addEventListener("submit", function(event) {
    event.preventDefault();
    loadImageURL(cx, form.querySelector("input").value);
  });
  return form;
};
```

Теперь мы определили все элементы управления, требующиеся нашей программе, но нужно добавить ещё несколько инструментов.

Закругляемся

Очень просто можно добавить инструмент для вывода текста, который выводит запрос пользователю, куда он должен ввести текст.

```
tools.Text = function(event, cx) {
  var text = prompt("Text:", "");
  if (text) {
    var pos = relativePos(event, cx.canvas);
    cx.font = Math.max(7, cx.lineWidth) + "px sans-serif"
;
    cx.fillText(text, pos.x, pos.y);
  }
};
```

Можно было бы добавить полей для размера текста и шрифта, но для простоты мы всегда используем шрифт sans-serif и размер шрифта, как у текущей кисти. Минимальный размер – 7 пикселей, потому что меньше текст будет нечитаемый.

Ещё один необходимый инструмент для каляк-маляк – “аэрозоль”. Она рисует случайные точки под кистью, пока нажата кнопка мыши, создавая более или менее густые точки в зависимости от скорости движения курсора.

```
tools.Spray = function(event, cx) {
  var radius = cx.lineWidth / 2;
  var area = radius * radius * Math.PI;
  var dotsPerTick = Math.ceil(area / 30);

  var currentPos = relativePos(event, cx.canvas);
  var spray = setInterval(function() {
    for (var i = 0; i < dotsPerTick; i++) {
      var offset = randomPointInRadius(radius);
      cx.fillRect(currentPos.x + offset.x,
                  currentPos.y + offset.y, 1, 1);
    }
  }, 25);
  trackDrag(function(event) {
    currentPos = relativePos(event, cx.canvas);
  }, function() {
    clearInterval(spray);
  });
};
```

Аэрозоль использует setInterval для выплёвывания цветных точек каждые 25 миллисекунд, пока нажата кнопка мыши. Функция trackDrag используется для того, чтобы currentPos указывала на текущее положение курсора, и для выключения интервала при отпускании кнопки.

Чтобы посчитать, сколько точек нужно нарисовать каждый раз по окончанию интервала, функция подсчитывает размер области текущей кисти и делит его

на 30. Для поиска случайного положения под кистью используется функция randomPointInRadius.

```
function randomPointInRadius(radius) {
  for (;;) {
    var x = Math.random() * 2 - 1;
    var y = Math.random() * 2 - 1;
    if (x * x + y * y <= 1)
      return {x: x * radius, y: y * radius};
  }
}
```

Эта функция создаёт точки в квадрате между (-1,-1) и (1,1). Используя теорему Пифагора, она проверяет, лежит ли созданная точка внутри круга с радиусом 1. Когда такая точка находится, она возвращает её координаты, умноженные на радиус.

Цикл нужен для равномерного распределения точек. Проще было бы создавать точки в круге, взяв случайный угол и радиус и вызвав Math.sin и Math.cos для создания точки. Но тогда точки с большей вероятностью появлялись бы ближе к центру круга. Это ограничение можно обойти, но результат будет сложнее, чем предыдущий цикл.

Теперь наша программа для рисования готова. Запустите код и попробуйте.

```
<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
</body>
```

Упражнения

В программе ещё очень много чего можно улучшить. Давайте добавим ей возможностей.

Прямоугольники

Определите инструмент Rectangle, заполняющий прямоугольник (см. метод fillRect из главы 16) текущим цветом. Прямоугольник должен появляться из той точки, где пользователь нажал кнопку мыши, и до той точки, где он отпустил кнопку. Заметьте, что последнее действие может произойти левее или выше первого.

Когда это заработает, вы заметите, что изображение прямоугольника дрожит и его плохо видно. Можете ли вы придумать способ показа прямоугольника во время движения мыши, но чтобы он не выводился на холст, пока кнопка не отпущена?

Если не придумаете, вспомните о стиле position: absolute, который мы обсуждали в главе 13, который можно использовать, чтобы выводить узел поверх остального документа. Свойства pageX и pageY событий мыши можно использовать для точного расположения элемента под мышью, записывая нужные значения в стили left, top, width и height.

```
<script>
  tools.Rectangle = function(event, cx) {
    // Ваш код
  };
</script>

<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
</body>
```

Выбор цвета

Ещё один часто встречающийся инструмент – выбор цвета, который позволяет щелчком мыши на картинке выбрать цвет, который находится под курсором. Сделайте такой инструмент.

Для его изготовления понадобится доступ к содержимому холста. Метод toDataURL примерно это и делал, но получить информацию о пикселе из URL с данными

сложно. Вместо этого мы возьмём метод контекста getImageData, возвращающий прямоугольный кусок картинки в виде объекта со свойствами width, height и data. В свойстве data содержится массив чисел от 0 до 255, и для каждого пикселя хранится четыре номера — red, green, blue и alpha (прозрачность).

Этот пример получает числа из одного пикселя холста, один раз, когда тот пуст (все пиксели – прозрачные чёрные), и один раз, когда пиксель окрашен в красный цвет.

```
function pixelAt(cx, x, y) {
  var data = cx.getImageData(x, y, 1, 1);
  console.log(data.data);
}

var canvas = document.createElement("canvas");
var cx = canvas.getContext("2d");
pixelAt(cx, 10, 10);
// → [0, 0, 0, 0]

cx.fillStyle = "red";
cx.fillRect(10, 10, 1, 1);
pixelAt(cx, 10, 10);
// → [255, 0, 0, 255]
```

Аргументы getImageData показывают начальные координаты прямоугольника x и y, которые нам надо получить, за которыми идут ширина и высота.

Игнорируйте прозрачность в этом упражнении, работайте только с первыми тремя цифрами для заданного пикселя. Также не волнуйтесь по поводу обновления поля color при выборе цвета. Просто убедитесь, что fillStyle и strokeStyle контекста установлены в цвет, оказавшийся под курсором.

Помните, что эти свойства принимают любой цвет, который понимает CSS, включая запись вида rgb(R, G, B), которую вы видели в главе 15.

Метод getImageData имеет те же ограничения, что и toDataURL – он выдаст ошибку, когда на холсте содержатся пиксели картинки, скачанной с другого домена. Используйте запись try/catch для сообщения об этих ошибках через окно alert.

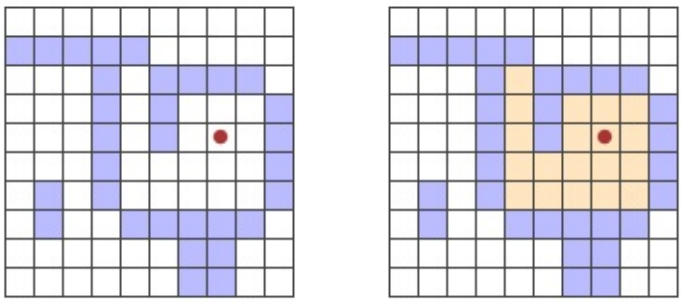
```
<script>
  tools["Pick color"] = function(event, cx) {
    // Your code here.
  };
</script>

<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
</body>
```

Заливка

Это упражнение более сложное, чем предыдущие, и оно потребует разработки нетривиального решения хитрой задачи. Убедитесь, что у вас есть свободное время и терпение перед началом работы, и не отчаивайтесь, если сразу у вас что-то не будет получаться.

Инструмент заливки окрашивает пиксель под курсором мыши и под целой группой пикселей вокруг него, имеющих тот же цвет. Для целей нашего упражнения мы будем считать, что эта группа включает все пиксели, до которых можно добраться от начального, двигаясь по одному пикселю по горизонтали и вертикали (но не по диагонали), не прикасаясь к пикселям, чей цвет отличается от исходного.

Следующая картинка демонстрирует набор пикселей, окрашиваемых, когда инструмент заливки применяется к помеченному пикселю:



Заливка не протекает через диагональные разрывы и не касается пикселей, которых нельзя достичь, даже если они того же цвета, что и исходный.

Вам вновь понадобится getImageData для выяснения цвета пикселя. Скорее всего, удобнее будет получить всю картинку за раз, а потом уже получать данные по пикселям из получившегося массива. Пиксели в массиве организованы схожим образом с решёткой из главы 7, по рядам, только каждый пиксель описывается четырьмя значениями. Первое значение для пикселя с координатами (x,y) находится на позиции (x + y × width) × 4

Включайте в рассмотрение четвёртое число (альфа), потому что нам нужно будет различать чёрные и пустые (прозрачные) пиксели.

Поиск соседних пикселей того же цвета требует пройти по поверхности пикселей вверх, вниз, влево и вправо, пока там находятся пиксели того же цвета. За первый проход всю группу пикселей найти не получится. Вместо этого нужно будет сделать что-то похожее на отслеживание в регулярных выражениях, описанное в главе 9. Когда у вас есть больше одного возможного направления, нужно сохранить все те, по которым вы прямо сейчас не идёте, и просмотреть их позже, по окончанию текущего шага.

У картинки среднего размера много пикселей. Постарайтесь свести работу программы к минимуму, или же она будет работать слишком долго. К примеру, игнорируйте пиксели, которые вы уже обрабатывали.

Рекомендую для окраски отдельных пикселей вызывать fillRect, и хранить какую-то структуру данных, где записано, какие пиксели вы уже обошли.

```
<script>
  tools["Flood fill"] = function(event, cx) {
    // Ваш код
  };
</script>

<link rel="stylesheet" href="css/paint.css">
<body>
  <script>createPaint(document.body);</script>
</body>
```

Node.js

Ученик спросил: «Программисты встарь использовали только простые компьютеры и программировали без языков, но они делали прекрасные программы. Почему мы используем сложные компьютеры и языки программирования?». Фу-Тзу ответил: «Строители встарь использовали только палки и глину, но они делали прекрасные хижины».

Мастер Юан-Ма, «Книга программирования»

На текущий момент вы учили язык JavaScript и использовали его в единственном окружении: в браузере. В этой и следующей главе мы кратко представим вам Node.js, программу, которая позволяет применять навыки JavaScript вне браузера. С ней вы можете написать всё, от утилит командной строки до динамических HTTP серверов.

Эти главы посвящены обучению важным идеям, составляющим Node.js и предназначены для передачи вам достаточного количества информации, чтобы вы могли писать полезные программы в этой среде. Они не пытаются быть всеобъемлющими справочниками по Node.

Код из предыдущих глав вы могли писать и исполнять прямо в браузере, но код из этой главы написан для Node и в браузере работать не будет.

Если вы хотите сразу запускать код из этой главы, начните с установки Node с сайта nodejs.org для вашей операционки. Также на этом сайте вы найдёте документацию по Node и его встроенным модулям.

Вступление

Одна из наиболее сложных проблем при написании систем, общающихся по сети – обработка ввода и вывода. Чтение и запись данных в сеть и из сети, на диск, и другие устройства. Перемещение данных требует времени, и грамотное планирование этих действий может сильно повлиять на время отклика системы для пользователя или сетевых запросов.

В традиционном методе обработки ввода и вывода принято, что функция, к примеру, readFile, начинает читать файл и возвращается только когда файл полностью прочитан. Это называется синхронным вводом-выводом (synchronous I/O, input/output).

Node был задуман с целью облегчить и упростить использование асинхронного I/O. Мы уже встречались с асинхронными интерфейсами, такими, как объект браузера XMLHttpRequest, обсуждавшийся в главе 17. Такой интерфейс позволяет скрипту продолжать работу, пока интерфейс делает свою, и вызывает функцию обратного вызова по окончанию работы. Таким образом в Node работает весь I/O.

JavaScript легко вписывается в систему типа Node. Это один из немногих языков, в которые не встроена система I/O. Поэтому JavaScript легко встраивается в довольно эксцентричный подход к I/O в Node и в результате не порождает две разных системы ввода и вывода. В 2009 году при разработке Node люди уже использовали I/O в браузере, основанный на обратных вызовах, поэтому сообщество вокруг языка было привычно к асинхронному стилю программирования.

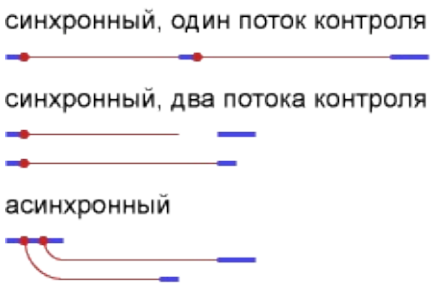
Асинхронность

Попробую проиллюстрировать разницу в синхронном и асинхронном подходах в I/O на небольшом примере, где программа должна получить два ресурса из интернета, и затем сделать что-то с данными.

В синхронном окружении очевидным способом решения задачи будет сделать запросы последовательно. У этого метода есть минус – второй запрос начнётся только после окончания первого. Общее время будет не меньше, чем сумма времени на обработку двух запросов. Это неэффективное использование компьютера, который большую часть времени будет простаивать, пока происходит передача данных по сети.

Решение проблемы в синхронной системе – запуск дополнительных потоков контроля исполнения программы (в главе 14 мы их уже обсуждали). Второй поток может запустить второй запрос, и затем оба потока будут ждать возврата результата, после чего они заново будут синхронизированы для сведения работы в один результат.

На диаграмме жирные линии обозначают время нормальной работы программы, а тонкие – время ожидания I/O. В синхронной модели время, затраченное на I/O, входит во временной график каждого из потоков. В асинхронной, запуск действия по I/O приводит к разветвлению временной линии. Поток, запустивший I/O, продолжает выполнение, а I/O выполняется параллельно ему, по окончанию работы делая обратный вызов функции.



Поток выполнения программы для синхронного и асинхронного I/O

Ещё один способ выразить эту разницу: в синхронной модели ожидание окончания I/O неявное, а в асинхронной – явное, и находится под нашим непосредственным контролем. Но асинхронность работает в обе стороны. С её помощью выражать программы, не работающие по принципу прямой линии, проще, но выражать прямолинейные программы становится сложнее.

В главе 17 я уже касался того факта, что обратные вызовы привносят кучу шума и делают программу менее упорядоченной. Является ли такой подход в общем хорошей идеей – спорный вопрос. В любом случае, требуется время, чтобы привыкнуть к нему.

Но для системы, основанной на JavaScript, я бы сказал, что использование асинхронности с обратными вызовами имеет смысл. Одна из сильных сторон JavaScript – простота, и попытки добавить в программу несколько потоков привели бы к сильному усложнению. Хотя

обратные вызовы не делают код простым, их идея очень проста и в то же время достаточно сильна для того, чтобы писать высокопроизводительные веб-серверы.

Команда node

Когда в вашей системе установлен Node.js, у вас появляется программа под названием node, которая запускает файлы JavaScript. Допустим, у вас есть файл hello.js со следующим кодом:

```
var message = "Hello world";
console.log(message);
```

Вы можете выполнить свою программу из командной строки:

```
$ node hello.js
Hello world
```

Метод console.log в Node действует так же, как в браузере. Выводит кусок текста. Но в Node текст выводится на стандартный вывод, а не в консоль JavaScript в браузере.

Если запустить node без файла, он выдаст вам строку запроса, в которой можно писать код на JavaScript и получать результат.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

Переменная process, так же как и console, доступна в Node глобально. Она обеспечивает несколько способов для инспектирования и манипулирования программой. Метод exit заканчивает процесс, и ему можно передать код статуса окончания программы, который сообщает программе, запустившей node (в данном случае, программной оболочке), завершилась ли программа удачно (нулевой код) или с ошибкой (любое другое число).

Для доступа к аргументам командной строки, переданным программе, можно читать массив строк process.argv. В него также включены имя команды node и имя вашего скрипта, поэтому список аргументов начинается с индекса 2. Если файл showargv.js содержит только инструкцию console.log(process.argv), его можно запустить так:

```
$ node showargv.js one --and two
["node", "/home/marijn/showargv.js", "one", "--and", "tw
o"]
```

Все стандартные глобальные переменные JavaScript — Array, Math, JSON, также есть в окружении Node. Но там отсутствует функционал, связанный с работой браузера, например document или alert.

Объект глобальной области видимости, который в браузере называется window, в Node имеет более осмысленное название global.

Модули

Кроме нескольких упомянутых переменных, вроде console и process, Node держит мало функционала в глобальной области видимости. Для доступа к остальным встроенным возможностям вам надо обращаться к системе модулей.

Система CommonJS, основанная на функции require, была описана в главе 10. Такая система встроена в Node и используется для загрузки всего, от встроенных модулей и скачанных библиотек до файлов, являющихся частями вашей программы.

При вызове require Node нужно преобразовать заданную строку в имя файла. Пути, начинающиеся с "/", "./" или "../", преобразуются в пути относительно текущего. "./" означает текущую директорию, "../" – директорию выше, а "/" – корневую директорию файловой системы. Если вы запросите "./world/world" из файла /home/marijn/elife/run.js, Node попробует загрузить файл /home/marijn/elife/world/world.js. Расширение .js можно опускать.

Когда передаётся строка, которая не выглядит как относительный или абсолютный путь, то предполагается, что это либо встроенный модуль, или модуль, установленный в директории node_modules. К примеру, require(«fs») выдаст вам встроенный модуль для работы с файловой системой, а require(«elife») попробует загрузить библиотеку из node_modules/elife/. Типичный метод установки библиотек – при помощи NPM, к которому я вернусь позже.

Для демонстрации давайте сделаем простой проект из двух файлов. Первый назовём main.js, и в нём будет определён скрипт, вызываемый из командной строки, предназначенный для искажения строк.

```
var garble = require("./garble");

// По индексу 2 содержится первый аргумент программы из
командной строки
var argument = process.argv[2];

console.log(garble(argument));
```

Файл garble.js определяет библиотеку искажения строк, которая может использоваться как заданной ранее программой для командной строки, так и другими скриптами, которым нужен прямой доступ к функции garble.

```
module.exports = function(string) {
  return string.split("").map(function(ch) {
    return String.fromCharCode(ch.charCodeAt(0) + 5);
  }).join("");
};
```

Замена module.exports вместо добавления к нему свойств позволяет нам экспортировать определённое значение из модуля. В данном случае, результатом запроса нашего модуля получится сама функция искажения.

Функция разбивает строку на символы, используя split с пустой строкой, и затем заменяет все символы на другие, с кодом на 5 единиц выше. Затем она соединяет результат обратно в строку.

Теперь мы можем вызвать наш инструмент:

```
$ node main.js JavaScript
Of{fXhwnuy
```

Установка через NPM

NPM, вскользь упомянутый в главе 10, это онлайн-хранилище модулей JavaScript, многие из которых написаны специально для Node. Когда вы ставите Node на компьютер, вы получаете программу npm, которая даёт удобный интерфейс к этому хранилищу.

К примеру, один из модулей NPM зовётся figlet, и он преобразует текст в “ASCII art”, рисунки, составленные из текстовых символов. Вот как его установить:

```
$ npm install figlet
npm GET https://registry.npmjs.org/figlet
npm 200 https://registry.npmjs.org/figlet
npm GET https://registry.npmjs.org/figlet/-/figlet-1.0.9
.tgz
npm 200 https://registry.npmjs.org/figlet/-/figlet-1.0.9
.tgz
figlet@1.0.9 node_modules/figlet
$ node
> var figlet = require("figlet");
> figlet.text("Hello world!", function(error, data) {
    if (error)
      console.error(error);
    else
      console.log(data);
  });
  _   _      _ _                             _     _ _
 | | | | ___| | | ___   __      _____  _ __| | __| | |
 | |_| |/ _ \ | |/ _ \  \ \ /\ / / _ \| '__| |/ _` | |
 |  _  |  __/ | | (_) |  \ V  V / (_) | |  | | (_| |_|
 |_| |_|\___|_|_|\___/    \_/\_/ \___/|_|  |_|\__,_(_)
```

После запуска npm install NPM создаст директорию node_modules. Внутри неё будет директория figlet, содержащий библиотеку. Когда мы запускаем node и вызываем require(«figlet»), библиотека загружается и мы можем вызвать её метод text, чтобы вывести большие красивые буквы.

Что интересно, вместо простого возврата строки, в которой содержатся большие буквы, figlet.text принимает функцию для обратного вызова, которой он передаёт результат. Также он передаёт туда ещё один аргумент, error, который в случае ошибки будет содержать объект error, а в случае успеха – null.

Такой принцип работы принят в Node. Для создания букв figlet должен прочесть файл с диска, содержащий буквы. Чтение файла – асинхронная операция в Node, поэтому figlet.text не может вернуть результат немедленно. Асинхронность заразительна – любая функция, вызывающая асинхронную, сама становится асинхронной.

NPM – это больше, чем просто npm install. Он читает файлы package.json, содержащие информацию в формате JSON про программу или библиотеку, в частности, на каких библиотеках она основана. Выполнение npm install в директории, содержащей такой файл, автоматически приводит к установке всех зависимостей, и в свою очередь их зависимостей. Также инструмент npm используется для размещения библиотек в онлайновом хранилище NPM, чтобы другие люди могли их находить, скачивать и использовать.

Больше мы не будем углубляться в детали использования NPM. Обращайтесь на npmjs.org за документацией поиску библиотек.

Модуль file system

Один из самых востребованных встроенных модулей Node – модуль “fs”, что означает «файловая система». Модуль обеспечивает функционал для работы с файлами и директориями.

К примеру, есть функция readFile, читающая файл и делающая обратный вызов с содержимым файла.

```
var fs = require("fs");
fs.readFile("file.txt", "utf8", function(error, text) {
  if (error)
    throw error;
  console.log("А в файле том было:", text);
});
```

Второй аргумент readFile задаёт кодировку символов, в которой нужно преобразовывать содержимое файла в строку. Текст можно преобразовать в двоичные данные разными способами, но самым новым из них является UTF-8. Если у вас нет оснований полагать, что в файле содержится текст в другой кодировке, можно смело

передавать параметр «utf8». Если вы не задали кодировку, Node выдаст вам данные в двоичной кодировке в виде объекта Buffer, а не строки. Это массивоподобный объект, содержащий байты из файла.

```
var fs = require("fs");
fs.readFile("file.txt", function(error, buffer) {
  if (error)
    throw error;
  console.log("В файле было ", buffer.length, " байт.",
              "Первый байт:", buffer[0]);
});
```

Схожая функция, writeFile, используется для записи файла на диск.

```
var fs = require("fs");
fs.writeFile("graffiti.txt", "Здесь был Node ", function(
err) {
  if (err)
    console.log("Ничего не вышло, и вот почему:", err);
  else
    console.log("Запись успешна. Все свободны.");
});
```

Здесь задавать кодировку не нужно, потому что writeFile полагает, что если ей на запись дали строку, а не объект Buffer, то её надо выводить в виде текста с кодировкой по умолчанию UTF-8.

Модуль “fs” содержит много полезного: функция readdir возвращает список файлов директории в виде массива строк, stat вернёт информацию о файле, rename переименовывает файл, unlink удаляет, и т.п. См. документацию на nodejs.org

Многие функции “fs” имеют как синхронный, так и асинхронный вариант. К примеру, есть синхронный вариант функции readFile под названием readFileSync.

```
var fs = require("fs");
console.log(fs.readFileSync("file.txt", "utf8"));
```

Синхронные функции использовать проще и полезнее для простых скриптов, где дополнительная скорость асинхронного метода не важна. Но заметьте – на время выполнения синхронного действия ваша программа полностью останавливается. Если ей надо отвечать на ввод пользователя или другим программам по сети, затыки ожидания синхронного I/O приводят к раздражающим задержкам.

Модуль HTTP

Ещё один основной модуль — «http». Он даёт функционал для создания HTTP серверов и HTTP запросов.

Вот всё, что нужно для запуска простейшего HTTP сервера:

```
var http = require("http");
var server = http.createServer(function(request, response
) {
  response.writeHead(200, {"Content-Type": "text/html"})
;
  response.write("<h1>Привет!</h1><p>Вы запросили `" +
                 request.url + "`</p>");
  response.end();
});
server.listen(8000);
```

Запустив скрипт на своей машины, вы можете направить браузер по адресу localhost:8000/hello, таким образом создав запрос к серверу. Он ответит небольшой HTML-страницей.

Функция, передаваемая как аргумент к createServer, вызывается при каждой попытке соединения с сервером. Переменные request и response – объекты, представляющие входные и выходные данные. Первый содержит информацию по запросу, например свойство url содержит URL запроса.

Чтобы отправить что-то назад, используются методы объекта response. Первый, writeHead, пишет заголовки ответа (см. главу 17). Вы даёте ему код статуса (в этом случае 200 для “OK”) и объект, содержащий значения заголовков. Здесь мы сообщаем клиенту, что он должен ждать документ HTML.

Затем идёт тело ответа (сам документ), отправляемое через response.write. Этот метод можно вызывать несколько раз, если хотите отправлять ответ по кускам, к примеру, передавая потоковые данные по мере их поступления. Наконец, response.end сигнализирует конец ответа.

Вызов server.listen заставляет сервер слушать запросы на порту 8000. Поэтому вам надо в браузере заходить на localhost:8000, а не просто на localhost (где портом по умолчанию будет 80).

Для остановки такого скрипта Node, который не завершается автоматически, потому что ожидает следующих событий (в данном случае, соединений), надо нажать Ctrl-C.

Настоящий веб-сервер делает гораздо больше того, что описано в примере. Он смотрит на метод запроса (свойство method), чтобы понять, какое действие пытается выполнить клиент, и на URL запроса, чтобы

понять, на каком ресурсе это действие должно выполняться. Далее вы увидите более продвинутую версию сервера.

Чтобы сделать HTTP-клиент, мы можем использовать функцию модуля “http” request.

```
var http = require("http");
var request = http.request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, function(response) {
  console.log("Сервис ответил с кодом ",
              response.statusCode);
});
request.end();
```

Первый аргумент request настраивает запрос, объясняя Node, с каким сервером будем общаться, какой путь будет у запроса, какой метод использовать, и т.д. Второй – функция, которую надо будет вызвать по окончанию запроса. Ей передаётся объект response, в котором содержится вся информация по ответу – к примеру, код статуса.

Как и объект response сервера, объект, возвращаемый request, позволяет передавать данные методом write и заканчивать запрос методом end. В примере не

используется write, потому что запросы GET не должны содержать данных в теле.

Для запросов на безопасные URL (HTTPS), Node предлагает модуль https, в котором есть своя функция запроса, схожая с http.request.

Потоки

Мы видели два примера потоков в примерах HTTP – объект response, в который сервер может вести запись, и объект request, который возвращается из http.request

Потоки с возможностью записи – популярная концепция в интерфейсах Node. У всех потоков есть метод write, которому можно передать строку или объект Buffer. Метод end закрывает поток, а при наличии аргумента, выведет перед закрытием кусочек данных. Обоим методам можно задать функцию обратного вызова через дополнительный аргумент, которую они вызовут по окончанию записи или закрытию потока.

Возможно создать поток, показывающий на файл, при помощи функции fs.createWriteStream. Затем можно использовать метод write для записи в файл по кусочкам, а не целиком, как в fs.writeFile.

Потоки с возможностью чтения будут чуть сложнее. Как переменная request, переданная функции для обратного вызова в сервере HTTP, так и переменная response, переданная в HTTP-клиенте, являются потоками с возможностью чтения. (Сервер читает запрос и потом пишет ответы, а клиент пишет запрос и читает ответа). Чтение из потока осуществляется через обработчики событий, а не через методы.

У объектов, создающих события в Node, есть метод on, схожий с методом браузера addEventListener. Вы даёте ему имя события и функцию, и он регистрирует эту функцию, чтоб её вызвали сразу, когда произойдёт событие.

У потоков с возможностью чтения есть события «data» и «end». Первое происходит при поступлении данных, второе – по окончанию. Эта модель подходит к потоковым данным, которые можно сразу обработать, даже если получен не весь документ. Файл можно прочесть в виде потока через fs.createReadStream.

Следующий код создаёт сервер, читающий тела запросов и отправляющий их в ответ потоком в виде текста из заглавных букв.

```
var http = require("http");
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"}
);
  request.on("data", function(chunk) {
    response.write(chunk.toString().toUpperCase());
  });
  request.on("end", function() {
    response.end();
  });
}).listen(8000);
```

Переменная chunk, передаваемая обработчику данных, будет бинарным Buffer, который можно преобразовать в строку, вызвав его метод toString, который декодирует его из кодировки по умолчанию (UTF-8).

Следующий код, будучи запущенным одновременно с сервером, отправит запрос на сервер и выведет полученный ответ:

```
var http = require("http");
var request = http.request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, function(response) {
  response.on("data", function(chunk) {
    process.stdout.write(chunk.toString());
  });
});
request.end("Hello server");
```

Пример пишет в process.stdout (стандартный вывод процесса, являющийся потоком с возможностью записи), а не в console.log. Мы не можем использовать console.log, так как он добавляет лишний перевод строки после каждого куска кода – это здесь не нужно.

Простой файловый сервер

Давайте скомбинируем наши новые знания о серверах HTTP и работе с файловой системой, и наведём мостик между ними: HTTP-сервер, предоставляющий удалённый доступ к файлам. У такого сервера много вариантов использования. Он позволяет веб-приложениям хранить и делиться данными, или может дать группе людей доступ к набору файлов.

Когда мы относимся к файлам, как к ресурсам HTTP, методы GET, PUT и DELETE можно использовать для чтения, записи и удаления файлов. Мы будем интерпретировать путь в запросе как путь к файлу.

Нам не надо открывать доступ ко всей файловой системе, поэтому мы будем интерпретировать эти пути как заданные относительно корневого каталога, и это будет каталог запуска скрипта. Если я запущу сервер из /home/marijn/public/ (или C:\Users\marijn\public\ на Windows), то запрос на /file.txt должен указать на /home/marijn/public/file.txt (или C:\Users\marijn\public\file.txt).

Программу мы будем строить постепенно, используя объект methods для хранения функций, обрабатывающих разные методы HTTP.

```
var http = require("http"), fs = require("fs");

var methods = Object.create(null);

http.createServer(function(request, response) {
  function respond(code, body, type) {
    if (!type) type = "text/plain";
    response.writeHead(code, {"Content-Type": type});
    if (body && body.pipe)
      body.pipe(response);
    else
      response.end(body);
  }
  if (request.method in methods)
    methods[request.method](urlToPath(request.url),
                            respond, request);
  else
    respond(405, "Method " + request.method +
            " not allowed.");
}).listen(8000);
```

Этот код запустит сервер, возвращающий ошибки 405 – этот код используется для обозначения того, что запрошенный метод сервером не поддерживается.

Функция respond передаётся функциям, обрабатывающим разные методы, и работает как обратный вызов для окончания запроса. Она принимает код статуса HTTP, тело, и, возможно, тип содержимого. Если переданное тело – поток с возможностью чтения, у него будет метод pipe, который используется для

передачи читаемого потока в записываемый. Если нет – предполагается, что это либо null (тело пустое), или строка, и тогда она передаётся напрямую в метод ответа end.

Чтобы получить путь из URL в запросе, функция urlToPath, используя встроенный модуль Node “url”, разбирает URL. Она принимает имя пути, нечто вроде /file.txt, декодирует, чтобы убрать экранирующие коды %20, и вставляет в начале точку, чтобы получить путь относительно текущего каталога.

```
function urlToPath(url) {
  var path = require("url").parse(url).pathname;
  return "." + decodeURIComponent(path);
}
```

Вам кажется, что функция urlToPath небезопасна? Вы правы. Вернёмся к этому вопросу в упражнениях.

Мы устроим метод GET так, чтобы он возвращал список файлов при чтении директории, и содержимое файла при чтении файла.

Вопрос на засыпку – какой тип заголовка Content-Type мы должны возвращать, читая файл. Поскольку в файле может быть всё, что угодно, сервер не может просто вернуть один и тот же тип для всех. Но NPM с этим может

помочь. Модуль mime (индикаторы типа содержимого файла вроде text/plain также называются MIME types) знает правильный тип для огромного количества расширений файлов.

Запустив следующую команду npm в директории, где живёт скрипт сервера, вы сможете использовать require(«mime») для запросов к библиотеке типов.

```
$ npm install mime
npm http GET https://registry.npmjs.org/mime
npm http 304 https://registry.npmjs.org/mime
mime@1.2.11 node_modules/mime
```

Когда запрошенного файла не существует, правильным кодом ошибки для этого случая будет 404. Мы будем использовать fs.stat для возврата информации по файлу, чтобы выяснить, есть ли такой файл, и не директория ли это.

```
methods.GET = function(path, respond) {
  fs.stat(path, function(error, stats) {
    if (error && error.code == "ENOENT")
      respond(404, "File not found");
    else if (error)
      respond(500, error.toString());
    else if (stats.isDirectory())
      fs.readdir(path, function(error, files) {
        if (error)
          respond(500, error.toString());
        else
          respond(200, files.join("\n"));
      });
    else
      respond(200, fs.createReadStream(path),
              require("mime").lookup(path));
  });
};
```

Поскольку запросы к диску занимают время, fs.stat работает асинхронно. Когда файла не существует, fs.stat передаст объект error с кодовым свойством «ENOENT» в функцию обратного вызова. Было бы здорово, если бы Node определил разные типы ошибок для разных ошибок, но такого нет. Вместо этого он выдаёт запутанные коды в стиле Unix.

Все неожиданные ошибки мы будем выдавать с кодом 500, обозначающим, что на сервере проблема – в отличие от кодов, начинающихся на 4, говорящих о

проблеме с запросом. В некоторых ситуациях это будет не совсем аккуратно, но для небольшой примерной программы этого будет достаточно.

Объект stats возвращаемый fs.stat, рассказывает нам о файле всё. Например, size – размер файла, mtime – дата модификации. Здесь нам нужно узнать, директория это или обычный файл – это нам сообщит метод isDirectory.

Для чтения списка файлов в директории мы используем fs.readdir, и через ещё один обратный вызов, возвращаем его пользователю. Для обычных файлов мы создаём читаемый поток через fs.createReadStream и передаём его в ответ, вместе с типом содержимого, который модуль “mime” выдал для этого файла.

Код обработки DELETE будет проще:

```
methods.DELETE = function(path, respond) {
  fs.stat(path, function(error, stats) {
    if (error && error.code == "ENOENT")
      respond(204);
    else if (error)
      respond(500, error.toString());
    else if (stats.isDirectory())
      fs.rmdir(path, respondErrorOrNothing(respond));
    else
      fs.unlink(path, respondErrorOrNothing(respond));
  });
};
```

Возможно, вам интересно, почему попытка удаления несуществующего файла возвращает статус 204 вместо ошибки. Можно сказать, что при попытке удалить несуществующий файл, так как файла там уже нет, то запрос уже исполнен. Стандарт HTTP поощряет людей делать идемпотентные запросы – то есть такие, при которых многократный повтор одного и того же действия не приводит к разным результатам.

```
function respondErrorOrNothing(respond) {
  return function(error) {
    if (error)
      respond(500, error.toString());
    else
      respond(204);
  };
}
```

Когда ответ HTTP не содержит данных, можно использовать код статуса 204 (“no content”). Так как нам нужно обеспечить функции обратного вызова, которые либо сообщают об ошибки, или возвращают ответ 204 в разных ситуациях, я написал специальную функцию respondErrorOrNothing, которая создаёт такой обратный вызов.

Вот обработчик запросов PUT:

```
methods.PUT = function(path, respond, request) {
  var outStream = fs.createWriteStream(path);
  outStream.on("error", function(error) {
    respond(500, error.toString());
  });
  outStream.on("finish", function() {
    respond(204);
  });
  request.pipe(outStream);
};
```

Здесь нам не нужно проверять существование файла – если он есть, мы его просто перезапишем. Опять мы используем pipe для передачи данных из читаемого потока в записываемый, в нашем случае – из запроса в файл. Если создать поток не удаётся, создаётся событие “error”, о чём мы сообщаем в ответе. Когда данные переданы успешно, pipe закроет оба потока, что приведёт к запуску события “finish”. А после этого мы можем отчитаться об успехе с кодом 204.

Полный скрипт сервера лежит тут: eloquentjavascript.net/code/file_server.js. Его можно скачать и запустить через Node. Конечно, его можно менять и дополнять для решения упражнений или экспериментов.

Утилита командной строки curl, общедоступная на unix-системах, может использоваться для создания HTTP запросов. Следующий фрагмент тестирует наш сервер. –

X используется для задания метода запроса, а –d для включения тела запроса.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

Первый запрос к file.txt завершается с ошибкой, поскольку файла ещё нет. Запрос PUT создаёт файл, и глядите-ка – следующий запрос его успешно получает. После его удаления через DELETE файл снова отсутствует.

Обработка ошибок

В коде файлового сервера есть шесть мест, где мы перенаправляем исключения, когда мы не знаем, как обрабатывать ошибки. Поскольку исключения не передаются автоматически в функции обратного вызова, но передаются им как аргументы, их надо каждый раз обрабатывать персонально. Это сводит на нет преимущество обработки исключений, а именно, возможность централизованно обрабатывать ошибки.

Что будет, когда что-то реально выбросит исключение в системе? Мы не используем блоки try, потому оно будет передано на самый верх стека вызовов. В Node это приводит к прекращению выполнения программы и выводу информации об исключении (вместе с отслеживанием стека) на стандартный вывод.

Поэтому наш сервер будет падать при возникновении проблем в коде – в отличие от проблем с асинхронностью, которые будут переданы как аргументы в функции вызова. Если нам надо обрабатывать все исключения, возникающие при обработке запроса, чтобы мы точно отправили ответ, нам надо добавлять блоки try/catch в каждом обратном вызове.

Это плохо. Много программ для Node написаны так, чтобы использовать как можно меньше работы с исключениями, подразумевая что в случае возникновения исключения программа не может его обработать, и поэтому надо падать.

Ещё один подход – использование обещаний, которые были описаны в главе 17. Они ловят исключения, выброшенные функциями обратного вызова и передают их как ошибки. В Node можно загрузить библиотеку promise и использовать её для обработки асинхронных вызовов. Немногие библиотеки Node интегрируют обещания, но обычно их довольно просто обернуть.

Отличный модуль “promise” с NPM содержит функцию denodeify, которая берёт асинхронную функцию вроде fs.readFile и преобразовывает её в функцию, возвращающую обещание.

```
var Promise = require("promise");
var fs = require("fs");

var readFile = Promise.denodeify(fs.readFile);
readFile("file.txt", "utf8").then(function(content) {
  console.log("The file contained: " + content);
}, function(error) {
  console.log("Failed to read file: " + error);
});
```

Для сравнения, я написал ещё одну версию файлового сервера с использованием обещаний, которую можно найти на eloquentjavascript.net/code/file_server_promises.js. Она почище, потому что функции теперь могут возвращать результаты, а не назначать обратные вызовы, и исключения передаются неявно.

Приведу несколько строк оттуда, чтобы продемонстрировать разницу в стилях.

Объект fsp, использующийся в коде, содержит варианты функций fs с обещаниями, обёрнутыми при помощи Promise.denodeify. Возвращаемый из обработчика метода объект, со свойствами code и body, становится

окончательным результатом цепочки обещаний, и он используется для определения того, какой ответ отправить клиенту.

```
methods.GET = function(path) {
  return inspectPath(path).then(function(stats) {
    if (!stats) // Does not exist
      return {code: 404, body: "File not found"};
    else if (stats.isDirectory())
      return fsp.readdir(path).then(function(files) {
        return {code: 200, body: files.join("\n")};
      });
    else
      return {code: 200,
              type: require("mime").lookup(path),
              body: fs.createReadStream(path)};
  });
};

function inspectPath(path) {
  return fsp.stat(path).then(null, function(error) {
    if (error.code == "ENOENT") return null;
    else throw error;
  });
}
```

Функция inspectPath – простая обёртка вокруг fs.stat, обрабатывающая случай, когда файл не найден. В этом случае мы заменяем ошибку на успех, возвращающий

null. Все остальные ошибки можно передавать. Когда обещание, возвращаемое из этих обработчиков, обламывается, сервер отвечает кодом 500.

Итог

Node – отличная простая система, позволяющая запускать JavaScript вне браузера. Изначально она разрабатывалась для работы по сети, чтобы играть роль узла в сети. Но она позволяет делать много всего, и если вы наслаждаетесь программированием на JavaScript, автоматизация ежедневных задач с Node работает отлично.

NPM предоставляет библиотеки для всего, что вам может прийти в голову (и даже для кое-чего, что вам не придёт в голову), и она позволяет скачивать и устанавливать их простой командой. Node также поставляется с набором встроенных модулей, включая “fs” для работы с файловой системой, и “http” для запуска HTTP серверов и создания HTTP запросов.

Весь ввод и вывод в Node делается асинхронно, если только вы не используете явно синхронный вариант функции, например fs.readFileSync. Вы предоставляете функции обратного вызова, а Node их вызывает в нужное время, когда операции I/O заканчивают работу.

Упражнения

И снова согласование содержания

В главе 17 первое упражнение было посвящено созданию запросов к eloquentjavascript.net/author, спрашивавших разные типы содержимого путём передачи разных заголовков Accept.

Сделайте это снова, используя функцию Node http.request. Запросите, по крайней мере, типы text/plain, text/html и application/json. Помните, что заголовки запроса можно передавать как объект в свойстве headers, первым аргументом http.request.

Выведите содержимое каждого ответа.

Устранение утечек

Для упрощения доступа к файлам я оставил работать сервер у себя на компьютере, в директории /home/marijn/public. Однажды я обнаружил, что кто-то получил доступ ко всем моим паролям, которые я хранил в браузере. Что случилось?

Если вам это непонятно, вспомните функцию urlToPath, которая определялась так:

```
function urlToPath(url) {
  var path = require("url").parse(url).pathname;
  return "." + decodeURIComponent(path);
}
```

Теперь вспомните, что пути, передаваемые в функцию “fs”, могут быть относительными. Они могут содержать путь “../” в верхний каталог. Что будет, если клиент отправит запросы на URL вроде следующих:

myhostname:8000/../.config/config/google-chrome/Default/Web%20Data
myhostname:8000/../.ssh/id_dsa
myhostname:8000/../../../etc/passwd

Поменяйте функцию urlToPath для устранения подобной проблемы. Примите во внимание, что на Windows Node разрешает как прямые так и обратные слеши для задания путей.

Кроме этого, помедитируйте над тем фактом, что как только вы выставляете сырую систему в интернет, ошибки в системе могут быть использованы против вас и вашего компьютера.

Создание директорий

Хотя метод DELETE работает и при удалении директорий (через fs.rmdir), пока сервер не предоставляет возможности создания директорий.

Добавьте поддержку метода MKCOL, который должен создавать директорию через fs.mkdir. MKCOL не является основным методом HTTP, но он существует, именно для этого, в стандарте WebDAV, который содержит расширения HTTP, чтобы использовать его для записи ресурсов, а не только для их чтения.

Общественное место в сети

Так как файловый сервер выдаёт любые файлы и даже возвращает правильный заголовок Content-Type, его можно использовать для обслуживания веб-сайта. Так как он разрешает всем удалять и заменять файлы, это был бы интересный сайт – который можно изменять, портить и удалять всем, кто может создать правильный HTTP-запрос. Но это всё равно был бы веб-сайт.

Напишите простую HTML страницу с простым файлом JavaScript. Разместите их в директории, обслуживаемой сервером и откройте в браузере.

Затем, в качестве продвинутого упражнения, скомбинируйте все полученные знания из книги, чтобы построить более дружественный интерфейс для

модификации веб-сайта изнутри самого сайта.

Используйте форму HTML (глава 18) для редактирования файлов, составляющих сайт, позволяя пользователю обновлять их на сервере через HTTP-запросы, как описано в главе 17.

Начните с одного файла, редактирование которого разрешено. Затем сделайте так, чтобы можно было выбирать файл для редактирования. Используйте тот факт, что наш файловый сервер возвращает списки файлов по запросу директории.

Не меняйте файлы непосредственно в коде файлового сервера – если вы сделаете ошибку, вы скорее всего испортите те файлы. Работайте в директории, недоступной снаружи, и копируйте их туда после тестирования.

Если ваш компьютер соединяется с интернетом напрямую, без firewall, роутера или других устройств, вы сможете пригласить друга на свой сайт. Для проверки сходите на whatismyip.com, скопируйте IP адрес в адресную строку и добавьте :8000 для выбора нужного порта. Если вы попали на свой сайт, то он доступен для просмотра всем.

Проект: веб-сайт по обмену опытом

На встречах по обмену опытом люди с общими интересами встречаются и делают небольшие неформальные презентации на тему своих знаний. На встрече по обмену опытом среди фермеров кто-нибудь может рассказать о выращивании сельдерея. На встрече программистов вы можете выступить с рассказом про Node.js

Такие встречи – отличный способ расширить свой кругозор, узнать о новинках области, или просто пообщаться с людьми со схожими интересами. Во многих городах есть встречи любителей JavaScript. Обычно их посещение бесплатное, и я нашёл те, которые посещал, дружелюбными и гостеприимными.

В последней главе-проекте мы устроим веб-сайт по обслуживанию выступлений, которые делаются на таких встречах. Представьте себе группу людей, которые регулярно встречаются в офисе одного из участников, чтобы поговорить о моноциклах. Проблема в том, что когда предыдущий организатор встреч переехал в другой город, никто не занял его место. Нам нужна система, которая позволит участникам предлагать и обсуждать темы друг с другом, без участия организатора.

Встречи моноциклистов

Как и в предыдущей главе, код написан для Node.js и запустить его в браузере не получится. Полный код доступен по ссылке.

Дизайн

У проекта есть серверная часть, написанная для Node.js, и клиентская, написанная для браузера. Серверная хранит системные данные и передаёт их клиенту. Также она отдаёт файлы HTML и JavaScript, которые создают систему на стороне клиента.

На сервере есть список тем для следующего собрания, и клиент их показывает. У каждой темы есть имя выступающего, название, описание и список комментариев. Клиент позволяет предлагать новые темы (добавлять их в список), удалять темы и комментировать существующие. Когда пользователь вносит это изменение, клиент делает HTTP-запрос, чтобы сообщить об этом серверу.

Your name: Bob

Unituning
by **Carlos**

Modifying your cycle for extra style

***Alice**: Will you talk about raising a cycle?*
***Carlos**: Definitely!*
***Alice**: I'll be there*

Add comment Delete talk

Submit a talk

Title:

Summary: Send

Будет создано приложение для показа текущих предложений тем и комментариев по ним. Когда кто-то где-то добавляет новую тему или оставляет комментарий, у всех людей, открывших страницу в браузере, изменения должны происходить мгновенно. Это непростая задача, потому что веб-сервер не может открывать соединение с клиентом, и потому что нет годного способа узнать, кто из клиентов сейчас просматривает данный веб-сайт.

Общепринятым решением проблемы являются длинные запросы (long polling), которые послужили одной из мотиваций к разработке Node.

Длинные запросы

Чтобы мгновенно оповестить клиента об изменениях, нам нужно соединение с клиентом. Браузеры традиционно не принимают запросов на соединения, и клиенты всё равно скрыты за устройствами, которые эти соединения не приняли бы, поэтому начинать соединение с сервера смысла не имеет.

Можно сделать так, чтобы клиент открывал соединение и держал его, чтобы сервер имел возможность отправлять через него информацию по необходимости.

Но запрос HTTP разрешает только простой обмен информацией – клиент отправляет запрос, сервер возвращает ответ, и всё. Есть технология под названием web sockets, которая поддерживается современными браузерами, позволяющая открывать соединения для обмена произвольными данными. Но их довольно сложно использовать.

В этой главе мы обратимся к относительно простой технологии, длинным запросам, когда клиенты постоянно запрашивают сервер о новой информации через обычные HTTP-запросы, а сервер просто медлит с ответом, когда ему нечего сообщить.

Пока клиент постоянно держит открытый запрос, он будет получать информацию с сервера немедленно. К примеру, если у Алисы в браузере открыто приложение для обмена опытом, браузер сделает запрос на обновления и будет ожидать ответа. Когда Боб из своего браузера отправит тему «Экстремальный спуск на моноцикле с горы», сервер заметит, что Алиса ждёт обновлений, и отправит информацию по новой теме в ответ на её ждущий запрос. Браузер Алисы получит данные и обновит страницу, показав новую тему.

Для предотвращения завершения соединений по таймауту (по истечению времени неактивные соединения обрываются), технология длинных запросов обычно устанавливает максимальное время для каждого запроса, по прошествии которого сервер в любом случае ответит, даже если ему нечего сообщить, а затем клиент запустит новый запрос. Периодическое обновление запроса делает технику более надёжной, позволяя клиентам восстанавливаться после временных обрывов или проблем на сервере.

У занятого сервера, использующего длинные запросы, могут висеть открытыми тысячи запросов, а, следовательно, и TCP соединений. Node хорошо подходит для такой системы, потому, что он позволяет с лёгкостью управлять многими соединениями без создания отдельных потоков.

Интерфейс HTTP

Перед тем, как мы начнём делать сервер или клиент, подумаем об их точке соприкосновения: интерфейсе HTTP, через который они взаимодействуют.

Интерфейс будет основан на JSON, и, как и в файловом сервере в главе 20, мы будем с выгодой использовать методы HTTP. Интерфейс сосредоточен вокруг пути /talks. Пути, которые не начинаются с /talks, будут использоваться для отдачи статичных файлов – HTML и JavaScript, определяющих клиентскую часть.

Запрос GET к /talks возвращает документ JSON типа этого:

```
{"serverTime": 1405438911833,
 "talks": [{"title": "Unituning ",
            "presenter": "Васисуалий",
            "summary": "Украшаем свой моноцикл",
            "comment": []}]}
```

Поле serverTime используется для надёжности длинных запросов. Вернёмся к нему позже.

Создание новой темы происходит через запрос PUT к URL вида /talks/Unituning, где часть после второго слеша – название темы. Тело запрос PUT должно содержать объект JSON, в котором описаны свойства presenter и summary.

Поскольку заголовки тем могут содержать пробелы и другие символы, которые нельзя вставлять в URL, при создании URL их надо закодировать при помощи функции encodeURIComponent.

```
console.log("/talks/" + encodeURIComponent("How to Idle"
));
// → /talks/How%20to%20Idle
```

Запрос на создание темы может выглядеть так:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{«presenter»: «Даша»,
 «summary»: «Неподвижно стоим на моноцикле»}
```

Такие URL поддерживают запросы GET для получения JSON-представления темы и DELETE для удаления темы.

Добавление комментария происходит через POST запрос к URL вида /talks/Unituning/comments, с объектом JSON, содержащим свойства author и message в теле запроса.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{«author»: «Alice»,
 «message»: «Will you talk about raising a cycle?»}
```

Для поддержки длинных запросов, запросы GET к /talks могут включать параметр под именем changesSince, показывающий, что клиенту нужны обновления, случившиеся после заданной точки во времени. Когда обновления появляются, они сразу же возвращаются.

Когда их нет, запрос задерживается, пока что-нибудь не случится, или пока не пройдёт заданный период времени (мы зададим 90 секунд).

Время используется в формате количества миллисекунд с начала 1970 года, в том же формате, что возвращает Date.now(). Чтобы удостовериться, что клиент получает все обновления, и не получает одно и то же обновление дважды, клиент должен передать время, в которое он в последний раз получил информацию с сервера. Часы сервера могут не совпадать с клиентом, и даже если б они совпадали, клиент не мог бы знать точное время, в которое сервер отправлял ответ, потому что передача данных по сети занимает время.

Поэтому в ответах на запросы GET к /talks и существует свойство serverTime. Оно сообщает клиенту точное время по часам сервера, когда были созданы передаваемые данные. Клиент просто сохраняет время и передаёт его вместе со следующим запросом, чтобы убедиться, что он получает только те обновления, которых ещё не получал.

```
GET /talks?changesSince=1405438911833 HTTP/1.1

(прошло время)

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 95

{«serverTime»: 1405438913401,
 «talks»: [{«title»: «Unituning»,
 «deleted»: true}]}
```

Когда тема меняется, создаётся или комментируется, в ответ на следующий запрос включается полная информация о теме. Когда тема удаляется, включаются только название и свойство deleted. Клиент может добавлять темы с заголовками, которые он ещё не видел, к странице, обновлять темы, которые он уже показывает, и удалять темы, которые были удалены.

Протокол, описываемый в этой главе, не осуществляет контроль доступа. Каждый может комментировать, менять и удалять темы. Так как интернет полон хулиганов, размещение такой системы в онлайне без защиты, скорее всего, закончится плохо.

Простым решением было бы разместить систему за обратным прокси – это HTTP-сервер, которая принимает соединения снаружи системы и перенаправляет их на

сервера HTTP, работающие локально. Такой proxy можно настроить, чтобы он спрашивал имя и пароль пользователя, и вы могли бы устроить так, чтобы пароль был только у членов вашей группы.

Сервер

Начнём с написания серверной части программы. Код работает на Node.js

Роутинг

Для запуска сервера будет использоваться http.createServer. В функции, обрабатывающей новый запрос, мы должны различать запросы (определяемые методом и путём), которые мы поддерживаем. Это можно сделать через длинную цепочку if / else, но можно и красивее.

Роутер – компонент, помогающий распределить запрос к функции, которая может его обработать. Можно сказать роутеру, что запросы PUT с путём, совпадающим с регуляркой /^\/talks\/(V+)$/ (что совпадает с /talks/, за которым идёт название темы), могут быть обработаны заданной функцией. Кроме того, он может помочь извлечь

осмысленные части пути, в нашем случае – название темы, заключённое в кавычки, и передать их вспомогательной функции.

В NPM есть много хороших модулей роутинга, но тут мы сами себе такой напишем, чтобы продемонстрировать принцип его работы.

Вот файл router.js, который будет запрашиваться через require из модуля сервера:

```
var Router = module.exports = function() {
  this.routes = [];
};

Router.prototype.add = function(method, url, handler) {
  this.routes.push({method: method,
                    url: url,
                    handler: handler});
};

Router.prototype.resolve = function(request, response) {
  var path = require("url").parse(request.url).pathname;

  return this.routes.some(function(route) {
    var match = route.url.exec(path);
    if (!match || route.method != request.method)
      return false;

    var urlParts = match.slice(1).map(decodeURIComponent
);
    route.handler.apply(null, [request, response]
                               .concat(urlParts));
    return true;
  });
};
```

Модуль экспортирует конструктор Router. Объект router позволяет регистрировать новые обработчики с методом add, и распределять запросы методом resolve.

Последний вернёт булевское значение, показывающее, был ли найден обработчик. Метод some массива путей будет пробовать их по очереди (в порядке, в каком они были заданы), и остановится с возвратом true, если путь найден.

Функции обработчиков вызываются с объектами request и response. Когда регулярка, проверяющая URL, возвращает группы, то представляющие их строки передаются в обработчик в качестве дополнительных аргументов. Эти строчки надо декодировать из URL-стиля %20.

Выдача файлов

Когда тип запроса не совпадает ни с одним из типов, которые обрабатывает роутер, сервер должен интерпретировать его как запрос файла из общей директории. Можно было бы использовать файловый сервер из главы 20 для выдачи этих файлов, но нам не нужна поддержка PUT и DELETE, зато нам нужны дополнительные функции типа поддержки кеширования. Поэтому, давайте использовать проверенный и протестированный файловый сервер из NPM.

Я выбрал ecstatic. Это не единственный сервер на NPM, но он хорошо работает и удовлетворяет нашим требованиям. Модуль ecstatic экспортирует функцию, которую можно вызвать с объектом конфигурации, чтобы она выдала функцию обработчика. Мы используем опцию root, чтобы сообщить серверу, где нужно искать файлы. Обработчик принимает параметры request и response, и его можно передать напрямую в createServer, чтобы создать сервер, который отдаёт только файлы. Но сначала нам нужно проверить те запросы, которые мы обрабатываем особо – поэтому мы обёртываем его в ещё одну функцию.

```
var http = require("http");
var Router = require("./router");
var ecstatic = require("ecstatic");

var fileServer = ecstatic({root: "./public"});
var router = new Router();

http.createServer(function(request, response) {
  if (!router.resolve(request, response))
    fileServer(request, response);
}).listen(8000);

Функции respond и respondJSON используются в коде сервер
а, чтобы можно было отправлять ответы одним вызовом функ
ции.

function respond(response, status, data, type) {
  response.writeHead(status, {
    "Content-Type": type || "text/plain"
  });
  response.end(data);
}

function respondJSON(response, status, data) {
  respond(response, status, JSON.stringify(data),
          "application/json");
}
```

Темы как ресурсы

Сервер хранит предложенные темы в объекте talks, у которого именами свойств являются названия тем. Они будут выглядеть как ресурсы HTTP по адресу /talks/[title], поэтому нам нужно добавить в роутер обработчиков, реализующих различные методы, которые клиенты могут использовать для работы с ними.

Обработчик для запросов GET одной темы должен найти её и либо вернуть данные в JSON, либо выдать ошибку 404.

```
var talks = Object.create(null);

router.add("GET", /^\/talks\/([^\/]+)$/,
           function(request, response, title) {
  if (title in talks)
    respondJSON(response, 200, talks[title]);
  else
    respond(response, 404, "No talk '" + title + "' foun
d");
});

Удаление темы делается удалением из объекта talks.

router.add("DELETE", /^\/talks\/([^\/]+)$/,
           function(request, response, title) {
  if (title in talks) {
    delete talks[title];
    registerChange(title);
  }
  respond(response, 204, null);
});
```

Функция registerChange, которую мы определим позже, уведомляет длинные запросы об изменениях.

Чтобы было просто получать контент тел запросов, закодированных при помощи JSON, мы определяем функцию readStreamAsJSON, которая читает всё содержимое потока, разбирает его по правилам JSON и затем делает обратный вызов.

```
function readStreamAsJSON(stream, callback) {
  var data = "";
  stream.on("data", function(chunk) {
    data += chunk;
  });
  stream.on("end", function() {
    var result, error;
    try { result = JSON.parse(data); }
    catch (e) { error = e; }
    callback(error, result);
  });
  stream.on("error", function(error) {
    callback(error);
  });
}
```

Один из обработчиков, которому нужно читать ответы в JSON – это обработчик PUT, который используется для создания новых тем. Он должен проверить, есть ли у данных свойства presenter и summary, которые должны

быть строками. Данные, приходящие снаружи, всегда могут оказаться мусором, и мы не хотим, чтобы из-за плохого запроса была сломана наша система.

Если данные выглядят приемлемо, обработчик сохраняет объект, представляющий новую тему, в объекте talks, при этом, возможно, перезаписывая существующую тему с таким же заголовком, и опять вызывает registerChange.

```
router.add("PUT", /^\/talks\/([^\/]+)$/,
           function(request, response, title) {
  readStreamAsJSON(request, function(error, talk) {
    if (error) {
      respond(response, 400, error.toString());
    } else if (!talk ||
              typeof talk.presenter != "string" ||
              typeof talk.summary != "string") {
      respond(response, 400, "Bad talk data");
    } else {
      talks[title] = {title: title,
                      presenter: talk.presenter,
                      summary: talk.summary,
                      comments: []};
      registerChange(title);
      respond(response, 204, null);
    }
  });
});
```

Добавление комментария к теме работает сходным образом. Мы используем readStreamAsJSON для получения содержимого сообщения, проверяем результирующие данные и сохраняем их как комментарий, если они приемлемы.

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           function(request, response, title) {
  readStreamAsJSON(request, function(error, comment) {
    if (error) {
      respond(response, 400, error.toString());
    } else if (!comment ||
              typeof comment.author != "string" ||
              typeof comment.message != "string") {
      respond(response, 400, "Bad comment data");
    } else if (title in talks) {
      talks[title].comments.push(comment);
      registerChange(title);
      respond(response, 204, null);
    } else {
      respond(response, 404, "No talk '" + title + "' fo
und");
    }
  });
});
```

Попытка добавить комментарий к несуществующей теме должна возвращать ошибку 404.

Поддержка длинных запросов

Самый интересный аспект сервера – часть, которая поддерживает длинные запросы. Когда на адрес /talks поступает запрос GET, это может быть простой запрос всех тем, или запрос на обновления с параметром changesSince.

Есть много различных ситуаций, в которых нам нужно отправить клиенту список тем, поэтому мы сначала определим вспомогательную функцию, присоединяющую поле serverTime к таким ответам.

```
function sendTalks(talks, response) {
  respondJSON(response, 200, {
    serverTime: Date.now(),
    talks: talks
  });
}
```

Обработчик должен посмотреть на все параметры запроса в его URL, чтобы проверить, не задан ли параметр changesSince. Если дать функции parse модуля “url” второй аргумент значения true, он также распарсит вторую часть URL – query, часть запроса. У

возвращаемого объекта будет свойство query, в котором будет ещё один объект, с именами и значениями параметров.

```
router.add("GET", /^\/talks$/, function(request, response
) {
  var query = require("url").parse(request.url, true).qu
ery;
  if (query.changesSince == null) {
    var list = [];
    for (var title in talks)
      list.push(talks[title]);
    sendTalks(list, response);
  } else {
    var since = Number(query.changesSince);
    if (isNaN(since)) {
      respond(response, 400, "Invalid parameter");
    } else {
      var changed = getChangedTalks(since);
      if (changed.length > 0)
         sendTalks(changed, response);
      else
        waitForChanges(since, response);
    }
  }
});
```

При отсутствии параметра changesSince обработчик просто строит список всех тем и возвращает его.

Иначе, сперва надо проверить параметр changeSince на предмет того, что это число. Функция getChangedTalks, которую мы вскоре определим, возвращает массив изменённых тем с некоего заданного времени. Если она возвращает пустой массив, то серверу нечего возвращать клиенту, так что он сохраняет объект response (при помощи waitForChanges), чтобы ответить попозже.

```
var waiting = [];

function waitForChanges(since, response) {
  var waiter = {since: since, response: response};
  waiting.push(waiter);
  setTimeout(function() {
    var found = waiting.indexOf(waiter);
    if (found > -1) {
      waiting.splice(found, 1);
      sendTalks([], response);
    }
  }, 90 * 1000);
}
```

Метод splice используется для вырезания куска массива. Ему задаётся индекс и количество элементов, и он изменяет массив, удаляя это количество элементов после заданного индекса. В этом случае мы удаляем один элемент – объект, ждущий ответ, чей индекс мы узнали через indexOf. Если вы передадите

дополнительные аргументы в splice, их значения будут вставлены в массив на заданной позиции, и заместят удалённые элементы.

Когда объект response сохранён в массиве waiting, задаётся таймаут. После 90 секунд он проверяет, ждёт ли ещё запрос, и если да – отправляет пустой ответ и удаляет его из массива waiting.

Чтобы найти именно те темы, которые сменились после заданного времени, нам надо отслеживать историю изменений. Регистрация изменения при помощи registerChange запомнит это изменение, вместе с текущим временем, в массиве changes. Когда случается изменение, это значит – есть новые данные, поэтому всем ждущим запросам можно немедленно ответить.

```
var changes = [];

function registerChange(title) {
  changes.push({title: title, time: Date.now()});
  waiting.forEach(function(waiter) {
    sendTalks(getChangedTalks(waiter.since), waiter.resp
onse);
  });
  waiting = [];
}
```

Наконец, getChangedTalks использует массив changes, чтобы построить массив изменившихся тем, включая объекты со свойством deleted для тем, которых уже не существует. При построении массива getChangedTalks должна убедиться, что одна и та же тема не включается дважды, так как тема могла измениться несколько раз с заданного момента времени.

```
function getChangedTalks(since) {
  var found = [];
  function alreadySeen(title) {
    return found.some(function(f) {return f.title == tit
le;});
  }
  for (var i = changes.length - 1; i >= 0; i--) {
    var change = changes[i];
    if (change.time <= since)
      break;
    else if (alreadySeen(change.title))
      continue;
    else if (change.title in talks)
      found.push(talks[change.title]);
    else
      found.push({title: change.title, deleted: true});
  }
  return found;
}
```

Вот и всё с кодом сервера. Запуск написанного кода даст вам сервер, работающий на порту 8000, который выдаёт файлы из публичной поддиректории и управляет

интерфейсом тем по адресу /talks.

Клиент

Клиентская часть веб-сайта по управлению темами состоит из трёх файлов: HTML-страница, таблица стилей и файл JavaScript.

HTML

Серверы по общепринятой схеме в случае запроса пути, соответствующего директории, отдают файл под именем index.html из этой директории. Модуль файлового сервера ecstatic поддерживает это соглашение. При запросе пути / сервер ищет файл ./public/index.html (где ./public – это корневая директория) и возвращает его, если он там есть.

Значит, если надо показать страницу, когда браузер будет запрашивать наш сервер, её надо положить в public/index.html. Вот начало файла index:

```
<!doctype html>

<title>Обмен опытом</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Обмен опытом</h1>

<p>Ваше имя: <input type="text" id="name"></p>

<div id="talks"></div>
```

Определяется заголовок и включается таблица стилей, где определяются стили – в числе прочего, рамочка вокруг тем. Затем добавлен заголовок и поле name. Пользователь должен вписать своё имя, чтобы оно было присоединено к его темам и комментариям.

Элемент `<div>` с ID “talks” будет содержать список тем. Скрипт заполняет список, когда он получает его с сервера.

Затем идёт форма для создания новой темы.

```
<form id="newtalk">
  <h3>Submit a talk</h3>
  Заголовок: <input type="text" style="width: 40em" name=
"title">
  <br>
  Summary: <input type="text" style="width: 40em" name="
summary">
  <button type="submit">Отправить </button>
</form>
```

Скрипт добавит обработчик события “submit” в форму, из которого он сможет сделать HTTP-запрос, сообщающий серверу про тему.

Затем идёт загадочный блок, у которого стиль display установлен в none, и который поэтому не виден на странице. Догадаетесь, зачем он нужен?

```
<div id="template" style="display: none">
  <div>
    <h2>{{title}}</h2>
    <div>by <span>{{presenter}}</span></div>
    <p>{{summary}}</p>
    <div></div>
    <form>
      <input type="text" name="comment">
      <button type="submit">Добавить комментарий</button>

      <button type="button">Удалить тему</button>
    </form>
  </div>
  <div>
    <span>{{author}}</span>: {{message}}
  </div>
</div>
```

Создание сложных структур DOM через JavaScript приводит к уродливому коду. Можно сделать его покрасивее при помощи вспомогательных функций типа elt из главы 13, но результат всё равно будет выглядеть хуже, чем HTML, который в каком-то смысле является языком для построения DOM-структур.

Для создания DOM-структур для тем обсуждений, наша программа определит простую систему шаблонов, которая использует скрытые структуры, включаемые в

документ, для создания новых структур – заменяя метки в файле между двойными фигурными кавычками на значения для конкретной темы.

И наконец, HTML включает файл скрипта, содержащего клиентский код.

```
<script src="skillsharing_client.js"></script>
```

Запуск

Первое, что клиент должен сделать при загрузке страницы, это запросить с сервера текущий набор тем. Так как мы будем делать много HTTP-запросов, мы определим небольшую обёртку вокруг XMLHttpRequest, которая примет объект для настройки запроса и обратного вызова по окончанию запроса.

```
function request(options, callback) {
  var req = new XMLHttpRequest();
  req.open(options.method || "GET", options.pathname, tr
ue);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(null, req.responseText);
    else
      callback(new Error("Request failed: " + req.status
Text));
  });
  req.addEventListener("error", function() {
    callback(new Error("Network error"));
  });
  req.send(options.body || null);
}
```

Начальный запрос показывает полученные темы на экране и начинает процесс длинных запросов, вызывая waitForChanges.

```
var lastServerTime = 0;

request({pathname: "talks"}, function(error, response) {
  if (error) {
    reportError(error);
  } else {
    response = JSON.parse(response);
    displayTalks(response.talks);
    lastServerTime = response.serverTime;
    waitForChanges();
  }
});
```

Перменная lastServerTime используется для отслеживания времени последнего обновления, полученного с сервера. После начального запроса, вид тем у клиента соответствует виду тем сервера, которые был у него в момент запроса. Таким образом, свойство serverTime, включаемое в ответ, предоставляет правильное начальное значение lastServerTime.

Когда запрос не удался, нам не надо, чтобы страница просто сидела и ничего не делала. Мы определим простую функцию под названием reportError, которая хотя бы покажет пользователю диалог, сообщающий об ошибке.

```
function reportError(error) {
  if (error)
    alert(error.toString());
}
```

Функция проверяет, есть ли ошибка, и выводит сообщение только при её наличии. Таким образом, мы можем напрямую передавать эту функцию в запрос для тех запросов, ответ на которые можно игнорировать. Тогда если запрос завершится с ошибкой, то об ошибке будет сообщено пользователю.

Показ тем

Чтобы иметь возможность обновлять список тем при поступлении изменений, клиент должен отслеживать темы, которые он показывает сейчас. Тогда, если поступает новая версия темы, которая уже есть на экране, её можно заменить прямо на месте обновлённой версией. Сходным образом, когда поступает информация об удалении темы, нужный элемент DOM можно удалить из документа.

Функция displayTalks используется как для построения начального экрана, так и для его обновления при изменениях. Она будет использовать объект shownTalks,

связывающий заголовки тем с узлами DOM, чтобы запомнить темы, которые уже есть на экране.

```
var talkDiv = document.querySelector("#talks");
var shownTalks = Object.create(null);

function displayTalks(talks) {
  talks.forEach(function(talk) {
    var shown = shownTalks[talk.title];
    if (talk.deleted) {
      if (shown) {
        talkDiv.removeChild(shown);
        delete shownTalks[talk.title];
      }
    } else {
      var node = drawTalk(talk);
      if (shown)
        talkDiv.replaceChild(node, shown);
      else
        talkDiv.appendChild(node);
      shownTalks[talk.title] = node;
    }
  });
}
```

Структура DOM для тем строится по шаблону, включённому в HTML документ. Сначала нужно определить instantiateTemplate, который находит и заполняет шаблон.

Параметр name – имя шаблона. Чтобы найти элемент шаблона, мы ищем элементы, у которых имя класса совпадает с именем шаблона, который является дочерним у элемента с ID “template”. Метод querySelector облегчает этот процесс. На странице есть шаблоны “talk” и “comment”.

```
function instantiateTemplate(name, values) {
  function instantiateText(text) {
    return text.replace(/\{\{(\w+)\}\}/g, function(_, na
me) {
      return values[name];
    });
  }
  function instantiate(node) {
    if (node.nodeType == document.ELEMENT_NODE) {
      var copy = node.cloneNode();
      for (var i = 0; i < node.childNodes.length; i++)
        copy.appendChild(instantiate(node.childNodes[i])
);
      return copy;
    } else if (node.nodeType == document.TEXT_NODE) {
      return document.createTextNode(
                instantiateText(node.nodeValue));
    }
  }

  var template = document.querySelector("#template ." +
name);
  return instantiate(template);
}
```

Метод cloneNode, который есть у всех узлов DOM, создаёт копию узла. Он не скопирует дочерние узлы, если не передать ему первым аргументом true. Функция instantiate рекурсивно создаёт копию шаблона, заполняя его по ходу дела.

Второй аргумент instantiateTemplate должен быть объектом, чьи свойства содержат строки, которые надо ввести в шаблон. Метка вроде будет заменена значением свойства “title”.

Этот подход к шаблонам довольно груб, но для создания drawTalk его будет достаточно.

```
function drawTalk(talk) {
  var node = instantiateTemplate("talk", talk);
  var comments = node.querySelector(".comments");
  talk.comments.forEach(function(comment) {
    comments.appendChild(
      instantiateTemplate("comment", comment));
  });

  node.querySelector("button.del").addEventListener(
    "click", deleteTalk.bind(null, talk.title));

  var form = node.querySelector("form");
  form.addEventListener("submit", function(event) {
    event.preventDefault();
    addComment(talk.title, form.elements.comment.value);
    form.reset();
  });
  return node;
}
```

После завершения обработки шаблона “talk” нужно много чего подлатать. Во-первых, нужно вывести комментарии, путём многократного добавления шаблона “comment” и добавления результатов к узлу класса «comments». Затем, обработчики событий нужно присоединить к кнопке, которая удаляет задачу и к форме, добавляющей комментарий.

Обновление сервера

Обработчики событий, зарегистрированные в drawTalk, вызывают функции deleteTalk и addComment непосредственно для действий, необходимых для удаления темы или добавления комментария. Это будет нужно для построения URL, которые ссылаются на темы с заданным именем, для которых мы определяем вспомогательную функцию talkURL.

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

Функция deleteTalk запускает запрос DELETE и сообщает об ошибке в случае неудачи.

```
function deleteTalk(title) {
  request({pathname: talkURL(title), method: "DELETE"},
          reportError);
}
```

Для добавления комментария нужно построить его представление в формате JSON и отправить его как часть POST-запроса.

```
function addComment(title, comment) {
  var comment = {author: nameField.value, message: comme
nt};
  request({pathname: talkURL(title) + "/comments",
           body: JSON.stringify(comment),
           method: "POST"},
          reportError);
}
```

Переменная nameField, используемая для установки свойства комментария author, ссылается на поле `<input>` вверху страницы, которое позволяет пользователю задать его имя. Мы также подключаем это поле к localStorage, чтобы его не приходилось заполнять каждый раз при перезагрузке страницы.

```
var nameField = document.querySelector("#name");

nameField.value = localStorage.getItem("name") || "";

nameField.addEventListener("change", function() {
  localStorage.setItem("name", nameField.value);
});

Форма внизу страницы для создания новой темы получает об
работчик событий “submit”. Этот обработчик запрещает дей
ствие по умолчанию (что привело бы к перезагрузке страни
цы), очищает форму и запускает PUT-запрос для создания т
емы.

var talkForm = document.querySelector("#newtalk");

talkForm.addEventListener("submit", function(event) {
  event.preventDefault();
  request({pathname: talkURL(talkForm.elements.title.val
ue),
           method: "PUT",
           body: JSON.stringify({
             presenter: nameField.value,
             summary: talkForm.elements.summary.value
           })}, reportError);
  talkForm.reset();
});
```

Обнаружение изменений

Хочу отметить, что разные функции, изменяющие состояние приложения, создавая или удаляя темы, или добавляя к ним комментарии, абсолютно не заботятся о том, чтобы их деятельность была видна на экране. Они просто говорят что-то серверу и надеются на механизм длинных запросов, который должен вызывать соответствующие изменения.

Учитывая созданную на сервере систему и то, как мы определили displayTalks для обработки изменений тем, которые уже есть на странице, сам механизм длинных запросов оказывается неожиданно простым.

```
function waitForChanges() {
  request({pathname: "talks?changesSince=" + lastServerT
ime},
          function(error, response) {
    if (error) {
      setTimeout(waitForChanges, 2500);
      console.error(error.stack);
    } else {
      response = JSON.parse(response);
      displayTalks(response.talks);
      lastServerTime = response.serverTime;
      waitForChanges();
    }
  });
}
```

Эта функция вызывается однажды, когда программа запускается, и затем продолжает вызывать себя, чтобы убедиться, что запросы всегда работают. Когда запрос не удаётся, мы не вызываем reportError, чтобы не раздражать пользователя всплывающим окном каждый раз при проблеме соединения с сервером. Вместо этого ошибка выводится в консоль (для облегчения отладки), и делается следующая попытка через 2.5 секунды.

Когда запрос удаётся, на экран выводятся новые данные, и lastServerTime обновляется, чтобы отражать тот факт, что мы получили данные в соответствии с этим новым моментом времени. Запрос сразу стартует заново, и ждёт следующего обновления.

Если вы запустите сервер, и откроете два окна браузера с адресом localhost:8000, вы увидите, что действия, выполняемые вами в одном окне, моментально отображаются в другом.

Упражнения

Следующие упражнения заключаются в изменении системы, описанной в этой главе. Для работы над ними, убедитесь, что вы скачали код и установили Node.js.

Сохранение состояния на диск

Сервер держит все данные в памяти. Если он упадёт или перезапустится, все темы и комментарии будут потеряны.

Расширьте его функциональность с тем, чтобы он сохранял данные на диске и автоматически загружал их при перезагрузке. Не волнуйтесь насчёт эффективности, сделайте самый простой вариант.

Обнуление полей комментариев

Общая перерисовка всех тем работает неплохо, потому что нет различия между узлом DOM и его заменой, когда они одинаковые. Но есть исключения. Если вы начнётё печатать что-либо в поле комментария к теме в одном окне браузера, а затем в другом окне добавите комментарий к этой теме, поле в первом окне будет перерисовано, и будет потеряно и его содержимое, и фокус.

При горячем обсуждении, когда несколько человек добавляют комментарии к одной теме, это очень раздражало бы. Можете ли вы придумать, как избежать этого?

Улучшенные шаблоны

Большинство шаблонизаторов делают больше, чем просто заполняют шаблоны строками. По меньшей мере они позволяют добавлять в шаблоны условия, аналогично оператору if, и повторения частей шаблона, аналогично циклам.

Если б мы могли повторять кусок шаблона для каждого элемента массива, второй шаблон («comment») был бы нам не нужен. Мы могли просто сказать шаблону “talk”, чтобы он повторялся для массива, содержащегося в свойстве comments, и создавал бы узлы, которые являются комментариями, для каждого элемента массива.

Это могло бы выглядеть так:

```
<div>
  <div template-repeat="comments">
    <span>{{author}}</span>: {{message}}
  </div>
</div>
```

Идея в следующем: когда при обработке шаблона встречается атрибут template-repeat, повторяющим шаблон, код проходит циклом по массиву, содержащемуся в свойстве, названном так же, как этот атрибут. Контекст шаблона (переменная values в

instantiateTemplate) при работе цикла показывала бы на текущий элемент массива так, чтобы метку искали бы в объекте comment, а не в теме.

Перепишите instantiateTemplate так, чтобы она это умела, и потом поменяйте шаблоны, чтоб они использовали эту возможность, и уберите лишние строки для создания комментариев из функции drawTalk.

Как бы вы организовали условное создание узлов, чтобы можно было опускать части шаблона, если определённое значение равно true или false?

А кто без скрипта?

Если кто-нибудь зайдёт на наш сайт с отключенным JavaScript, они получат сломанную неработающую страницу. Это не очень-то хорошо.

Некоторые разновидности веб-приложений не получится сделать без JavaScript. Для других не хватает финансирования или терпения, чтобы заботиться о посетителях без скриптов. Но для посещаемых страниц считается вежливым поддержать таких пользователей.

Попробуйте придумать способ, которым бы веб-сайт по обмену опытом можно было бы сделать работающим без JavaScript. Придётся ввести автоматические обновления

страниц, а перезагружать странички пользователям придётся по старинке. Но было бы неплохо уметь просматривать темы, создавать новые и отправлять комментарии.

Не заставляю вас его реализовывать. Достаточно описать возможное решение. Кажется ли вам такой вариант сайта более или менее элегантным, чем тот, что мы уже сделали?