Выразительный JavaScript

3-е издание

Современное
веб-программирование

Марейн
Хавербеке

no starch
press

ELOQUENT JAVASCRIPT

A Modern Introduction to Programming

by Marijn Haverbeke

no starch press

San Francisco

Выразительный JavaScript

Современное веб-программирование

3-е издание

Марейн Хавербеке

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2019

Марейн Хавербеке

Выразительный JavaScript. Современное веб-программирование

3-е издание

Серия «Для профессионалов»

Перевела с английского Е. Сандицкая

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Е. Рафалюк-Бузовская*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*К. Подольцева-Шабович*

ББК 32.988.02-018
УДК 004.738.5

Хавербеке Марейн

X12 Выразительный JavaScript. Современное веб-программирование. 3-е изд. — СПб.: Питер, 2019. — 480 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1226-5

«Выразительный JavaScript» позволит глубоко погрузиться в тему, научиться писать красивый и эффективный код. Вы познакомитесь с синтаксисом, стрелочными и асинхронными функциями, итератором, шаблонными строками и блочной областью видимости.

Марейн Хавербеке — практик. Получайте опыт и изучайте язык на множестве примеров, выполняя упражнения и учебные проекты. Сначала вы познакомитесь со структурой языка JavaScript, управляющими структурами, функциями и структурами данных, затем изучите обработку ошибок и исправление багов, модульность и асинхронное программирование, после чего перейдете к программированию браузеров.

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

ISBN 978-1593279509 англ.

ISBN 978-5-4461-1226-5

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 06.2019. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 06.06.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1500. Заказ № ВЗК-04150-19.
Отпечатано в АО «Первая Образцовая типография», филиал «Дом печати — ВЯТКА»
610033, г. Киров, ул. Московская, 122.

Краткое содержание

Введение 17
Глава 1. Значения, типы и операторы 27
Глава 2. Структура программы 40
Глава 3. Функции 59
Глава 4. Структуры данных: объекты и массивы 79
Глава 5. Функции высшего порядка 106
Глава 6. Тайная жизнь объектов 122
Глава 7. Проект: робот 144
Глава 8. Ошибки и дефекты 156
Глава 9. Регулярные выражения 173
Глава 10. Модули 199
Глава 11. Асинхронное программирование 214
Глава 12. Проект: язык программирования 241
Глава 13. JavaScript и браузер 257
Глава 14. Объектная модель документа 266
Глава 15. Обработка событий 289
Глава 16. Проект: игровая платформа 309
Глава 17. Рисование на холсте 335
Глава 18. HTTP и формы 362
Глава 19. Проект: растровый графический редактор 387
Глава 20. Node.js 410
Глава 21. Проект: сайт по обмену опытом 432
Советы по выполнению упражнений 453

Оглавление

Введение 17
О программировании 18
Почему так важен язык программирования 20
Что такое JavaScript 23
Код и что с ним делать 24
Обзор этой книги 25
Условные обозначения 26
Глава 1. Значения, типы и операторы 27
Значения 28
Числа 28
Арифметика 30
Специальные числа 31
Строки 31
Унарные операции 33
Логические значения 34
Сравнение 34
Логические операторы 35
Пустые значения 36
Автоматическое преобразование типов 36
Упрощенное вычисление логических операторов 38
Резюме 39
Глава 2. Структура программы 40
Выражения и инструкции 40
Привязки 41
Имена привязок 43
Окружение 44
Функции 44
Функция console.log 45
Возвращение значений 45

Последовательность выполнения 46
Условное выполнение 47
Циклы while и do 49
Код с отступами 51
Циклы for 51
Принудительный выход из цикла 52
Быстрое обновление привязок 53
Диспетчеризация по значению с помощью switch 54
Использование прописных букв 55
Комментарии 55
Резюме 56
Упражнения 57
Построение треугольника в цикле 57
FizzBuzz 57
Шахматная доска 58

Глава 3. Функции 59

Определение функции 59
Привязки и области видимости 61
Вложенные области видимости 62
Функции как значения 63
Декларативная запись 63
Стрелочные функции 64
Стек вызовов 65
Необязательные аргументы 66
Замыкание 68
Рекурсия 69
Разрастание функций 73
Функции и побочные эффекты 75
Резюме 76
Упражнения 77
Минимум 77
Рекурсия 77
Подсчет букв 78

Глава 4. Структуры данных: объекты и массивы 79

Белка-оборотень 80
Наборы данных 80
Свойства 81
Методы 82
Объекты 83
Изменяемость 86
Дневник оборотня 88
Вычисление корреляции 90

Перебор массива в цикле ... 91
Окончательный анализ ... 92
Дальнейшая массивология ... 94
Строки и их свойства ... 96
Дополнительные параметры ... 97
Объект Math ... 98
Деструктурирование ... 100
JSON ... 101
Резюме ... 102
Упражнения ... 103
Сумма диапазона ... 103
Массив в обратном порядке ... 103
Список ... 104
Глубокое сравнение ... 105

Глава 5. Функции высшего порядка ... 106

Абстракции ... 107
Абстрагирование повторов ... 108
Функции высшего порядка ... 109
Набор данных о шрифтах ... 110
Фильтрация массивов ... 112
Преобразование и отображение ... 112
Суммирование с помощью reduce ... 113
Компонуемость ... 114
Строки и коды символов ... 116
Распознавание текста ... 118
Резюме ... 120
Упражнения ... 120
Свертка ... 120
Ваш собственный цикл ... 120
Метод every ... 121
Доминирующее направление письма ... 121

Глава 6. Тайная жизнь объектов ... 122

Инкапсуляция ... 122
Методы ... 123
Прототипы ... 125
Классы ... 126
Запись классов ... 128
Переопределение производных свойств ... 129
Словари ... 130
Полиморфизм ... 132
Символы ... 133
Интерфейс итератора ... 134

Геттеры, сеттеры и статические методы ... 137
Наследование ... 138
Оператор instanceof ... 140
Резюме ... 140
Упражнения ... 141
Тип вектора ... 141
Группы ... 142
Итерируемые группы ... 142
Заимствование метода ... 143

Глава 7. Проект: робот ... 144
Деревня Медоуфилд ... 144
Задача ... 146
Постоянные данные ... 148
Моделирование ... 149
Маршрут почтового грузовика ... 151
Поиск пути ... 152
Упражнения ... 154
Измерение параметров робота ... 154
Эффективность робота ... 155
Постоянная группа ... 155

Глава 8. Ошибки и дефекты ... 156
Язык ... 156
Строгий режим ... 157
Типы ... 158
Тестирование ... 159
Отладка ... 161
Распространение ошибок ... 162
Исключения ... 164
Подчищаем за исключениями ... 166
Выборочный перехват исключений ... 168
Утверждения ... 170
Резюме ... 171
Упражнения ... 172
Повторная попытка ... 172
Запертый ящик ... 172

Глава 9. Регулярные выражения ... 173
Создание регулярных выражений ... 173
Проверка на соответствия ... 174
Множества символов ... 174
Повторяющиеся части шаблона ... 176
Группировка подвыражений ... 177

Соответствия и группы ... 178
Класс Date ... 179
Границы слов и строк ... 180
Выбор шаблонов ... 181
Механизм поиска соответствия ... 182
Поиск с возвратом ... 183
Метод replace ... 185
О жадности ... 187
Динамическое создание объектов RegExp ... 188
Метод search ... 189
Свойство lastIndex ... 189
Циклический поиск соответствий ... 191
Анализ INI-файла ... 191
Интернациональные символы ... 194
Резюме ... 195
Упражнения ... 197
Стиль цитирования ... 198
Снова числа ... 198

Глава 10. Модули ... 199
Зачем нужны модули ... 199
Пакеты ... 200
Импровизированные модули ... 202
Выполнение данных как кода ... 203
CommonJS ... 204
Модули ECMAScript ... 206
Сборка и комплектация ... 208
Структура модулей ... 209
Резюме ... 212
Упражнения ... 212
Модульный робот ... 212
Модуль Roads ... 213
Циклические зависимости ... 213

Глава 11. Асинхронное программирование ... 214
Асинхронность ... 215
Технологии воронов ... 216
Обратные вызовы ... 218
Промисы ... 220
Сбои ... 222
Сетевые трудности ... 224
Коллекции промисов ... 227
Лавина в сети ... 228
Маршрутизация сообщений ... 229
Асинхронные функции ... 232

Генераторы ... 234
Цикл событий ... 235
Дефекты асинхронного программирования ... 237
Резюме ... 238
Упражнения ... 239
Где скальпель? ... 239
Построение Promise.all ... 239

Глава 12. Проект: язык программирования ... 241
Синтаксический анализ ... 241
Интерпретатор ... 246
Специальные формы ... 248
Среда выполнения ... 250
Функции ... 251
Компиляция ... 252
Немного мошенничества ... 253
Упражнения ... 254
Массивы ... 254
Замыкание ... 255
Комментарии ... 255
Исправление области видимости ... 255

Глава 13. JavaScript и браузер ... 257
Интернет и другие сети ... 258
Web ... 259
HTML ... 260
HTML и JavaScript ... 263
В «песочнице» ... 264
Совместимость и браузерные войны ... 265

Глава 14. Объектная модель документа ... 266
Структура документа ... 266
Деревья ... 268
Стандарт ... 269
Перемещения по дереву ... 270
Поиск элементов ... 272
Изменение документа ... 273
Создание узлов ... 273
Атрибуты ... 275
Разметка ... 276
Стили ... 278
Каскадные стили ... 280
Селекторы запросов ... 282
Позиционирование и анимация ... 283
Резюме ... 286

Упражнения ... 286
Построение таблицы ... 286
Элементы по имени тега ... 287
Кошка и ее шляпа ... 287

Глава 15. Обработка событий ... 289
Обработчики событий ... 289
События и DOM-узлы ... 290
Объекты событий ... 291
Распространение событий ... 292
Действия по умолчанию ... 293
События клавиш ... 294
События мыши ... 296
Щелчки кнопкой мыши ... 296
Движения мыши ... 297
Сенсорные события ... 299
События прокрутки ... 300
События фокуса ... 301
Событие загрузки ... 302
События и цикл событий ... 303
Таймеры ... 304
Устранение повторных срабатываний ... 305
Резюме ... 307
Упражнения ... 307
Воздушный шарик ... 307
След мыши ... 308
Вкладки ... 308

Глава 16. Проект: игровая платформа ... 309
Игра ... 309
Технология ... 310
Уровни ... 311
Чтение уровня ... 312
Акторы ... 314
Инкапсуляция как бремя ... 318
Рисование ... 318
Движение и столкновения ... 324
Изменение акторов ... 328
Отслеживание нажатий клавиш ... 330
Игра в действии ... 331
Упражнения ... 333
Игра окончена ... 333
Приостановка игры ... 333
Монстр ... 334

Глава 17. Рисование на холсте ... 335
SVG ... 336
Элемент canvas ... 337
Линии и поверхности ... 338
Пути ... 339
Кривые ... 341
Рисование круговой диаграммы ... 344
Текст ... 345
Изображения ... 346
Преобразования ... 348
Сохранение и отмена преобразований ... 351
Возвращаясь к игре ... 352
Выбор графического интерфейса ... 358
Резюме ... 359
Упражнения ... 360
Фигуры ... 360
Круговая диаграмма ... 361
Прыгающий шарик ... 361
Заранее рассчитанное зеркальное отражение ... 361

Глава 18. HTTP и формы ... 362
Протокол ... 362
Браузеры и HTTP ... 365
Fetch ... 366
HTTP-«песочница» ... 368
Цените HTTP по достоинству ... 369
HTTPS и безопасность ... 370
Поля форм ... 370
Фокус ... 372
Отключенные поля ... 374
Форма в целом ... 374
Текстовые поля ... 375
Флажки и переключатели ... 377
Поля выбора ... 378
Поля выбора файлов ... 379
Хранение данных на стороне клиента ... 381
Резюме ... 384
Упражнения ... 385
Согласование содержимого ... 385
Среда выполнения JavaScript ... 386
Игра «Жизнь» Конвея ... 386

Глава 19. Проект: растровый графический редактор ... 387
Компоненты ... 388
Состояние ... 390

Построение DOM ... 392
Холст ... 392
Приложение ... 395
Инструменты рисования ... 398
Сохранение и загрузка ... 400
История действий ... 404
Давайте порисуем ... 405
Почему это так сложно? ... 406
Упражнения ... 407
Клавиатурные привязки ... 407
Эффективное рисование ... 408
Круги ... 408
Правильные линии ... 408

Глава 20. Node.js ... 410
Основы ... 411
Команда node ... 411
Модули ... 412
Установка с помощью NPM ... 414
Файлы пакетов ... 415
Версии ... 415
Модуль файловой системы ... 416
Модуль HTTP ... 418
Потоки ... 421
Файловый сервер ... 422
Резюме ... 429
Упражнения ... 429
Инструмент поиска ... 429
Создание каталога ... 430
Публичное пространство в сети ... 430

Глава 21. Проект: сайт по обмену опытом ... 432
Структура ... 433
Длительный опрос ... 434
HTTP-интерфейс ... 435
Сервер ... 437
Маршрутизация ... 437
Обслуживание файлов ... 439
Беседы как ресурсы ... 440
Поддержка длительных опросов ... 443
Клиент ... 444
HTML ... 445
Действия ... 445

Визуализация компонентов ... 447
Опросы ... 449
Приложение ... 450
Упражнения ... 451
Хранение на диске ... 451
Сброс поля комментариев ... 452
Советы по выполнению упражнений ... 453
Структура программы ... 453
Построение треугольника в цикле ... 453
FizzBuzz ... 453
Шахматная доска ... 454
Функции ... 454
Минимум ... 454
Рекурсия ... 454
Подсчет букв ... 455
Структуры данных: объекты и массивы ... 455
Сумма диапазона ... 455
Массив в обратном порядке ... 456
Список ... 457
Глубокое сравнение ... 457
Функции высшего порядка ... 458
Метод every ... 458
Доминирующее направление письма ... 458
Тайная жизнь объектов ... 459
Тип вектора ... 459
Группы ... 459
Итерируемые группы ... 459
Заимствование метода ... 460
Проект: робот ... 460
Измерение параметров робота ... 460
Эффективность робота ... 460
Постоянная группа ... 461
Ошибки и дефекты ... 461
Повторная попытка ... 461
Запертый ящик ... 462
Регулярные выражения ... 462
Стиль цитирования ... 462
Снова числа ... 462
Модули ... 463
Модульный робот ... 463
Модуль Roads ... 464
Циклические зависимости ... 464

Асинхронное программирование ... 465
Где скальпель? ... 465
Построение Promise.all ... 465
Проект: язык программирования ... 466
Массивы ... 466
Замыкание ... 466
Комментарии ... 467
Изменение области видимости ... 467
Объектная модель документа ... 467
Построение таблицы ... 467
Элементы по имени тега ... 468
Кошка и ее шляпа ... 468
Обработка событий ... 468
Воздушный шарик ... 468
След мыши ... 469
Вкладки ... 469
Проект: игровая платформа ... 470
Приостановка игры ... 470
Монстр ... 471
Рисование на холсте ... 471
Фигуры ... 471
Круговая диаграмма ... 472
Прыгающий шарик ... 473
Заранее рассчитанное зеркальное отражение ... 473
HTTP и формы ... 474
Согласование содержимого ... 474
Среда выполнения JavaScript ... 474
Игра «Жизнь» Конвея ... 474
Проект: растровый графический редактор ... 475
Клавиатурные привязки ... 475
Эффективное рисование ... 475
Круги ... 476
Правильные линии ... 476
Node.js ... 477
Инструмент поиска ... 477
Создание каталога ... 478
Публичное пространство в сети ... 478
Проект: сайт по обмену опытом ... 479
Хранение на диске ... 479
Сброс поля комментариев ... 479

Введение

> Мы думаем, что создаем систему для своих собственных целей. Мы считаем, что формируем ее по своему образу... Но на самом деле компьютер не похож на нас. Это проекция очень малой части нас самих: части, которая отвечает за логику, порядок, ясность и соблюдение правил.
>
> *Эллен Ульман. Рядом с машиной: технофилия и ее недостатки*

Это книга об обучении компьютеров. Сегодня компьютер встречается чаще, чем отвертка, но он гораздо сложнее, и заставить его делать то, что нам нужно, не всегда легко.

Если вы хотите, чтобы компьютер выполнил стандартную и понятную задачу, такую как просмотр электронной почты или выполнение функций калькулятора, то достаточно открыть соответствующее приложение и приступить к работе. Но для уникальных или открытых задач, скорее всего, приложения не найдется.

И здесь нам на помощь приходит программирование. *Программирование* — это процесс создания *программы*, набора точных инструкций, говорящих компьютеру, что делать, поскольку компьютер — тупое и педантичное устройство, программирование которого большей частью утомительное и досадное занятие.

К счастью, если вы способны это преодолеть и, возможно, даже получать удовольствие от строгого мышления в терминах, понятных тупой машине, программирование может принести пользу. Оно позволяет вам за считаные секунды делать вещи, выполнение которых вручную может длиться бесконечно. Это способ заставить ваш компьютер сделать то, что он не мог делать раньше. А еще — отличная тренировка абстрактного мышления.

Программирование большей частью выполняется на языках программирования. *Язык программирования* — это искусственный язык, созданный для написания инструкций для компьютеров. Интересно, что самый эффективный из найденных нами способов общения с компьютером мы в значительной мере заимствовали из того, как мы общаемся друг с другом. Подобно человеческим языкам, компьютерные языки позволяют разными способами сочетать слова и фразы, что дает возможность выражать все новые и новые понятия.

Было время, когда языковые интерфейсы, такие как командные строки BASIC и DOS 1980-х и 1990-х годов, являлись основным способом взаимодействия с компьютерами. Затем они повсеместно были заменены визуальными интерфейсами — такие интерфейсы проще освоить, хотя они предоставляют меньше свободы. Но компьютерные языки все еще сохраняются — надо только знать, где искать. Один из таких языков, JavaScript, встроен во все современные браузеры и, таким образом, доступен практически на любом устройстве.

Эта книга познакомит вас с данным языком, и вы узнаете, какие полезные и забавные вещи можно делать с его помощью.

О программировании

В этой книге я не только объясню JavaScript, но и познакомлю вас с основными принципами программирования. Программировать, оказывается, трудно. Основные правила просты и понятны, но программы, построенные на основе таких правил, со временем становятся достаточно сложными, вводят собственные правила и собственную сложность. Вы сами создаете свой лабиринт, в котором потом будет легко заблудиться.

Временами чтение этой книги будет вам даваться с трудом. Если вы новичок в программировании, то вам встретится много незнакомого материала, который придется переварить. Многие его части будут *сочетаться между собой*, так что вам потребуется усвоить эти дополнительные связи.

Вам придется приложить определенные усилия. Пытаясь изо всех сил следовать книге, не спешите с выводами о своих способностях. С вами все в порядке — вам только нужно во всем разобраться. Сделайте перерыв, перечитайте материал и убедитесь, что, читая, вы понимаете примеры программ и упражнений. Обучение — трудная работа, но все, что вы изучите, будет принадлежать вам и облегчит последующее обучение.

> Когда активная деятельность перестает приносить пользу, начинайте тихо собирать информацию; когда сбор информации перестает приносить пользу, ложитесь спать.
>
> *Урсула К. Ле Гуин. Левая рука Тьмы*

В программе есть много всего. Это и кусок текста, набранный программистом; и движущая сила, которая заставляет компьютер делать то, что он делает; это и данные в памяти компьютера, но одновременно и управление действиями, выполняемыми в той же памяти. Попытки провести аналогии между программами и уже знакомыми нам объектами, как правило, терпят неудачу. Одна из приблизительно подходящих — это машина: в ней обычно задействовано много отдельных частей. И чтобы понять принцип их работы, нужно рассмотреть, как они взаимодействуют, обеспечивая работу целого.

Компьютер — физическая машина, играющая роль хозяина для этих нематериальных машин. Сами компьютеры могут выполнять только до глупого простые вещи. Причина, по которой они так полезны, заключается в том, что они делают это поразительно быстро. Программа может изобретательно комбинировать невероятное количество простых действий и таким образом делать очень сложные вещи.

Программа — это мысленная конструкция. Построить ее ничего не стоит, и она ничего не весит, она легко растет под нашими пальцами, печатающими на клавиатуре.

Но без должного внимания размер и сложность программы выходят из-под контроля, что способно привести в замешательство даже ее создателя. Держать программы под контролем — главная проблема программирования. Пока программа работает, все отлично. Искусство программирования — это умение контролировать сложность. Покорить великолепную программу — значит сделать ее простой в своей сложности.

Некоторые программисты считают, что лучше всего справиться с проблемой, если использовать в программах лишь небольшой набор хорошо понятных приемов. Они разработали строгие правила («рекомендуемые методы»), предписывающие формы, которые должны иметь программы, и советуют тщательно придерживаться их, чтобы оставаться в безопасной зоне.

Это не только скучно, но и неэффективно. Новые проблемы часто требуют новых решений. Сфера программирования молода и все еще быстро развивается, и она достаточно разнообразна, чтобы в ней оставалось место

для совершенно разных подходов. При разработке программ сделано много ужасных ошибок; вам следует двигаться вперед и тоже совершать ошибки, чтобы в них разбираться. Понимания того, как выглядит хорошая программа, можно достичь на практике, но не выучить, руководствуясь списком правил.

Почему так важен язык программирования

Поначалу, когда компьютеры только появились, языков программирования не существовало. Программы выглядели примерно 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
```

Эта программа складывает числа от 1 до 10 и выводит результат: `1 + 2 + ... + + 10 = 55`. Она могла бы работать на простой гипотетической машине. Для программирования первых компьютеров требовалось установить в правильное положение множество переключателей или же пробить отверстия в картонных перфокартах и скормить их компьютеру. Представляете, какой утомительной была эта процедура и сколько в ней возникало ошибок! Даже написание простых программ требовало большого умения и дисциплины, а о сложных нечего было и думать.

Конечно, программист, вводивший вручную такие тайные коды в виде битов (единиц и нулей), в глубине души начинал чувствовать себя могущественным волшебником. В смысле удовлетворения от работы это чего-то да стоит.

Каждая строка предыдущей программы содержит одну инструкцию. По-русски это можно описать так.

1. Сохраните номер 0 в ячейке памяти 0.
2. Сохраните номер 1 в ячейке памяти 1.
3. Сохраните значение из ячейки памяти 1 в ячейке памяти 2.

4. Вычтите число 11 из значения в ячейке памяти 2.
5. Если значение в ячейке памяти 2 является числом 0, продолжайте выполнять инструкции, начиная с пункта 9.
6. Прибавьте значение из ячейки памяти 1 к значению из ячейки памяти 0.
7. Прибавьте число 1 к значению из ячейки памяти 1.
8. Продолжайте выполнение инструкций, начиная с пункта 3.
9. Выведите значение из ячейки памяти 0.

Это уже гораздо понятнее, чем суп из битов, но все еще маловразумительно. Гораздо удобнее использовать в инструкциях имена вместо чисел и номеров ячеек памяти:

```
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".
```

Теперь вы видите, как работает программа? В первых двух строках двум ячейкам памяти присваиваются начальные значения: `total` будет использоваться для получения результата вычислений, а в `count` будет записано число, с которым мы сейчас работаем.

Наиболее странно, пожалуй, выглядят строки, где используется `compare`. Программа хочет узнать, равно ли значение `count` 11, чтобы решить, следует ли прекратить работу. Поскольку наша гипотетическая машина довольно примитивна, она может только проверить, равно ли число нулю, и на основе этого принять решение. Поэтому программа задействует ячейку памяти с именем `compare` для вычисления значения `count - 11` и принимает соответствующее решение. Следующие две строки добавляют к результату значение `count` и увеличивают `count` на 1 каждый раз, когда программа решает, что значение `count` еще не равно 11.

Вот та же программа на JavaScript:

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

В этой версии добавилось еще несколько улучшений. Самое главное, теперь не нужно указывать способ, которым программа должна перепрыгивать назад и вперед, — об этом заботится конструкция `while`. Программа продолжает выполнять расположенный под `while` блок (заключенный в фигурные скобки), пока выполняется заданное в `while` условие. Это условие `count <= 10`, что означает «`count` меньше или равно 10». Нам больше не нужно создавать временное значение и сравнивать его с нулем, что было просто неинтересной подробностью. Одна из возможностей языков программирования заключается в том, что они сами позаботятся о неинтересных для нас деталях.

В конце программы, после того как выполнена конструкция `while`, для вывода результата используется операция `console.log`.

Наконец, вот как могла бы выглядеть программа, если бы в нашем распоряжении были удобные операции `range` и `sum`, которые соответственно создавали бы коллекцию чисел в заданном диапазоне и вычисляли ее сумму:

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

Мораль этой истории в том, что одна и та же программа может быть создана длинной или короткой, нечитаемой и читаемой. Первая версия нашей программы была совершенно непонятной, тогда как последняя написана почти на обычном английском языке: «запишите (log) сумму (sum) чисел в диапазоне (range) от 1 до 10». (В следующих главах вы научитесь определять операции, такие как `sum` и `range`.)

Хороший язык программирования помогает программисту, позволяя ему говорить о действиях, которые должен выполнить компьютер, на более высоком уровне. Язык программирования дает возможность опустить детали, предоставляет удобные строительные блоки (такие как `while` и `console.log`), позволяет определять собственные строительные блоки (такие как `sum` и `range`) и легко компоновать их.

Что такое JavaScript

JavaScript появился в 1995 году как способ программирования веб-страниц в браузере Netscape Navigator. С тех пор язык был принят во всех остальных ведущих графических браузерах. Это сделало возможным применение современных веб-приложений — приложений, с которыми можно взаимодействовать напрямую, не перезагружая страницу при каждом действии. JavaScript также используется на более традиционных сайтах для выполнения различных интерактивных «умных» действий.

Важно отметить, что JavaScript никогда не имел ничего общего с языком программирования под названием Java. Похожее название было продиктовано не здравым смыслом, а маркетинговыми соображениями. Когда появился JavaScript, язык Java активно продавался и приобретал популярность. Кто-то решил, что будет хорошей идеей бесплатно прокатиться на волне чужого успеха. Теперь приходится за это расплачиваться.

После того как JavaScript прижился за пределами Netscape, был написан стандартный документ. Он оговаривал, как должен работать JavaScript, чтобы различные программы, утверждающие, что поддерживают JavaScript, в действительности имели в виду один и тот же язык. Это так называемый стандарт ECMAScript, по названию организации Ecma International, которая выполнила стандартизацию. На практике термины ECMAScript и JavaScript взаимозаменяемы — представляют собой два названия одного и того же языка.

О JavaScript рассказывают *ужасные* вещи. И многие из них правда. Когда мне впервые пришлось написать кое-что на JavaScript, он вызвал у меня отвращение. Он принимал почти все, что я вводил, но интерпретировал все совершенно не так, как я имел в виду. Во многом это было связано с тем, что я, конечно, тогда понятия не имел, что делаю, но здесь есть и реальная проблема: JavaScript смехотворно либерален в отношении того, что он позволяет делать. Идея такого подхода заключалась в том, чтобы облегчить программирование на JavaScript для новичков. В действительности это главным образом затрудняет поиск проблем в программах, поскольку система не указывает вам на них.

Однако у такой гибкости есть свои преимущества. Она оставляет место для множества приемов, которые невозможны в более жестких языках, и, как мы увидим (например, в главе 10), ее можно использовать для преодоления отдельных недостатков JavaScript. Хорошо изучив язык и поработав с ним некоторое время, я по-настоящему полюбил JavaScript.

JavaScript претерпел несколько версий. Версия ECMAScript 3 широко поддерживалась во времена, когда JavaScript только находился на пути к своему господству, примерно между 2000 и 2010 годами. В это время велась работа над амбициозной версией 4, в которой планировался ряд радикальных улучшений и расширений языка. Такое радикальное изменение живого, широко используемого языка оказалось политически трудным, и в 2008 году работы над версией 4 были прекращены, что привело к появлению в 2009 году гораздо менее амбициозной версии 5, которая внесла лишь отдельные бесспорные улучшения. Затем в 2015 году вышла версия 6, серьезное обновление, включающее в себя некоторые идеи, запланированные для версии 4. С тех пор каждый год появляются небольшие обновления.

Тот факт, что язык развивается, означает, что браузеры должны постоянно идти с ними в ногу, и если вы используете старый браузер, он может не поддерживать все функции. Разработчики языка стараются не вносить никаких изменений, которые могли бы нарушить работоспособность существующих программ, поэтому старые программы на новых браузерах по-прежнему будут работать. В книге я использую версию JavaScript 2017 года.

Браузеры не единственные платформы, где применяется JavaScript. Некоторые базы данных, такие как MongoDB и CouchDB, используют JavaScript в качестве языка сценариев и запросов. Ряд платформ для программирования настольных ПК и серверов, в частности проект Node.js (тема главы 20), предоставляют среду для программирования на JavaScript вне браузера.

Код и что с ним делать

Код — это текст, из которого состоит программа. Большинство глав данной книги содержат довольно много кода. Я считаю, что его чтение и написание являются неотъемлемой частью обучения программированию. Постарайтесь не просто просматривать примеры, а внимательно читать, чтобы понять каждый из них. Сначала это может быть медленно, код будет выглядеть запутанным, но я обещаю, что вы быстро освоитесь. То же самое касается упражнений. Не думайте, что вы все поняли, пока действительно не напишете рабочее решение.

Я советую вам проверять свои решения упражнений в реальном интерпретаторе JavaScript. Таким образом вы получите немедленную обратную связь — работает ли то, что вы сделали, и я надеюсь, у вас появится соблазн экспериментировать и выходить за рамки упражнений.

Самый простой способ выполнить пример кода из этого издания и поэкспериментировать с ним — найти его в онлайн-версии книги по адресу https://eloquentjavascript.net. На данной странице можно щелкнуть на любом примере кода, чтобы отредактировать его, запустить и увидеть результат, который он выдает. Чтобы поработать с упражнениями, перейдите по адресу https://eloquentjavascript.net/code, где размещен начальный код для каждого упражнения, что позволяет взглянуть на решения.

Если вы хотите выполнять программы, описанные в этой книге, вне ее сайта, то потребуется определенная осторожность. Многие примеры являются независимыми и должны работать в любой среде JavaScript. Но код из последних глав часто написан для конкретной среды (браузера или Node.js) и может выполняться только там. Кроме того, во многих главах описаны более крупные программы, и приведенные там фрагменты кода зависят друг от друга или от внешних файлов. «Песочница», встроенная в сайт книги, содержит ссылки на ZIP-файлы. Они включают в себя все сценарии и файлы данных, необходимые для выполнения кода данной главы.

Обзор этой книги

Эта книга делится на три большие части. В первых 12 главах обсуждается язык JavaScript. Следующие семь глав посвящены браузерам и тому, как JavaScript используется для их программирования. Наконец, две главы посвящены Node.js, еще одной среде для программирования на JavaScript.

На протяжении всей книги вам встретится пять *глав проектов*, в которых описаны более крупные примеры программ, чтобы вы могли почувствовать вкус настоящего программирования. В порядке их появления мы будем работать над созданием робота доставки, языка программирования, игровой платформы, растрового графического редактора и динамического сайта.

Языковая часть книги начинается с четырех глав, которые познакомят вас с основной структурой языка JavaScript. Вы узнаете, что такое управляющие структуры (такие как ключевое слово `while`, уже встречавшееся вам во введении), функции (написание собственных строительных блоков) и структуры данных. После этого вы сможете писать простейшие программы. Далее, в главах 5 и 6, описаны способы использования функций и объектов, позволяющие писать более *абстрактный* код и контролировать его сложность.

После главы первого проекта будет продолжена языковая часть книги — следующие главы посвящены обнаружению и исправлению ошибок, регулярным выражениям (важный инструмент для работы с текстом), модульности (еще одна защита от сложности) и асинхронному программированию (работа с событиями, которые длятся какое-то время). Первую часть книги завершает глава второго проекта.

Во второй части, в главах с 13-й по 19-ю, описаны инструменты, к которым имеет доступ браузер с поддержкой JavaScript. Вы научитесь отображать элементы на экране (главы 14 и 17), реагировать на ввод данных пользователем (глава 15) и обмениваться ими по сети (глава 18). В данной части также содержатся две главы проектов.

После этого в главе 20 описывается Node.js, а в главе 21 создается небольшой сайт с использованием указанного инструмента.

Условные обозначения

Моноширинным шрифтом в этой книге набраны элементы программ — иногда это самостоятельные фрагменты, а иногда упоминания в тексте частей ближайшей программы. Программы (некоторые из них вы уже видели) написаны следующим образом:

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

Иногда, чтобы показать, что программа выводит на выходе, ожидаемый вывод записывается после нее с двумя косыми чертами и стрелочкой в начале строки:

```
console.log(factorial(8));
// → 40320
```

Удачи!

1 Значения, 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 остается скрытой внутри...
>
> *Мастер Юан-Ма. Книга программирования*

Внутри компьютерного мира существуют только данные. Вы можете читать данные, изменять данные, создавать новые данные, но с тем, что не является данными, вы ничего не можете сделать. Все эти данные хранятся в виде длинных последовательностей битов, и, таким образом, принципиально все данные представляют собой одно и то же.

Биты — это что угодно, что может принимать два значения, обычно описываемые как ноль и единица. Внутри компьютера биты представлены как высокий или низкий электрический заряд, сильный или слабый сигнал, блестящая или тусклая точка на поверхности компакт-диска. Любая дискретная информация может быть представлена в виде последовательности нулей и единиц — следовательно, в битах.

Например, можно выразить в битах число 13. Это похоже на представление десятичного числа, но вместо десяти разных цифр у нас есть только две, и вес каждой цифры, если смотреть справа налево, увеличивается в два раза. Вот биты, образующие число 13, с показанным под ними весом цифр:

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

Таким образом, получаем двоичное число 00001101. Его ненулевые цифры соответствуют 8, 4 и 1, что в сумме дает 13.

Значения

Представьте себе море битов, океан битов. В энергозависимом хранилище данных (рабочей памяти) обычного современного компьютера более 30 миллиардов бит. В энергонезависимом хранилище (на жестком диске или его эквиваленте), как правило, их на несколько порядков больше.

Чтобы иметь возможность работать с таким количеством битов и не потеряться среди них, нужно разделить их на куски, представляющие отдельные фрагменты информации. В среде JavaScript эти куски называются *значениями*. Все значения состоят из битов, но все они играют разные роли. Каждое значение имеет тип, который определяет его роль. Одни значения являются числами, другие — фрагментами текста, третьи — функциями и т. д.

Чтобы создать значение, достаточно просто вызвать его по имени. Это удобно — вам не приходится собирать строительные материалы для значений или платить за них. Вы просто вызываете его, и — вжжух — оно у вас есть. Разумеется, значения создаются из ничего. Каждое должно храниться где-то, и если вы хотите использовать огромное количество значений одновременно, то вам может не хватить памяти. К счастью, это становится проблемой, только если все значения нужны вам одновременно. Как только значение больше не используется, оно исчезает, оставляя после себя свободные биты для повторного применения в качестве строительного материала следующему поколению значений.

В этой главе представлены атомарные элементы программ JavaScript, то есть простые типы значений и операторы, которые взаимодействуют с ними.

Числа

Значения *числового* типа, что неудивительно, являются числовыми значениями. В программе JavaScript они записываются так:

Если использовать это в программе, то в памяти компьютера появится битовая комбинация, соответствующая числу 13.

Для хранения одиночного числового значения в JavaScript применяется фиксированное количество битов — 64. Существует много вариантов того, что можно сделать с 64 битами, и это означает, что количество чисел, которые могут быть представлены таким образом, ограничено. Имея N десятичных цифр, можно представить 10^N чисел. Аналогично, имея 64 двоичных числа, можно представить 2^{64} различных чисел, что составляет около 18 квинтиллионов (18 с 18 нулями). Это много.

Раньше компьютерная память была гораздо меньше, и люди старались использовать для представления чисел группы из 8 или 16 бит. Такие маленькие числа было легко случайно *переполнить* — в итоге получалось число, которое не укладывалось в данное количество битов. Сегодня даже компьютеры, умещающиеся в кармане, имеют достаточно памяти, поэтому можно безбоязненно использовать 64-битные блоки, и остается беспокоиться о переполнении только при работе с действительно астрономическими числами.

Однако не все целые числа, которые меньше 18 квинтиллионов, соответствуют числу JavaScript. Такие биты также хранят и отрицательные числа, поэтому один бит указывает знак числа. Еще бóльшая проблема заключается в необходимости представления нецелых чисел. Для этого несколько битов используется для хранения положения десятичной точки. Фактическое максимальное целое число, которое может быть сохранено, не превышает 9 квадриллионов (15 нолей), что все еще довольно много.

Дробные числа записываются с помощью точки:

```
9.81
```

Для очень больших или очень маленьких чисел также можно использовать экспоненциальную запись с добавлением буквы *e* (от слова «экспонента»), за которой следует экспонента числа:

```
2.998e8
```

Это означает $2{,}998 \times 10^8 = 299\,800\,000$.

Вычисления с целыми *(integer)* числами, меньшими, чем вышеупомянутые 9 квадриллионов, гарантированно всегда будут точными. К сожалению,

о вычислениях с дробными числами этого, как правило, сказать нельзя. Подобно тому как число π (пи) не может быть точно выражено конечным числом десятичных цифр, многие числа частично теряют точность, когда для их хранения доступно только 64 бита. Это досадно, но на практике вызывает проблемы только в специфических ситуациях. Об этом важно помнить и рассматривать дробные цифровые числа как приблизительные, а не как точные значения.

Арифметика

Основное, что обычно делают с числами, — выполняют арифметические действия. Арифметические операции, такие как сложение или умножение, принимают два числовых значения и производят из них новое число. Вот как они выглядят в JavaScript:

```
100 + 4 * 11
```

Символы `+` и `*` называются *операторами*. Первый из них обозначает сложение, второй — умножение. Если поместить оператор между двумя значениями, то он будет применен к ним и будет создано новое значение.

Но значит ли этот пример «прибавить 4 к 100 и умножить результат на 11», или же умножение выполняется до сложения? Как вы уже, возможно, догадались, сначала выполняется умножение. Но, как и в математике, эту последовательность можно изменить, заключив операцию сложения в скобки.

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

Для вычитания предназначен оператор `-`, а деление выполняется с помощью оператора `/`.

При использовании нескольких операторов без скобок порядок их выполнения определяется *приоритетом* операторов. Как видно в примере, умножение выполняется перед сложением. Оператор `/` имеет тот же приоритет, что и `*`. Аналогичное правило действует для `+` и `-`. Если несколько операторов с одинаковым приоритетом появляются рядом, как в выражении `1 - 2 + 1`, то они применяются слева направо: `(1 - 2) + 1`.

Пускай эти правила приоритета вас не беспокоят. Если сомневаетесь, просто добавьте скобки.

Есть еще один арифметический оператор, который не так легко узнать. Символ `%` используется для представления операции *остатка от деления*. `X % Y` — это остаток от деления `X` на `Y`. Например, `314 % 100` равно `14`, а `144 % 12` равно `0`. Приоритет оператора остатка такой же, как и у умножения и деления. Данный оператор также часто называют *делением по модулю*.

Специальные числа

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

Первые два — это `Infinity` и `-Infinity`, представляющие положительную и отрицательную бесконечность. `Infinity - 1` — это все еще `Infinity` и т. д. Однако не стоит слишком доверять вычислениям на основе бесконечности. Подобное не имеет математического смысла и быстро приведет к следующему специальному числу: `NaN`.

`NaN` означает «не число», хотя это значение числового типа. Такой результат можно получить, если, например, попытаться вычислить `0/0` (разделить ноль на ноль), `Infinity - Infinity` или выполнить любую другую числовую операцию, которая не дает осмысленного результата.

Строки

Следующий основной тип данных — это строка. Строки используются для представления текста. Их записывают, заключая содержимое в кавычки.

```
`В море`
"На волнах океана"
'Плавает в океане'
```

Для обозначения строк можно задействовать одинарные, двойные или обратные кавычки — лишь бы их вид в начале и конце строки совпадал.

Между кавычками можно поместить практически все что угодно — из всего этого JavaScript сделает строковое значение. Но есть несколько символов, которые усложняют дело. Очевидно, что трудно разместить внутри строки текст в кавычках. *Переводы строки* (символы, которые создаются при нажатии клавиши Enter) могут быть включены без экранирования только в том случае, если строка заключена в обратные одиночные кавычки (\`).

Чтобы сделать возможной вставку в строку подобных символов, используется следующая запись: всякий раз, когда внутри кавычек встречается обратная косая черта (\), это означает, что следующий символ имеет специальное значение. Это называется *экранированием* символа. Кавычка, перед которой стоит обратная косая черта, не завершает строку, а становится ее частью. Если после обратной косой черты стоит символ n, он интерпретируется как перевод строки. Аналогично t после обратной косой черты означает символ табуляции. Рассмотрим следующую строку:

```
"Это первая строка,\na это вторая"
```

На самом деле здесь содержится следующий текст:

```
Это первая строка,
а это вторая
```

Конечно, бывают ситуации, когда нужно, чтобы обратная косая черта в строке была просто обратной косой, а не специальным кодом. Если две обратные косые черты идут подряд, они сливаются и в результирующем строковом значении останется только одна. Например, строку *«Символ новой строки записывается как "\n".»* можно записать так:

```
"Символ новой строки записывается как \"\\n\"."
```

Внутри компьютера строки также представляются в виде последовательности битов. Способ, которым JavaScript это делает, основан на стандарте *Unicode*. Данный стандарт присваивает число любому символу, который вам когда-либо понадобится, включая буквы греческого, арабского, японского, армянского и других алфавитов. Если у нас есть число для каждого символа, то строку можно описать последовательностью чисел.

Именно это и делает JavaScript. Но здесь есть сложность: в представлении JavaScript каждый элемент занимает 16 бит, что позволяет описать до 2^{16} различных символов. Но в Unicode определено больше символов — на данный момент примерно вдвое больше. Поэтому некоторые символы, такие как многочисленные смайлики, занимают в строках JavaScript два «знакоместа». Мы вернемся к этому в главе 5.

Строки нельзя делить, умножать или вычитать, но оператор + к ним применим. Он не складывает, а объединяет — выполняет конкатенацию, склеивает две строки. Следующее выражение создает строку «конкатенация»:

```
"кон" + "кате" + "на" + "ция"
```

Для строковых значений существует ряд функций *(методов)*, которые можно использовать для выполнения с ними других операций. Подробнее об этом я расскажу в главе 4.

Строки, заключенные в одинарные и двойные кавычки, ведут себя практически одинаково. Единственное различие состоит в том, какой тип кавычек нужно экранировать внутри них. Строки, заключенные в обратные кавычки, обычно называемые *литералами шаблонов*, позволяют выполнять еще несколько хитрых трюков. В них можно не только разделять строки, но и встраивать другие значения.

```
`половина от 100 равна ${100/2}`
```

Если записать что-то внутри литерала шаблона `${}`, то будет вычислен результат, а затем он будет преобразован в строку и вставлен на это место. В данном примере получится *«половина от 100 равна 50»*.

Унарные операции

Не все операторы обозначаются одним символом. Отдельные записываются словами — как, например, оператор `typeof`, результатом которого является строковое значение, соответствующее типу значения, предоставленного оператору.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

Мы будем использовать `console.log` в примерах кода, чтобы показать, что хотим видеть результат вычисления чего-либо. Подробнее об этом читайте в следующей главе.

Все остальные представленные операторы работают с двумя значениями, но `typeof` принимает только одно. Операторы, задействующие два значения, называются *бинарными*, а операторы, принимающие одно значение, — *унарными*. Оператор «минус» можно использовать в качестве как бинарного, так и унарного оператора.

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

Логические значения

Часто полезно иметь значение, которое различает только две возможности, такие как «да» и «нет» или «включено» и «выключено». Для этой цели в JavaScript есть логический (булев) тип. Он имеет только два значения: «истина» и «ложь», которые записываются соответственно в виде слов `true` и `false`.

Сравнение

Вот один из способов получить логическое значение:

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

Знаки > и < являются традиционными обозначениями операций «больше» и «меньше» соответственно. Это бинарные операторы. Результатом их применения будет логическое значение, которое показывает, истинно ли данное выражение.

Строки также можно сравнивать аналогичным способом.

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

Строки сравниваются просто по алфавиту, но не совсем так, как вы ожидаете увидеть в словаре: заглавные буквы всегда «меньше» строчных, поэтому `"Z" < "a"`, а неалфавитные символы (`!`, `-` и т. д.) тоже участвуют в сравнении. При сравнении строк JavaScript перебирает символы слева направо, последовательно сравнивая их коды Unicode.

Есть и другие подобные операторы: `>=` (больше или равно), `<=` (меньше или равно), `==` (равно) и `!=` (не равно).

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

В JavaScript существует только одно значение, которое не равно самому себе, и это `NaN` («не число»).

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

Предполагается, что `NaN` обозначает результат бессмысленных вычислений и как таковой он не равен результату любых *других* бессмысленных вычислений.

Логические операторы

Существует также ряд операций, которые применяются к самим логическим значениям. JavaScript поддерживает три логических оператора: «И», «ИЛИ» и «НЕ». Они могут быть использованы для «рассуждения» о логических значениях.

Оператор `&&` представляет логическое «И». Это бинарный оператор, и его результат истинен, только если истинны оба заданные ему значения.

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

Оператор `||` обозначает логическое «ИЛИ». Его результатом является истина, если истинно хотя бы одно из переданных ему значений.

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

Операция «НЕ» обозначается восклицательным знаком (`!`). Это унарный оператор, который меняет заданное значение на противоположное: результатом `!true` будет `false`, а результатом `!false` — `true`.

При совместном использовании логических, арифметических и других операторов не всегда очевидно, когда нужны скобки. На практике обычно достаточно знать, что из уже рассмотренных нами операторов `||` имеет самый низкий приоритет, затем идет `&&`, потом операторы сравнения (`>`, `==` и т. д.), а затем остальные. Такой порядок был выбран для того, чтобы в типичных выражениях, подобных представленному ниже, требовалось как можно меньше скобок:

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

Последний логический оператор, о котором я расскажу, не унарный и не двоичный, а *троичный*, то есть работает с тремя значениями. Он записывается с помощью вопросительного знака и двоеточия, например:

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

Этот оператор называется *условным* (или иногда просто *троичным*, поскольку он единственный такой оператор в языке). Значение слева от знака вопроса «выбирает», какое из двух других значений станет результатом операции. Когда первое значение истинно, будет выбрано второе значение, а когда оно ложно — третье.

Пустые значения

Есть два специальных значения, обозначаемые как `null` и `undefined`, которые используются для описания отсутствия *осмысленного* значения. Сами по себе они являются значениями, но не несут никакой информации.

Результатом многих операций языка, не дающих осмысленного значения (некоторые из них мы увидим позже), часто выступает `undefined` просто потому, что необходимо выдать *какое-то* значение.

Разница между `undefined` и `null` появилась случайно в ходе разработки JavaScript, и, как правило, она не имеет значения. В тех случаях, когда вам действительно важны эти значения, я рекомендую рассматривать их как взаимозаменяемые.

Автоматическое преобразование типов

Во введении я упоминал, что JavaScript совершает все возможное, чтобы выполнить практически любую программу, которую вы ему предлагаете, даже если эта программа делает странные вещи. Это хорошо видно в следующих выражениях:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
```

```
// → NaN
console.log(false == 0)
// → true
```

Когда оператор применяется к «неправильному» типу значения, JavaScript незаметно для вас преобразует это значение в нужный тип, используя набор правил, которые часто не соответствуют вашим желаниям или ожиданиям. Это называется *приведением типов*. `null` в первом выражении заменяется на `0`, а `"5"` во втором выражении превращается в `5` (строка становится числом). Однако в третьем выражении оператор `+` пытается сначала выполнить конкатенацию строк, а затем сложение чисел, поэтому `1` преобразуется в `"1"` (число — в строку).

Когда что-то, что нельзя преобразовать в число очевидным способом (например, `"пять"` или `undefined`), все же преобразуется в число, получается значение `NaN`. Результатом последующих арифметических операций с `NaN` будет `NaN`. Поэтому, если вы обнаружите данное значение там, где его не должно быть, ищите случайные преобразования типов.

При сравнении значений одного и того же типа с использованием оператора `==` результат легко предсказуем: это должно быть `true`, когда значения одинаковы, если только они не равны `NaN`. Но если типы различаются, то JavaScript задействует сложный и запутанный набор правил, чтобы определить, что делать. В большинстве случаев он просто пытается преобразовать одно из значений в тип другого. Но если с любой стороны от оператора появляется `null` или `undefined`, то результатом будет `true`, только если обе стороны равны `null` или же обе равны `undefined`.

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

Такое поведение часто бывает полезным. Если вы хотите проверить, это определенное значение или же `null` или `undefined`, то можете сравнить его с `null` с помощью оператора `==` (или `!=`).

Но как быть, если вы хотите убедиться, что какое-либо значение точно равно `false`? Выражения типа `0 == false` и `"" == false` являются истинными вследствие автоматического преобразования типов. На тот случай, если вы *не хотите*, чтобы выполнялись какие-либо преобразования типов, есть два дополнительных оператора: `===` и `!==`. Первый из них проверяет, *точно* ли

значение равно другому, а второй — *точно* ли оно не равно. Таким образом, `"" === false` будет ложным, как и ожидалось.

В целях защиты от неожиданных преобразований типов я рекомендую использовать трехсимвольные операторы сравнения. Но если вы уверены, что типы с обеих сторон будут одинаковыми, то проблем с применением более коротких операторов не возникнет.

Упрощенное вычисление логических операторов

Логические операторы `&&` и `||` по-разному обрабатывают значения разных типов. Они преобразуют значение с левой стороны от оператора в логический тип, чтобы решить, что делать, но в зависимости от оператора и результата преобразования они возвращают либо *исходное* левое значение, либо правое значение.

Например, оператор `||` вернет значение, расположенное слева от него, если оно может быть преобразовано в `true`, в противном случае он вернет значение, расположенное справа. Этот эффект ожидаем, если оба значения являются логическими либо аналогичными значениями других типов.

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

Мы можем использовать подобную функциональность как способ вернуться к значению по умолчанию. Если у нас есть значение, которое может быть пустым, то мы можем поставить после него `||` и затем — заменяющее значение. Если исходное значение может быть преобразовано в `false`, то мы получим вместо него замену. Согласно правилам преобразования строк и чисел в логические значения, `0`, `NaN` и пустая строка (`""`) эквивалентны `false`, тогда как все остальные значения — `true`. Таким образом, `0 || -1` даст `-1`, а `"" || "!?"` — `"!?"`.

Оператор `&&` работает похожим образом, но наоборот. Когда значение слева от него может быть преобразовано в `false`, то возвращается это значение; в противном случае он возвращает значение, расположенное справа.

Другим важным свойством двух данных операторов является то, что их правая часть вычисляется только при необходимости. Например, в случае

`true || X`, независимо от того, что такое `X` — даже если какое-то ужасное выражение, — результатом будет `true` и `X` никогда не будет вычислен. То же самое относится к `false && X`, чьим результатом является `false`: `X` будет игнорироваться. Это называется *упрощенным вычислением*.

Условный оператор работает аналогичным образом. Из второго и третьего значений вычисляется только то, которое было выбрано.

Резюме

В этой главе мы рассмотрели четыре типа значений JavaScript: числа, строки, логические и неопределенные значения.

Данные значения создаются путем ввода их имени (`true`, `null`) или собственно значения (`13`, `"abc"`). Значения можно использовать совместно и преобразовывать одно в другое с помощью операторов. Мы познакомились с бинарными операторами для выполнения арифметических действий (`+`, `-`, `*`, `/` и `%`), конкатенацией строк (`+`), операторами сравнения (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) и логическими операторами (`&&`, `||`), а также с несколькими унарными операторами (`-` для отрицательного числа, `!` для логического отрицания и `typeof` — для поиска типа значения) и троичным оператором (`?:`) для выбора одного из двух значений в зависимости от третьего значения.

Это дает нам достаточно информации для использования JavaScript в качестве карманного калькулятора, но не более того. В следующей главе мы будем объединять выражения в простейшие программы.

2 Структура программы

> Сердце мое сияет ярко-красным светом под тонкой, прозрачной кожей, и 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, Почемучкин (трогательный) гайд по Ruby*

В этой главе мы приступим к тому, что действительно можно назвать *программированием*. Мы расширим наше владение языком JavaScript за пределы уже знакомых нам существительных и отрывков предложений до такой степени, что сможем выразить осмысленную прозу.

Выражения и инструкции

В главе 1 мы создали значения и применили к ним операторы для получения новых значений. Создание таких значений — суть любой программы на JavaScript. Но эта суть, чтобы быть полезной, должна быть представлена в виде более обширной структуры. Именно этим мы сейчас и займемся.

Фрагмент кода, в котором создается значение, называется *выражением*. Каждое значение, написанное буквально (например, `22` или `"psychoanalysis"`), является выражением. Выражение, заключенное в скобки, также является выражением, как и бинарный оператор, применяемый к двум выражениям, или унарный оператор, применяемый к одному.

Именно в этом частично заключается красота языкового интерфейса. Выражения могут содержать в себе другие выражения, подобно предложениям человеческого языка: сложное предложение включает в себя простые и т. д. Это позволяет создавать выражения, которые описывают сколь угодно сложные вычисления.

Если выражение соответствует фрагменту предложения, то *инструкция* JavaScript соответствует полному предложению. Программа представляет собой последовательность инструкций.

Простейший вид инструкции — это инструкция, после которой стоит точка с запятой. Вот программа:

```
1;
!false;
```

Но это бесполезная программа. Выражение может просто создавать значение, которое затем будет использовано кодом, заключающим в себе это выражение. Инструкция же самодостаточна, она чего-то стоит, только если влияет на мир. Инструкция может выводить что-то на экран — это считается изменением мира — или изменить внутреннее состояние машины таким образом, что это повлияет на следующие инструкции. Такие изменения называются *побочными эффектами*. Операторы в предыдущем примере просто создают значения `1` и `true`, а затем сразу отбрасывают их. Это никак не влияет на окружающий мир. Если вы запустите эту программу, ничего заметного не произойдет.

Иногда JavaScript позволяет не ставить точку с запятой в конце инструкции. В других случаях она там необходима, иначе следующая строка будет считаться частью предыдущей инструкции. Правила, по которым можно не ставить точку с запятой, довольно сложны, и легко допустить ошибку. Поэтому в данной книге в конце каждой инструкции, где должна стоять точка с запятой, она будет там стоять. Я рекомендую вам поступать так же — по крайней мере пока вы не познакомитесь ближе с последствиями пропуска точек с запятой.

Привязки

Как программа поддерживает внутреннее состояние? Как она может что-то запомнить? Мы видели, как создаются новые значения из старых, но при этом старые не меняются, а новое должно быть сразу же использовано, иначе оно тут же пропадет. Чтобы получать и сохранять значения, в JavaScript есть штуки, называемые *привязками* или *переменными*:

```
let caught = 5 * 5;
```

Это второй вид инструкции. Специальное *(ключевое)* слово `let` указывает на то, что в данном предложении будет определена привязка. Дальше следует имя привязки и, если мы хотим сразу присвоить ей значение, оператор `=` и выражение.

В предыдущей инструкции была создана привязка с именем `caught` и задействована для захвата числа, полученного путем умножения 5 на 5.

После того как привязка определена, ее имя можно использовать как выражение. Значением такого выражения является значение, которое привязка содержит в данный момент. Например:

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

Если привязка указывает на значение, это не означает, что она всегда связана с ним. К уже существующей привязке можно в любое время применить оператор `=`, чтобы отменить привязку к текущему значению и сделать так, чтобы она указывала на новое значение.

```
let mood = "легкое";
console.log(mood);
// → легкое
mood = "тяжелое";
console.log(mood);
// → тяжелое
```

Привязки лучше представлять себе не как коробки, а как щупальца: привязки не *содержат* значения, а *захватывают* их. Две привязки могут ссылаться на одно и то же значение. Программа имеет доступ только к тем значениям, на которые есть ссылки. Если вам нужно что-то запомнить, вы отращиваете «щупальце», чтобы удерживать это, или захватываете его одним из уже существующих «щупалец».

Рассмотрим еще один пример. Чтобы запомнить количество долларов, которые вам должен Луиджи, вы создаете привязку. А когда он вам вернет 35 долларов, вы дадите этой привязке новое значение.

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

Когда вы создаете привязку, не давая ей значения, щупальцу нечего схватить и оно повисает в воздухе. Если вы попробуете узнать значение пустой привязки, то получите значение `undefined`.

Один оператор `let` может определять несколько привязок. Определения должны разделяться запятыми.

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

Вместо `let` для создания привязок можно также использовать слова `var` и `const`.

```
var name = "Аида";
const greeting = "Привет, ";
console.log(greeting + name);
// → Привет, Аида
```

Первый вариант, `var` (сокращенное *variable*), — это способ объявления привязок в версиях JavaScript до 2015 года. В следующей главе мы еще вернемся к нему и рассмотрим, чем именно он отличается от `let`. Пока запомните, что в основном он делает то же самое, что и `let`, но в данной книге мы будем его использовать редко, потому что некоторые его свойства способны внести путаницу.

Слово `const` происходит от *constant*. Оно определяет постоянную привязку, указывающую на одно и то же значение на протяжении всей своей жизни. Это полезный вид привязок, так как позволяет дать имя значению, чтобы впоследствии можно было легко к нему обращаться.

Имена привязок

Именем привязки может быть любое слово. В имя привязки могут входить цифры — например, `catch22` является допустимым именем, — но имя не должно начинаться с цифры. В имени привязки может присутствовать знак доллара (`$`) или подчеркивания (`_`), но не должно быть других знаков препинания или специальных символов.

Слова со специальным значением, такие как `let`, являются ключевыми словами и не могут служить именами привязок. Есть также ряд слов, которые зарезервированы для использования в будущих версиях JavaScript

и которые также нельзя задействовать в качестве имен привязок. Полный список ключевых и зарезервированных слов довольно длинный.

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

Не трудитесь запоминать этот список. Если вы попробуете создать привязку, именем которой является зарезервированное слово, возникнет непредвиденная синтаксическая ошибка.

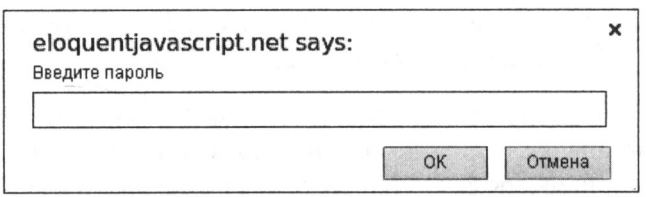
Окружение

Множество привязок и их значений, которые существуют в данный момент времени, называется *окружением*. При запуске программы это окружение не является пустым. Оно всегда содержит привязки, выступающие частью языкового стандарта, а также, в большинстве случаев, привязки, которые обеспечивают способы взаимодействия с окружающей системой. Например, в браузере есть функции для взаимодействия с загруженным в данный момент сайтом и для чтения информации, вводимой с помощью мыши и клавиатуры.

Функции

Многие значения, представляемые в окружении по умолчанию, имеют *function*. Функция — это часть программы, обернутая в значение. Такие значения могут *применяться* для выполнения обернутой программы. Например, в окружении браузера привязка `prompt` содержит функцию, которая выводит небольшое диалоговое окно с запросом на ввод данных пользователем. Эта функция используется так:

```
prompt("Введите пароль");
```



Выполнение функции называется ее *вызовом* или *применением*. Чтобы вызвать функцию, можно также поставить скобки после выражения, которое производит значение функции. Как правило, вы будете напрямую использовать имя привязки, содержащей функцию. Значения в скобках передаются программе, находящейся внутри функции. В данном примере функция `prompt` применяет строку, которую мы ей передаем, в качестве текста, выводимого в диалоговом окне. Значения, передаваемые функциям, называются *аргументами*. Разным функциям могут потребоваться разное количество и разные типы аргументов.

В современном веб-программировании функция `prompt` не используется — в основном потому, что не позволяет управлять внешним видом получаемого диалогового окна, но она может быть полезной в учебных программах и экспериментах.

Функция console.log

В предыдущих примерах я применял для вывода значений функцию `console.log`. Большинство систем JavaScript (включая все современные браузеры и Node.js) предоставляют функцию `console.log`, которая передает аргументы на *какое-нибудь* устройство вывода текста. В браузерах этот вывод попадает в консоль JavaScript. По умолчанию данная часть интерфейса браузера скрыта, но в большинстве браузеров она открывается, если нажать F12 или, на Mac, COMMAND-OPTION-I. Если это не сработает, найдите в меню пункт Developer Tools (Инструменты разработчика) или аналогичный.

Несмотря на то что в именах привязок нельзя использовать точки, в `console.log` точка есть. Потому что `console.log` не является простой привязкой. В действительности это выражение, которое извлекает свойство `log` из значения, хранящегося в привязке `console`. О том, что именно это значит, вы узнаете в главе 4.

Возвращение значений

Отображение диалогового окна или вывод текста на экран являются *побочными эффектами*. Многие функции полезны именно благодаря побочным

эффектам, которые они производят. Кроме того, функции могут создавать значения, и в этом случае они полезны и без побочных эффектов. Например, функция `Math.max` принимает любое количество числовых аргументов и возвращает наибольшее из них.

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

О функции, которая создает значение, говорят, что она *возвращает* это значение. Все, что создает значение, в JavaScript является выражением, а значит, вызовы функций могут применяться в выражениях большего размера. В следующем примере вызов функции `Math.min`, выступающей противоположностью `Math.max`, используется как часть выражения сложения:

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

В следующей главе вы узнаете, как писать собственные функции.

Последовательность выполнения

Если программа содержит более одной инструкции, то они выполняются так, как если бы это был рассказ: по порядку, сверху вниз. В следующем примере программа состоит из двух инструкций. Первая предлагает пользователю ввести число, а вторая, которая выполняется после первой, выводит квадрат этого числа.

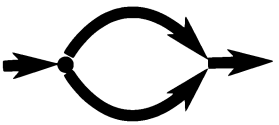
```
let theNumber = Number(prompt("Введите число"));
console.log("Это число является квадратным корнем из " +
            theNumber * theNumber);
```

Функция `Number` преобразует значение в число. Это преобразование необходимо, поскольку результатом `prompt` является строковое значение, а нам нужно число. Для строкового и логического типов существуют аналогичные функции `String` и `Boolean` соответственно.

Вот довольно простое схематическое представление линейной последовательности выполнения программы:

Условное выполнение

Не каждая программа представляет собой прямую дорогу. Например, мы можем создать перекресток, на котором программа будет выбирать нужный путь в зависимости от ситуации. Это называется *условным выполнением*.



В JavaScript условное выполнение создается с помощью ключевого слова `if`. В простейшем случае мы хотим, чтобы некоторый код выполнялся тогда и только тогда, когда выполняется определенное условие. Например, мы можем захотеть, чтобы квадрат введенного числа выводился только в том случае, если пользователь ввел именно число.

```
let theNumber = Number(prompt("Введите число"));
if (!Number.isNaN(theNumber)) {
  console.log("Это число является квадратным корнем из " +
              theNumber * theNumber);
}
```

После такого изменения программы если вы введете слово «попугай», то ничего не будет выведено.

Ключевое слово `if` выполняет или не выполняет инструкцию в зависимости от значения логического выражения. Решающее выражение записывается после ключевого слова и помещается в скобки, за которыми следует выполняемая инструкция.

Функция `Number.isNaN` — это стандартная функция JavaScript, возвращающая `true` только в том случае, если переданный ей аргумент имеет значение `NaN`. Функция `Number` возвращает значение `NaN`, если передать ей строку, которая не представляет собой корректную запись числа. Таким образом, условие можно прочитать как «сделать это, только если `theNumber` является числом».

Инструкция, которая стоит после `if` в этом примере, заключена в фигурные скобки (`{` и `}`). Фигурные скобки можно использовать для того, чтобы сгруппировать любое количество операторов в один оператор, называемый *блоком*. В данном случае скобки можно было бы пропустить, поскольку они

содержат только один оператор; но, чтобы избежать необходимости каждый раз думать о том, нужны ли они, большинство программистов JavaScript используют скобки для каждой обернутой инструкции, как здесь. В данной книге мы по большей части будем придерживаться этого соглашения, за исключением отдельных однострочных инструкций.

```
if (1 + 1 == 2) console.log("Это выражение истинно");
// → Это выражение истинно
```

Часто желательно иметь не только код, который выполняется при определенном условии, но и код, обрабатывающий другой вариант. Такой альтернативный путь представлен второй стрелкой на рисунке. Для создания двух отдельных, альтернативных путей выполнения программы совместно с `if` используют ключевое слово `else`.

```
let theNumber = Number(prompt("Введите число"));
if (!Number.isNaN(theNumber)) {
  console.log("Это число является квадратным корнем из " +
              theNumber * theNumber);
} else {
  console.log("Эй, почему вы не ввели число?");
}
```

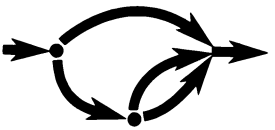
Если у вас есть более двух вариантов выбора, то можно «связать» вместе несколько пар `if/else`, например:

```
let num = Number(prompt("Введите число"));

if (num < 10) {
  console.log("Маленькое");
} else if (num < 100) {
  console.log("Среднее");
} else {
  console.log("Большое");
}
```

Программа сначала проверит, меньше ли `num`, чем 10. Если это так, то она выберет данную ветку, выведет слово «маленькое» и завершится. Если это не так, то программа перейдет к ветви `else`, внутри которой есть второе условие `if`. Если это второе условие (`<100`) выполняется, то есть число находится в диапазоне от 10 до 100, то будет выведено слово «среднее». Если же это не так, то будет выбрана вторая и последняя ветвь `else`.

Схема выполнения данной программы выглядит примерно так:



Циклы while и do

Рассмотрим программу, которая выводит все четные числа от 0 до 12. Вот один из способов написать такую программу:

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

Это работает, но идея написания программы состоит в том, чтобы работы было *меньше*, а не больше. Если бы нам понадобилось вывести все четные числа до 1000, то такой подход был бы непригоден. Нам нужен способ выполнить фрагмент кода несколько раз. Такой вариант последовательности выполнения называется *циклом*.

Зацикливание последовательности выполнения позволяет вернуться в некоторую точку программы, где мы уже были, и повторить ее из текущего состояния программы. Если объединить это с привязкой, которая умеет считать, то получим что-то вроде этого:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
// ... и так далее
```

Инструкция, начинающаяся с ключевого слова `while`, создает цикл. За словом `while` следует выражение в скобках, а затем инструкция — точно так же, как в `if`. Цикл продолжает выполнять эту инструкцию до тех пор, пока выражение выдает значение, которое при преобразовании в логический тип равно `true`.

Привязка `number` показывает, как привязка может управлять ходом выполнения программы. При каждом повторном выполнении цикла `number` получает значение, которое на 2 больше предыдущего. В начале каждого повтора значение `number` сравнивается с числом 12, чтобы определить, следует ли завершить работу программы.

Теперь в качестве примера, который действительно делает что-то полезное, мы можем взять программу, вычисляющую и показывающую значение 2^{10} (от 2 в степени 10). Мы используем две привязки: одну для отслеживания результата и одну для подсчета того, как часто результат умножался на 2. Цикл проверяет, достигла ли вторая привязка числа 10, и, если нет, обновляет обе привязки.

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

Счетчик также мог бы начинаться с 1 и проверяться на `<= 10`, но по причинам, которые станут ясными в главе 4, лучше привыкать к отсчету от 0.

Цикл `do` — это управляющая структура, похожая на цикл `while`. Он отличается только одним моментом: цикл `do` всегда выполняет свое тело хотя бы один раз и проверяет, нужно ли остановиться, только после первого выполнения. Результат того, что проверка выполняется после тела цикла, виден в следующем примере.

```
let yourName;
do {
  yourName = prompt("Кто вы?");
} while (!yourName);
console.log(yourName);
```

Эта программа предлагает вам ввести имя. Она будет спрашивать имя снова и снова, пока не получит что-то, что не является пустой строкой. Применение оператора ! преобразует значение в логический тип перед его отрицанием, а все строки, кроме "", преобразуются в `true`. Это означает, что цикл будет выполняться, пока вы не введете непустое имя.

Код с отступами

В примерах я добавляю пробелы перед инструкциями, которые являются частью более крупной инструкции. Эти пробелы необязательны — компьютер отлично примет программу и без них. Более того, даже разрывы строк в программах необязательны. При желании вы можете написать программу в виде одной длинной строки.

Роль таких отступов внутри блоков состоит в том, чтобы выделить структуру кода. Если в коде внутри одних блоков открываются другие блоки, может быть трудно понять, где заканчивается один блок и начинается другой. С правильными отступами визуальная форма программы соответствует форме блоков внутри нее. Мне нравится использовать два пробела для каждого нового блока, но у каждого свои вкусы — некоторые ставят четыре пробела, другие задействуют символы табуляции. Главное, чтобы у каждого нового блока были отступы одинакового размера.

```
if (false != true) {
  console.log("Это имеет смысл.");
  if (1 < 2) {
    console.log("Ничего удивительного.");
  }
}
```

Большинство программ — редакторов кода помогают автоматически создавать для новых строк отступы правильного размера.

Циклы for

Многие циклы следуют шаблону, показанному в примерах с `while`: сначала создается привязка-счетчик для контроля за выполнением цикла. Затем идет цикл `while`, обычно с проверочным выражением, в котором проверяется, достиг ли счетчик конечного значения. В конце тела цикла счетчик изменяется, чтобы отслеживать состояние цикла.

Из-за распространенности этого шаблона в JavaScript и аналогичных языках есть несколько более короткая и общая форма цикла — `for`.

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
// ... и так далее
```

Эта программа в точности эквивалентна предыдущему примеру с печатью четных чисел. Единственное изменение заключается в том, что все инструкции, связанные с состоянием цикла, сгруппированы после ключевого слова `for`.

В скобках после `for` должны содержаться две точки с запятой. Перед первой выполняется *инициализация* цикла, обычно путем определения привязки. Вторая часть — это выражение, которое *проверяет*, следует ли продолжить выполнение цикла. В последней части *обновляется* состояние цикла после каждой итерации. В большинстве случаев эта запись короче и понятнее, чем конструкция `while`.

Вот код, в котором вычисляется 2^{10} с использованием `for` вместо `while`:

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

Принудительный выход из цикла

Ситуация, когда условие продолжения цикла равно `false`, не единственный вариант завершения цикла. Существует специальная инструкция, называемая `break`, которая позволяет немедленно «выпрыгнуть» из замкнутого цикла.

В следующей программе показано, как работает инструкция `break`. Программа находит первое число, которое больше или равно 20 и при этом делится на 7.

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
```

```
    break;
  }
}
// → 21
```

Использование оператора остатка от деления (`%`) — простой способ проверить, делится ли данное число на другое число. Если это так, то остаток от их деления равен нулю.

Конструкция `for` в указанном примере не имеет части, в которой проверяется условие выхода из цикла. Это означает, что цикл никогда не закончится, если только не выполнится расположенная внутри инструкция `break`.

Если удалить оператор `break` или случайно написать условие завершения, которое всегда равно `true`, то программа застрянет в *бесконечном цикле*. Такая программа никогда не закончит работу, что обычно имеет плохие последствия.

Ключевое слово `continue` похоже на `break` в том смысле, что оно тоже влияет на выполнение цикла. Когда в теле цикла встречается слово `continue`, управление «выпрыгивает» из него и выполнение цикла продолжается со следующей итерации.

Быстрое обновление привязок

В программах, особенно в циклах, часто требуется обновить привязку, присвоив ей новое значение, основанное на предыдущем значении этой же привязки.

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

В JavaScript есть для этого короткая запись:

```
counter += 1;
```

Аналогичные сокращения существуют и для других операторов — например, `result *= 2` для удвоения `result` или `counter -= 1` для уменьшения на единицу.

Это позволяет еще немного сократить пример со счетчиком.

```
for (let number = 0; number <= 12; number += 2) {
  console.log(number);
}
```

Для записей типа `counter += 1` и `counter -= 1` есть еще более короткие эквиваленты: `counter++` и `counter--`.

Диспетчеризация по значению с помощью switch

Часто бывает, что код выглядит так:

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

Существует конструкция, называемая `switch`, которая предназначена для выражения такой диспетчеризации более непосредственным способом. К сожалению, синтаксис, который используется для этого в JavaScript (и унаследован от семейства языков программирования C/Java), немного неуклюжий — иногда цепочка операторов `if` выглядит лучше. Вот пример:

```
switch (prompt("Какая сейчас погода?")) {
  case "дождь":
    console.log("Не забудьте взять зонтик.");
    break;
  case "солнечно":
    console.log("Одевайтесь легко.");
  case "облачно":
    console.log("Пойдите погуляйте.");
    break;
  default:
    console.log("Неизвестная погода!");
    break;
}
```

Внутри блока `switch` можно поместить любое количество вариантов `case`. Программа начнет выполнение с той метки, которая соответствует значению, заданному в `switch`, или с `default`, если подходящего значения не найдено. Программа будет выполнять все инструкции подряд, даже «сквозь» другие метки, пока не достигнет инструкции `break`. В некоторых случаях, таких как вариант `"солнечно"` в нашем примере, это можно использовать, чтобы один и тот же код выполнялся для нескольких вариантов (рекомендуется сходить погулять как в солнечную, так и в облачную погоду). Но будьте осторожны — об отсутствии `break` легко забыть, и тогда программа будет выполнять код, который не должен выполняться.

Использование прописных букв

В именах привязок не должно быть пробелов, однако часто бывает полезно использовать в них несколько слов, чтобы четко описать, что представляет данная привязка. Это, в сущности, ваш выбор — как писать имя привязки, состоящее из нескольких слов:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

Первый вариант может быть трудночитаемым. Мне больше нравится вариант с подчеркиваниями, хотя такой текст не очень удобно набирать. Стандартные функции JavaScript и большинство программистов на JavaScript используют стиль, показанный в нижней строке, — пишут с заглавной буквы каждое слово, кроме первого. К таким мелочам нетрудно привыкнуть, а код со смешанными стилями именования может быть неудобно читать, поэтому мы тоже будем следовать данному соглашению.

В некоторых случаях, таких как с функцией `Number`, первая буква привязки также является большой. Это было сделано для того, чтобы пометить указанную функцию как конструктор. Что такое конструктор, вы узнаете в главе 6. Пока что не стоит беспокоиться о таком явном отсутствии согласованности.

Комментарии

Часто сам код не передает всю информацию, которую хотелось бы передавать посредством программы читателям-людям, или же передает ее настолько загадочно, что не все могут ее понять. В других случаях просто хочется включить в программу некоторые сопутствующие мысли. Именно для этого нужны *комментарии*.

Комментарий представляет собой текст, являющийся частью программы, но полностью игнорируемый компьютером. В JavaScript есть два способа написания комментариев. Для того чтобы написать однострочный комментарий, можно использовать два символа косой черты (`//`), после которых пишется сам текст комментария.

```
let accountBalance = calculateBalance(account);
// Зеленая дыра; цепляясь бездумно,
```

```
accountBalance.adjust();
// За серебро травы, на дне река журчит,
let report = new Report();
// И солнце юное с вершины недоступной
addToReport(accountBalance, report);
// Горит. То тихий лог, где пенятся лучи.
// (Артюр Рембо. Уснувший в ложбине. Пер. Г. Петникова)
```

Комментарий `//` действует только до конца строки. Если же заключить текст между `/*` и `*/`, то он будет полностью игнорироваться независимо от того, содержит ли он разрывы строк. Это полезно для добавления блоков информации о файле или части программы.

```
/*
  Я впервые обнаружил, что это число написано на обороте старой записной
  книжки. С тех пор оно мне часто попадалось среди номеров телефонов
  и штрихкодов продуктов, которые я покупал. Похоже, я ему понравился,
  поэтому я решил его оставить.
*/
const myNumber = 11213;
```

Резюме

Теперь вы знаете, что программа состоит из инструкций, которые сами по себе тоже иногда состоят из инструкций. Инструкции, как правило, содержат выражения, которые сами по себе тоже могут состоять из меньших выражений.

Инструкции, размещаемые последовательно, одна за другой, образуют программу, выполняемую сверху вниз. Эту последовательность выполнения можно изменять, используя условные (`if`, `else` и `switch`) и циклические (`while`, `do` и `for`) инструкции.

Для хранения именованных фрагментов данных применяются привязки, они полезны для отслеживания состояния программы. Множество определенных привязок называется окружением. Любая система JavaScript всегда помещает в окружение ряд полезных стандартных привязок.

Функции — это специальные значения, в которых инкапсулирована часть программы. Чтобы вызвать функцию, нужно написать `functionName(argument1, argument2)`. Такой вызов функции является выражением и может возвращать значение.

Упражнения

Если вы не знаете, как проверить свои решения упражнений, обратитесь к разделу «Введение».

Каждое упражнение начинается с описания задачи. Прочитайте это описание и попробуйте выполнить упражнение. Если у вас возникнут проблемы, почитайте подсказки в конце книги. Полные решения упражнений не включены в данную книгу, но вы найдете их в Интернете по адресу https://eloquentjavascript.net/code. Если вы хотите чему-то научиться на этих упражнениях, я рекомендую смотреть решения только после выполнения упражнения или по крайней мере после того, как в результате долгих и настойчивых попыток его решить у вас не возникнет легкая головная боль.

Построение треугольника в цикле

Напишите цикл, который делает семь вызовов `console.log` и выводит следующий треугольник:

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

Возможно, вам будет полезно узнать: чтобы получить длину строки, нужно написать после нее `.length`.

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

FizzBuzz

Напишите программу, в которой с помощью `console.log` выводятся все числа от 1 до 100 с двумя исключениями. Для чисел, кратных 3, вместо числа выводится `"Fizz"`, а для чисел, кратных 5 (но не 3), — `"Buzz"`.

Когда это заработает, измените программу так, чтобы она печатала `"FizzBuzz"` для чисел, которые делятся и на 3, и на 5 (и по-прежнему печатайте `"Fizz"` или `"Buzz"` для чисел, кратных только одному из них).

На самом деле такой вопрос задают на собеседованиях и, по утверждениям, отсеивают на нем значительную долю кандидатов в программисты. Поэтому если вы решили эту задачу, то стоимость вашего труда резко возросла.

Шахматная доска

Напишите программу, которая создает строку, представляющую сетку 8 × 8, используя для разделения строк символы новой строки. В каждой позиции сетки стоит либо пробел, либо символ `"#"`. Эти символы должны располагаться в шахматном порядке.

Передавая данную строку в `console.log`, вы должны получить что-то вроде этого:

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

Если вы уже написали программу, которая генерирует этот узор, определите привязку `size = 8` и измените программу так, чтобы она работала для любого `size`, выводя сетку заданных ширины и высоты.

3 Функции

> Люди считают, что компьютерные науки — это искусство для гениев. В реальности все наоборот — просто множество людей делают вещи, одну на основе другой, будто строят стену из маленьких камушков.
>
> *Дональд Кнут*

Функции — это хлеб с маслом программирования на JavaScript. Концепция представления части программы в виде значения имеет множество применений. Она позволяет нам структурировать большие программы, уменьшить количество повторений, присваивать имена подпрограммам и изолировать эти подпрограммы одну от другой.

Наиболее очевидное применение функций — определение нового словаря. Создавать новые слова в прозе — обычно признак дурного стиля. Но в программировании это необходимо.

Словарный запас взрослого человека составляет в среднем около 20 000 слов. Редкий язык программирования насчитывает 20 000 встроенных команд. Но их доступный словарный запас, как правило, точнее определен и, следовательно, менее гибок, чем в человеческом языке. Поэтому мы обычно *вынуждены* вводить новые концепции, чтобы не повторяться слишком часто.

Определение функции

Определение функции — это обычная привязка, значением которой является функция. Например, в следующем коде определена привязка `square`, ссылающаяся на функцию, которая вычисляет квадрат заданного числа:

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

console.log(square(12));
// → 144
```

Функция состоит из выражения, начинающегося с ключевого слова `function`. Каждая функция имеет набор *параметров* (в данном случае только `x`) и тело, состоящее из инструкций, которые должны выполняться при вызове функции. Тело функции, созданной таким образом, всегда должно быть заключено в фигурные скобки, даже если оно состоит только из одной инструкции.

Функция может иметь несколько параметров или же не иметь их вообще. В следующем примере функция `makeNoise` не имеет параметров, тогда как у `power` их два:

```
const makeNoise = function() {
  console.log("Эгегей!");
};

makeNoise();
// → Эгегей!

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

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

Отдельные функции, такие как `power` и `square`, выдают значение, а другие, такие как `makeNoise`, — нет; их единственным результатом является побочный эффект. Значение, которое возвращает функция, определяет инструкция `return`. Когда управление программой доходит до этой инструкции, оно сразу выходит из текущей функции и передает возвращенное значение в код, ее вызвавший. Ключевое слово `return` без следующего за ним выражения приводит к тому, что функция возвращает `undefined`. Функции, которые вообще не имеют инструкции `return`, такие как `makeNoise`, также возвращают `undefined`.

Параметры, передаваемые функции, ведут себя как обычные привязки, но их начальные значения задаются *вызывающим объектом*, а не кодом самой функции.

Привязки и области видимости

Каждая привязка имеет *область видимости* — это часть программы, в которой видна данная привязка. Для привязок, определенных вне какой-либо функции или блока, область видимости — вся программа, и вы можете ссылаться на такие привязки где хотите. Они называются *глобальными*.

Но на привязки, созданные как параметры функции или объявленные внутри нее, можно ссылаться только в данной функции, поэтому они называются *локальными* привязками. Каждый раз, когда вызывается функция, создаются новые экземпляры этих привязок. Так обеспечивается некоторая изоляция между функциями — каждый их вызов действует в собственном маленьком мире (в локальной среде), и часто его легко понять, практически ничего не зная о том, что происходит в глобальной среде.

Привязки, объявленные с помощью ключевых слов `let` и `const`, на самом деле являются локальными для *блока*, внутри которого они объявлены. Поэтому если вы создадите одну из таких привязок внутри цикла, то код до и после данного цикла не сможет его «увидеть». До 2015 года в JavaScript только функции могли создавать новые области видимости, поэтому привязки, созданные в старом стиле с помощью ключевого слова `var`, видны во всей функции, в которой они появились, либо во всей глобальной области видимости, если они не входят в функцию.

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// привязка y здесь не видна
console.log(x + z);
// → 40
```

Каждая область видимости может «выглядывать» в область, которая ее окружает, поэтому в данном примере привязка `x` видна внутри блока.

Исключение составляют случаи, когда несколько привязок имеют одно и то же имя — тогда код может видеть только самую внутреннюю из привязок. Например, когда код внутри функции `halve` ссылается на `n`, то он видит собственную привязку `n`, а не глобальную `n`.

```
const halve = function(n) {
  return n / 2;
};

let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

Вложенные области видимости

JavaScript различает не только *глобальные* и *локальные* привязки. Внутри блоков и функций можно создавать другие блоки и функции, образуя несколько степеней локальности.

Например, внутри этой функции, которая выводит ингредиенты, необходимые для приготовления порции хумуса, есть еще одна функция:

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "банка", "нута");
  ingredient(0.25, "стакан", "кунжутной пасты");
  ingredient(0.25, "стакан", "лимонного сока");
  ingredient(1, "зубок", "чеснока");
  ingredient(2, "столовые ложки", "оливкового масла");
  ingredient(0.5, "чайной ложки", "кумина");
};
```

Код, размещенный внутри функции `ingredient`, видит привязку `factor` из внешней функции. Но ее локальные привязки, такие как `unit` и `componentAmount`, во внешней функции не видны.

Множество привязок, видимых внутри блока, определяется его положением в тексте программы. Каждая локальная область видимости также видит все локальные области, которые ее содержат, и все области видят глобальную область. Такой подход к видимости привязок называется *лексической областью видимости*.

Функции как значения

Привязка функции — это обычно просто имя для определенной части программы. Такая привязка определяется один раз и никогда не изменяется, в связи с чем можно легко перепутать функцию и ее имя.

Но это разные вещи. Значение функции может делать все то же, что и другие значения, — его можно не только вызывать, но и использовать в произвольных выражениях. Можно сохранить значение функции в новой привязке, передать ее в качестве аргумента другой функции и т. д. Аналогично привязка, которая содержит функцию, тем не менее представляет собой обычную привязку и может, если только не является константой, получить новое значение, например, так:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* ничего не делаем */};
}
```

В главе 5 мы рассмотрим интересные вещи, которые можно сделать, передавая значения функций другим функциям.

Декларативная запись

Существует несколько более короткий способ создания привязки для функции. Если в начале инструкции поставить ключевое слово `function`, то это будет работать немного по-другому.

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

Это *объявление* функции. Инструкция определяет привязку `square` и назначает ей ссылку на данную функцию. Такую инструкцию немного проще писать и можно не ставить после функции точку с запятой.

У подобной формы определения функции есть один нюанс.

```
console.log("Голос из будущего:", future());

function future() {
  return "Летающих машин не будет";
}
```

Этот код работает, несмотря на то что функция определена *после* кода, в котором она применяется. Объявления функций не являются частью обычной последовательности выполнения программы сверху вниз. Они концептуально перемещаются в верхнюю часть своей области видимости и могут использоваться любой частью кода в пределах данной области. Это иногда полезно, поскольку открывает возможность упорядочить код таким образом, который кажется вам более осмысленным, не беспокоясь о необходимости определить все функции прежде, чем они будут использованы.

Стрелочные функции

Существует третий способ записи для функций, и он очень отличается от остальных. Вместо ключевого слова `function` используется стрелка (`=>`), состоящая из знака равенства и символа «больше» (не путать с оператором «больше или равно», который пишется как `>=`).

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

Стрелка ставится *после* списка параметров, затем идет тело функции. Стрелка выражает что-то вроде «эти входные данные (параметры) дают этот результат (тело функции)».

Если имя параметра только одно, можно опустить круглые скобки, в которые заключен список параметров. Если тело функции представляет собой

единственное выражение, а не блок в фигурных скобках, то функция будет возвращать это выражение. Таким образом, следующие два определения `square` делают одно и то же:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

Если у стрелочной функции вообще нет параметров, то ее список параметров представляет собой просто пустые скобки.

```
const horn = () => {
  console.log("Toot");
};
```

Особой причины, по которой в языке было бы необходимо иметь и стрелочные функции, и выражения `function`, нет. За исключением незначительных деталей, которые мы обсудим в главе 6, они делают одно и то же. Стрелочные функции появились в 2015 году главным образом для того, чтобы стало возможным писать небольшие функциональные выражения менее многословным способом. Мы будем их часто использовать в главе 5.

Стек вызовов

Отдельный интерес представляет собой передача управления в процессе выполнения программы с функциями. Давайте внимательнее присмотримся к этому. Вот простая программа с несколькими вызовами функций:

```
function greet(who) {
  console.log("Привет, " + who);
}
greet("Гарри");
console.log("Пока!");
```

Выполнение этой программы происходит примерно так: при вызове функции `greet` управление переходит к началу данной функции (строка 2). Она, в свою очередь, вызывает функцию `console.log`, которая берет управление на себя, выполняет свою работу, а затем возвращает управление в строку 2. Там функция `greet` заканчивается, поэтому управление возвращается к месту, откуда она была вызвана, — к строке 4. В следующей строке снова вызывается `console.log`, после чего программа заканчивается.

Схематически передачу управления можно было бы представить так:

```
за пределами функции
  в функции greet
              в функции console.log
  в функции greet
за пределами функции
  в функции console.log
за пределами функции
```

Поскольку после выполнения функция должна вернуться в ту точку, откуда была вызвана эта функция, компьютер должен запомнить контекст, из которого произошел вызов. В первом случае функция `console.log` после выполнения должна вернуться в функцию `greet`; во втором — она возвращается в конец программы.

Место, где компьютер хранит этот контекст, является *стеком вызовов*. При каждом вызове функции текущий контекст сохраняется вверху данного стека. Когда функция возвращает управление, она удаляет из стека верхний контекст и использует его для продолжения выполнения.

Хранение этого стека требует места в компьютерной памяти. Если стек становится слишком большим, компьютер может дать сбой и вывести сообщение типа «недостаточно места в стеке» или «слишком глубокая рекурсия». В следующем коде проиллюстрирована подобная ситуация: компьютеру задан слишком сложный вопрос, который вызывает бесконечное переключение между двумя функциями. Точнее, это переключение *было бы* бесконечным, если бы у компьютера был бесконечный стек. Но поскольку он не бесконечен, мы исчерпаем допустимую память или, другими словами, «взорвем» стек.

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

Необязательные аргументы

Следующий код является допустимым и выполняется без проблем:

```
function square(x) { return x * x; }
console.log(square(4, true, "ежик"));
// → 16
```

Мы определили функцию `square` только с одним параметром. Тем не менее, когда мы вызываем ее с тремя аргументами, язык не жалуется. Он вычисляет квадрат первого из них и игнорирует остальные.

JavaScript чрезвычайно терпимо относится к количеству аргументов, которые вы передаете функции. Если передать их слишком много, то лишние игнорируются. Если передать слишком мало, то отсутствующим параметрам будет присвоено значение `undefined`.

Недостатком такого подхода является то, что возможно — даже вероятно, — что неверное количество аргументов было передано функции случайно. И никто не скажет вам об этом.

Положительный момент в том, что такое поведение можно использовать для вызова функции с разным количеством аргументов. Например, следующая функция `minus` имитирует оператор `-`, воздействуя на один или два аргумента:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

Если после параметра поставить оператор `=`, а затем выражение, то значение этого выражения будет заменять аргумент, если он не задан.

Например, в следующей версии функции `power` второй аргумент является необязательным. Если не предоставить его или передать значение `undefined`, то по умолчанию этот аргумент будет равен двум и функция будет вести себя как `square`.

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

console.log(power(4));
```

```
// → 16
console.log(power(2, 6));
// → 64
```

В следующей главе мы познакомимся со способом, позволяющим передать в тело функции весь список переданных аргументов. Это полезно, поскольку позволяет функции принимать любое количество аргументов. Например, именно так сделано в `console.log` — эта функция выводит все переданные ей значения.

```
console.log("C", "O", 2);
// → C O 2
```

Замыкание

Возможность рассматривать функции как значения в сочетании с тем фактом, что локальные привязки создаются заново при каждом вызове функции, вызывает интересный вопрос. Что происходит с локальными привязками, когда создавший их вызов функции больше не активен?

Пример такой ситуации показан в следующем коде. Здесь определена функция `wrapValue`, создающая локальную привязку. Затем данная функция возвращает другую функцию, которая обращается к этой локальной привязке и возвращает ее.

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

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

Это допускается и работает так, как мы и рассчитывали, — оба экземпляра привязки доступны. Такая ситуация является хорошей демонстрацией того, что локальные привязки создаются заново для каждого вызова, и один вызов не может уничтожить локальные привязки другого.

Такое свойство — возможность ссылаться на конкретный экземпляр локальной привязки в пределах замкнутой области видимости — называется

замыканием. Функция, которая ссылается на привязки из окружающих ее локальных областей видимости, называется замыканием. Такое поведение не только избавляет нас от необходимости беспокоиться о времени жизни привязок, но также позволяет творчески использовать значения функций.

Немного изменив предыдущий пример, мы можем превратить его в функцию, которая умножает число на произвольную величину.

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

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

Явная привязка `local` из примера с `wrapValue` на самом деле не нужна, поскольку сам параметр является локальной привязкой.

Чтобы понимать такие программы, нужно немного практики. Хорошая ментальная модель — представлять, что функции — это значения, содержащие и тело с кодом, и окружение, в котором они созданы. При вызове тело функции видит не то окружение, где вызвана функция, а то, где она была создана.

В рассмотренном примере вызванная функция `multiplier` создает окружение, где ее параметр `factor` привязан к значению 2. Возвращаемое значение функции, которое сохраняется в `twice`, запоминает это окружение. Поэтому, когда данное значение вызывается, оно умножает свой аргумент на 2.

Рекурсия

Для функции совершенно нормально вызывать саму себя, главное — не делать это слишком часто, чтобы не переполнить стек. Функция, вызывающая себя, называется *рекурсивной*. Рекурсия позволяет писать некоторые функции по-другому. Возьмем, к примеру, такую альтернативную реализацию функции `power`:

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

```
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

Это очень похоже на то, как математики определяют возведение в степень, и, возможно, описывает концепцию более ясно, чем вариант с циклом. Чтобы получить многократное умножение, функция несколько раз вызывает сама себя, каждый раз с меньшим показателем степени.

Но у подобной реализации есть одна проблема: в типичных реализациях JavaScript она работает примерно в три раза медленнее, чем версия с циклом. Выполнение простого цикла, как правило, менее затратно, чем многократный вызов функции.

Что выбрать: скорость или элегантность кода? Интересный вопрос. Это можно рассматривать как своего рода непрерывную линию между человекоориентированностью и машиноориентированностью. Практически любую программу можно заставить работать быстрее, сделав ее более длинной и запутанной. Программисту каждый раз приходится выбирать подходящий баланс.

В случае с функцией `power` неэлегантная версия (с циклом) все же довольно проста и понятна, так что нет смысла заменять ее рекурсивной версией. Но часто программа оперирует такими сложными концепциями, что полезно отказаться от некоторой доли эффективности, чтобы сделать программу более простой.

Беспокойство по поводу эффективности способно довести до отчаяния. Это еще один фактор, который усложняет разработку программы: когда она и без того достаточно сложна, дополнительная забота способна окончательно ввести в ступор.

Поэтому всегда начинайте с написания чего-то понятного, что корректно работает. Если вас беспокоит слишком медленная работа — что вряд ли, поскольку большая часть кода просто не выполняется достаточно часто, чтобы занять какое-то значительное время, — позже вы всегда сможете измерить скорость и при необходимости улучшить код.

Рекурсия не всегда является неэффективной альтернативой циклу. Некоторые проблемы действительно проще решить с помощью рекурсии, чем

с помощью цикла. Чаще всего это проблемы, требующие изучения или обработки нескольких ветвей кода, каждая из которых может, в свою очередь, разделяться на еще большее количество ветвей.

Рассмотрим такую задачу: если, начиная с числа 1, каждый раз либо прибавлять к исходному числу 5, либо умножать его на 3, можно получить бесконечный набор чисел. Как бы вы написали функцию, пытающуюся найти последовательность таких сложений и умножений, в результате которых получилось бы заданное число?

Например, число 13 можно получить, если сначала умножить 1 на 3, а затем дважды прибавить 5, тогда как число 15 вообще нельзя так получить.

Вот рекурсивное решение этой задачи:

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

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

Обратите внимание, что эта программа не обязательно находит *самую короткую* последовательность операций. Она успешно завершается, если вообще находит какую-либо подходящую последовательность.

Если вам пока непонятно, как это работает, — ничего страшного. Давайте разберемся, поскольку это важное упражнение на рекурсивное мышление.

Настоящую рекурсию выполняет внутренняя функция `find`. Она принимает два аргумента: текущее число и строку, в которой записано, как число было получено. Если функция находит решение, она возвращает эту строку, где показано, как можно достичь заданной цели. Если решение для указанного числа не найдено, то возвращается `null`.

Для этого функция выполняет одно из трех действий. Если текущее число `current` является заданным числом `target`, то текущее значение `history` представляет собой способ достижения цели, поэтому оно и возвращается. Если текущее число больше заданного, то дальнейшее исследование этой ветви не имеет смысла: и сложение, и умножение только увеличат текущее число, так что программа возвращает `null`. Наконец, если мы все еще не достигли заданного числа, функция проверяет оба возможных пути, которые начинаются с текущего значения, дважды вызывая саму себя, — один раз для сложения и один раз для умножения. Если первый вызов возвращает что-то, не равное `null`, то возвращается это значение. В противном случае возвращается результат второго вызова, независимо от того, сгенерировал он строку или `null`.

Чтобы лучше понять, как эта функция выдает нужный результат, рассмотрим все вызовы, которые нужно выполнить при поиске 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!
```

Отступы отражают глубину стека вызовов. При первом вызове функция `find` начинает с вызова самой себя, чтобы исследовать решение, которое начинается с `(1 + 5)`. Затем этот вызов будет с помощью рекурсии исследовать *каждое* продолжение указанного решения, выдающее число, которое меньше заданного или равно ему. Поскольку варианта, когда получается заданное число, найдено не будет, то этот вызов вернет `null`. Там оператор `||` вызовет второй вариант `find`, который исследует `(1 * 3)`. Этот поиск окажется более удачным — его первый рекурсивный вызов посредством *еще одного* рекурсивного вызова попадет в цель. Внутренний вызов возвращает строку, и каждый из операторов `||` в промежуточных вызовах также возвращает эту строку, которая в итоге и возвращается как решение.

Разрастание функций

Существует два более или менее естественных способа ввести функцию в программу.

Первый — когда обнаруживается, что вы несколько раз написали весьма похожий код. Вы бы предпочли такого не делать. Чем больше кода — тем больше места для скрытых ошибок и больше строк придется прочитать людям, пытающимся понять программу. Поэтому вы берете повторяющуюся функциональность, присваиваете ей подходящее имя и помещаете ее в функцию.

Второй способ заключается в том, что обнаруживается: вам нужен некий функционал, который вы еще не написали, и он, похоже, заслуживает собственной функции. Вы начинаете с ее названия, а затем пишете ее тело. Вы можете даже сначала написать код, использующий эту функцию, и только потом фактически определить ее саму.

То, насколько легко удается подобрать удачное имя функции, — хороший показатель того, насколько вам ясна концепция, которую вы пытаетесь описать. Рассмотрим это на примере.

Мы хотим написать программу, печатающую два числа: число коров и цыплят на ферме, после которых идут слова `коров` и `цыплят`; перед обоими числами должны ставиться нули таким образом, чтобы данные числа всегда были длиной три цифры.

```
007 коров
011 цыплят
```

Это требует функции с двумя аргументами — количеством коров и количеством цыплят. Давайте закодируем.

```
function printFarmInventory(cows, chickens) {
  let cowString = String(cows);
  while (cowString.length < 3) {
    cowString = "0" + cowString;
  }
  console.log(`${cowString} коров`);
  let chickenString = String(chickens);
  while (chickenString.length < 3) {
    chickenString = "0" + chickenString;
  }
  console.log(`${chickenString} цыплят`);
}
printFarmInventory(7, 11);
```

Если написать `.length` после строкового выражения, получим длину этой строки. Таким образом, в цикле `while` перед числовой строкой добавляются нули до тех пор, пока длина строки не станет равной как минимум трем символам.

Миссия выполнена! Мы уже собрались отправить код (вместе с солидным счетом) хозяйке фермы, но тут она звонит нам и говорит, что завела еще и свиней, и не могли бы мы, пожалуйста, дополнить программу так, чтобы она печатала еще и число свиней?

Конечно, можем. Но, копируя и вставляя эти четыре строки кода еще раз, мы остановились и задумались: должен быть способ получше. Вот наша первая попытка:

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

function printFarmInventory(cows, chickens, pigs) {
  printZeroPaddedWithLabel(cows, "коров");
  printZeroPaddedWithLabel(chickens, "цыплят");
  printZeroPaddedWithLabel(pigs, "свиней");
}

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

Это работает! Но имя `printZeroPaddedWithLabel` выглядит несколько странно. Здесь в одну функцию объединены три процесса: печать, заполнение нулями и добавление надписи.

Вместо того чтобы выделять всю повторяющуюся часть программы одним куском, попробуем представить в виде функции отдельную *концепцию*.

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

```
function printFarmInventory(cows, chickens, pigs) {
  console.log(`${zeroPad(cows, 3)} коров`);
  console.log(`${zeroPad(chickens, 3)} цыплят`);
  console.log(`${zeroPad(pigs, 3)} свиней`);
}

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

Встретив в коде функцию с красивым очевидным именем, таким как `zeroPad`, любой, кто читает код, легко поймет, что она делает. И потом, такая функция полезна не только в этой конкретной программе, но и в других ситуациях. Например, ее можно использовать, чтобы печатать красивые, ровные таблицы чисел.

Насколько умной и универсальной *должна* быть функция? Мы могли бы написать что угодно, от ужасно простой функции, которая бы дополняла число нулями до трех символов, до сложной обобщенной системы форматирования чисел, учитывающей дробные и отрицательные числа, делающей выравнивание по десятичной точке, заполнение разными символами и т. д.

Полезный принцип — не добавлять функционал, если вы не уверены, что он вам понадобится. Бывает заманчиво написать обобщенный «каркас» для каждой функциональности, которая вам встретилась. Не поддавайтесь этому желанию. Так вы не выполните реальную работу — вы просто будете писать код, который никогда не пригодится.

Функции и побочные эффекты

Функции можно условно разделить на те, которые вызываются ради выполнения побочных эффектов, и те, которые вызываются ради возвращаемого значения. (Хотя функция может делать и то и другое: и выполнять побочные эффекты, и возвращать значение.)

Первая вспомогательная функция в примере с фермой, `printZeroPaddedWithLabel`, вызывается ради побочного эффекта: она печатает строку. Вторая версия, `zeroPad`, вызывается ради возвращаемого значения. То, что вторая функция оказывается полезной в большем количестве ситуаций, чем первая, — не случайность. Функции, создающие значения, легче комбинировать разными способами, чем функции, непосредственно выполняющие побочные эффекты.

Чистая функция — это особый вид функции, возвращающей значение. Она не только не имеет побочных эффектов, но и не зависит от побочных эффектов, создаваемых в других частях кода, — например, она не читает глобальные привязки, значение которых может изменяться. Приятное свойство чистой функции заключается в том, что такая функция, вызываемая с одинаковыми аргументами, всегда возвращает одно и то же значение (и больше ничего не делает). Вызов такой функции может быть заменен ее возвращаемым значением, и остальной код не пострадает. Если вы не уверены, что чистая функция работает правильно, вы можете легко ее протестировать, просто вызвав ее и зная, что если она работает в данном контексте, то будет работать в любом. Для тестирования нечистых функций, как правило, требуется специальное окружение.

Тем не менее не стоит расстраиваться, если вам приходится писать функции, которые не являются чистыми, или вести священную войну за чистоту функций в вашем коде. Побочные эффекты часто бывают полезны. Например, не существует возможности написать чистую версию `console.log`, а `console.log` очень полезна. Некоторые операции также проще выразить, используя побочные эффекты, так что скорость вычислений может стать причиной написания нечистых функций.

Резюме

В этой главе мы научились писать собственные функции. Ключевое слово `function`, будучи использованным как выражение, позволяет создать значение функции. При применении в качестве инструкции оно позволяет объявить привязку и присвоить ей функцию в качестве значения. Еще одним способом создания функций являются стрелочные функции.

```
// Определяем f для хранения значения функции
const f = function(a) {
  console.log(a + 2);
};

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

// Более лаконичное описание функции
let h = a => a % 3;
```

Ключевым аспектом в понимании функций выступает понимание областей видимости. Каждый блок создает новую область видимости. Параметры и привязки, объявленные в данной области видимости, являются локальными и невидимы извне. Привязки, объявленные с помощью ключевого слова `var`, ведут себя иначе: они принадлежат области видимости ближайшей функции или глобальной области видимости.

Задачи, выполняемые программой, полезно разделять на функции. Тогда вам придется меньше повторяться; кроме того, функции позволяют упорядочить программу, сгруппировав код в виде фрагментов, выполняющих определенные задачи.

Упражнения

Минимум

В предыдущей главе была представлена стандартная функция `Math.min`, которая возвращает наименьший из ее аргументов. Теперь мы можем сами создать нечто подобное. Напишите функцию `min`, которая принимает два аргумента и возвращает их минимум.

Рекурсия

Как мы видели, оператор `%` (оператор остатка) можно применять для проверки, является число четным или нечетным. Для этого нужно использовать `%2`, чтобы узнать, делится ли оно на два. Вот еще один способ определить, является ли положительное целое число четным или нечетным:

- ноль четный;
- единица нечетная;
- четность любого другого числа N совпадает с четностью $N - 2$.

Определите рекурсивную функцию `isEven`, соответствующую этому описанию. Функция должна принимать один параметр (положительное целое число) и возвращать логическое значение.

Проверьте эту функцию на числах 50 и 75. Посмотрите, как она ведет себя для -1. Почему? Можете ли вы придумать способ, как это исправить?

Подсчет букв

Чтобы получить N-й символ или букву из строки, нужно написать `"string"[N]`. Возвращаемым значением будет строка, содержащая только один символ (например, `"b"`). Первый символ имеет позицию 0, в результате чего последний находится в позиции `string.length - 1`. Другими словами, длина строки из двух символов равна 2, а ее символы находятся в позициях 0 и 1.

Напишите функцию `countBs`, которая принимает строку в качестве единственного аргумента и возвращает число, показывающее, сколько больших букв «B» содержится в этой строке.

Затем напишите функцию `countChar`, которая ведет себя как `countBs`, за исключением того, что принимает второй аргумент, указывающий, какие именно символы нужно посчитать (вместо того чтобы считать только большие буквы «B»). Перепишите `countBs`, чтобы использовать эту новую функцию.

4 Структуры данных: объекты и массивы

> Два раза меня спрашивали: «Скажите, м-р Бэббидж, если вы введете в машину неправильные данные, получится ли правильный ответ?» [...] Какую же путаницу нужно иметь в голове, чтобы задавать подобные вопросы!
>
> *Чарльз Бэббидж. Отрывки из жизни философа (1864)*

Числа, логические значения и строки — это атомы, из которых строятся структуры данных. Однако для многих типов информации требуется более одного атома. *Объекты* позволяют нам группировать значения — включая другие объекты, — чтобы строить более сложные структуры.

Программы, которые мы создавали до сих пор, были ограничены тем, что оперировали только простыми типами данных. В этой главе будут представлены основные структуры данных. Дочитав ее до конца, вы будете знать достаточно, чтобы начать писать полезные программы.

В этой главе мы рассмотрим более или менее реалистичный пример программирования, добавляя в него новые концепции по мере их применения в решении проблемы. Код примера часто будет основываться на функциях и привязках, которые были представлены ранее в тексте.

Онлайн-«песочница» для программирования, прилагаемая к этой книге (https://eloquentjavascript.net/code), позволяет выполнять код из каждой главы. Если вы решите работать с примерами в другой среде, сначала загрузите полный код примера из этой главы со страницы, размещенной в «песочнице».

Белка-оборотень

Время от времени, как правило, между 8 и 10 часами вечера Жак превращается в маленького пушистого грызуна с густым хвостом.

С одной стороны, Жак очень рад, что у него не классическая ликантропия. Превращение в белку вызывает меньше проблем, чем в волка. Можно не переживать о том, чтобы случайно не съесть соседа (это было бы неловко). Но зато он беспокоится о том, что его может съесть соседский кот. Он уже дважды просыпался на верхушке дуба, на слишком тонкой ветке, обнаженный и сбитый с толку. Он пытался запирать на ночь двери и окна своей комнаты и клал на пол горсть грецких орехов, чтобы отвлечь самого себя.

Это решило проблемы с кошками и деревьями. Но Жак предпочел бы полностью избавиться от своего состояния. Нерегулярность трансформаций наводит его на мысль, что они могут быть вызваны чем-то. Некоторое время он считал, что такое происходит только в те дни, когда он находится рядом с дубами. Он попробовал держаться подальше от дубов, но это не решило проблему.

Выбрав более научный подход, Жак стал вести дневник, куда записывал, что он делал в данный день и изменял ли он тогда форму. По этим данным он надеется определить конкретные условия, которые приводят к трансформациям.

Главное, что ему нужно, — структура данных для хранения этой информации.

Наборы данных

Для того чтобы работать с порцией цифровых данных, сначала нужно найти способ представить их в памяти нашей машины. Например, предположим, что мы хотим представить множество чисел: 2, 3, 5, 7 и 11.

Мы могли бы творчески использовать строки — в конце концов, строка может иметь любую длину, поэтому в нее можно поместить много данных и представить их как `"2 3 5 7 11"`. Но это неудобно. Чтобы иметь доступ к данным, нам нужно как-то извлекать их оттуда и преобразовывать обратно в числа.

К счастью, в JavaScript существует специальный тип данных, предназначенный для хранения последовательностей значений. Он называется *массивом*

и записывается как список значений, разделенных запятыми, который заключен в квадратные скобки.

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

Для того чтобы извлечь элемент из массива, также используется запись с квадратными скобками. Выражение в квадратных скобках, поставленное сразу после другого выражения, приведет к тому, что из результата левого выражения будет извлечен элемент, *индекс* которого соответствует значению, заданному выражением в скобках.

Индекс первого элемента массива — ноль, а не единица. Таким образом, чтобы получить первый элемент, нужно написать `listOfNumbers[0]`. Отсчет с нуля имеет давнюю традицию в информационных технологиях и в определенном смысле весьма оправдан, но к этому нужно привыкнуть. Пока что представляйте себе индекс как количество элементов, которые нужно пропустить, считая от начала массива.

Свойства

В предыдущих главах нам встретилось несколько подозрительных на вид выражений, таких как `myString.length` (чтобы получить длину строки) и `Math.max` (функция поиска максимума). Это выражения, обращающиеся к *свойству* некоторого значения. В первом случае мы получаем доступ к свойству `length` значения `myString`, во втором — к свойству `max` объекта `Math` (который представляет собой множество математических констант и функций).

Почти все значения JavaScript имеют свойства. Исключениями являются `null` и `undefined`. Если вы попытаетесь получить доступ к свойству одного из этих «не-значений», то получите ошибку.

```
null.length;
// → TypeError: null has no properties
```

В JavaScript есть два основных способа получить доступ к свойствам: с помощью точки и квадратных скобок. И `value.x`, и `value[x]` обращаются к свойству значения `value`, но не обязательно к одному и тому же. Разница в том, как интерпретируется `x`. При использовании точки слово, стоящее после нее, является точным именем свойства. При использовании квадратных скобок для получения имени свойства *вычисляется* заключенное в скобки выражение. Если `value.x` извлекает из значения `value` свойство с именем `x`, то `value[x]` пытается вычислить выражение `x` и применяет результат, преобразованный в строку, в качестве имени свойства.

Поэтому если вы знаете, что интересующее вас свойство называется `color`, то сразу пишите `value.color`. Если же вы хотите извлечь свойство, имя которого есть значение, содержащееся в привязке `i`, то пишите `value[i]`. Имена свойств являются строками. Это могут быть любые строки, но точечная нотация работает только с именами, которые выглядят как допустимые имена привязок. Поэтому если вы хотите получить доступ к свойству с именем `2` или `John Doe`, то вы должны использовать квадратные скобки: `value[2]` или `value["John Doe"]`.

Элементы массива хранятся как свойства массива с числами в качестве имен. Поскольку мы не можем задействовать для чисел точечную нотацию и, как правило, используем привязку, в которой содержится индекс, то для получения элементов массива приходится применять скобочную нотацию.

Свойство массива `length` сообщает нам, сколько в данном массиве элементов. Такое имя свойства является допустимым именем привязки и известно заранее, так что с целью найти длину массива мы обычно пишем `array.length` — это проще, чем `array["length"]`.

Методы

Кроме свойства `length`, и строковые объекты, и объекты-массивы содержат ряд свойств, значениями которых являются функции.

```
let doh = "Эгегей";
console.log(typeof doh.toUpperCase);
// → функция
console.log(doh.toUpperCase());
// → ЭГЕГЕЙ
```

У любой строки есть свойство `toUpperCase`. При вызове оно возвращает копию строки, в которой все буквы преобразованы в прописные. Существует

также свойство `toLowerCase`, выполняющее, наоборот, преобразование в строчные буквы.

Любопытно, что, хотя при вызове `toUpperCase` не передаются никакие аргументы, функция каким-то образом получает доступ к строке `"Эгегей"` — значению, для которого мы ее вызвали. О том, как это работает, вы узнаете в главе 6.

Свойства, содержащие функции, обычно называют *методами* значения, к которому они принадлежат; так, `toUpperCase` — это метод строки.

В следующем примере показаны два метода, которые можно использовать для управления массивами:

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

Метод `push` добавляет значения в конец массива, а метод `pop`, наоборот, удаляет из массива последнее значение и возвращает его.

Эти несколько нелепые имена являются традиционными терминами для операций со *стеком*. Стек в программировании — структура данных, в которую можно помещать значения и извлекать их оттуда в обратном порядке, так что то, что попало в стек последним, будет удалено первым. Это часто встречается в программировании — вспомните хотя бы стек вызовов функций из предыдущей главы, который является примером той же идеи.

Объекты

Вернемся к белке-оборотню. Множество ежедневных записей из дневника может быть представлено в виде массива. Но запись — это не просто число или строка; каждая запись должна хранить список действий и логическое значение, которое показывает, превратился Жак в белку или нет. В идеале мы хотели бы сгруппировать их в единое значение, а затем поместить такие сгруппированные значения в массив дневниковых записей.

Значения типа *«объект»* представляют собой произвольные наборы свойств. Один из способов создать объект — использовать выражение в фигурных скобках.

```
let day1 = {
  squirrel: false,
  events: ["работал", "трогал дерево", "ел пиццу", "бегал"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

Внутри фигурных скобок находится список свойств, разделенных запятыми. Каждое свойство имеет имя, после которого стоят двоеточие и значение. Если записать объект в несколько строк с отступами, как в этом примере, то читать код будет гораздо удобнее. Имена свойств, не являющихся допустимыми именами привязок или действительными числами, нужно заключать в кавычки.

```
let descriptions = {
  work: "Пошел на работу",
  "трогал дерево": "Потрогал дерево"
};
```

Это означает, что скобки в JavaScript имеют *два* значения. Скобки, стоящие в начале инструкции, открывают блок операторов. В любой другой позиции скобки описывают объект. К счастью, инструкции редко начинаются с объекта в фигурных скобках, так что неоднозначность между этими двумя вариантами не является большой проблемой.

При попытке прочитать несуществующее свойство мы получим значение `undefined`.

Для того чтобы присвоить значение выражению свойства, можно воспользоваться оператором `=`. В результате значение свойства будет изменено, если оно уже существовало, или, если такого свойства еще не было, будет создано новое свойство объекта.

Возвращаясь к нашему представлению о привязках как о щупальцах, отмечу, что привязки свойств аналогичны. Они *захватывают* значения, но

другие привязки и свойства могут удерживать те же значения. Объект можно представить себе как осьминога с произвольным количеством щупалец, на каждом из которых вытатуировано имя.

Оператор `delete` отсекает щупальце такого осьминога. Это унарный оператор, который, будучи примененным к свойству объекта, удаляет из объекта данное именованное свойство. Такое встречается нечасто, но это возможно.

```
let 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`, будучи примененным к строке и объекту, сообщает, есть ли у данного объекта свойство с таким именем. Разница между присвоением свойству значения `undefined` и его фактическим удалением заключается в том, что в первом случае у объекта все еще *есть* такое свойство (оно просто не имеет сколько-нибудь интересного для нас значения), тогда как во втором случае свойство больше не существует и оператор `in` вернет `false`.

Для того чтобы узнать, какими свойствами обладает объект, можно использовать функцию `Object.keys`. Если передать ей объект, то она вернет массив строк — имена свойств объекта.

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

Также существует функция `Object.assign`, которая копирует все свойства из одного объекта в другой.

```
let objectA = {a: 1, b: 2};
Object.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

Таким образом, массивы являются просто своего рода объектами, специализированными для хранения последовательностей. Если ввести выражение `typeof []`, то его значением будет `"object"`. Массивы можно представить себе

в виде длинных плоских осьминогов, у которых все щупальца выстроились в аккуратный ряд и каждое помечено цифрой.

Дневник, который ведет Жак, мы можем представить как массив объектов.

```
let journal = [
  {events: ["работал", "трогал дерево", "ел пиццу",
            "бегал", "смотрел телевизор"],
   squirrel: false},
  {events: ["работал", "ел мороженое", "ел цветную капусту",
            "ел лазанью", "трогал дерево", "чистил зубы"],
   squirrel: false},
  {events: ["выходной", "катался на велосипеде", "отдыхал",
            "ел арахис", "пил пиво"],
   squirrel: true},
  /* и так далее... */
];
```

Изменяемость

Вот уже совсем скоро мы начнем программировать по-настоящему. Но прежде нужно пояснить еще немного теории.

Как мы видели, значения объекта можно изменять. Типы значений, которые обсуждались в предыдущих главах, такие как числа, строки и логические значения, являются *неизменяемыми* — значения этих типов невозможно изменить. Их можно комбинировать и получать новые значения, но если взять конкретное строковое значение, то оно всегда останется неизменным. Его текст не может быть изменен. Если у вас есть строка, содержащая слово `"кошка"`, другой код не сможет изменить в ней символ так, чтобы получилось `"мошка"`.

Объекты ведут себя иначе. Их свойства *можно* изменять, в результате чего одно значение объекта может иметь разное содержимое в разное время.

Если у нас есть два числа, 120 и 120, мы можем считать их совершенно одинаковыми, независимо от того, занимают ли они физически одни и те же биты. В случае объектов существует разница между двумя ссылками на один и тот же объект и двумя разными объектами, которые содержат одинаковые свойства. Рассмотрим следующий код:

```
let object1 = {value: 10};
let object2 = object1;
let 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
```

Привязки `object1` и `object2` захватывают *один и тот же* объект, поэтому при изменении `object1` значение `object2` также меняется. Говорят, что такие привязки имеют одинаковую *идентичность*. Привязка `object3` указывает на другой объект, который изначально содержит те же свойства, что и `object1`, но живет своей жизнью.

Привязки также могут быть изменяемыми или постоянными, но это отдельный вопрос, от которого не зависит поведение их значений. Даже если числовые значения не изменяются, можно использовать привязку `let`, чтобы отслеживать изменяющееся число, меняя значение, на которое указывает привязка. Аналогичным образом, хотя привязка `const` к объекту сама по себе не может быть изменена и будет продолжать указывать на тот же объект, содержимое этого объекта может измениться.

```
const score = {visitors: 0, home: 0};
// Так можно
score.visitors = 1;
// А так нельзя
score = {visitors: 1, home: 1};
```

При сравнении объектов с помощью оператора JavaScript `==` объекты сравниваются по идентичности: оператор вернет `true`, только если оба объекта имеют одно и то же значение. Результатом сравнения разных объектов будет `false`, даже если они имеют одинаковые свойства. В JavaScript нет встроенной операции «глубокого» сравнения, которая сравнивала бы объекты по содержимому, но такую операцию можно написать самостоятельно (и это одно из упражнений в конце данной главы).

Дневник оборотня

Итак, Жак запустил свой интерпретатор JavaScript и настроил окружение так, чтобы вести свой журнал.

```
let journal = [];
function addEntry(events, squirrel) {
  journal.push({events, squirrel});
}
```

Обратите внимание: объект, добавленный в журнал, выглядит странновато. Вместо того чтобы объявить свойства, такие как `events: events`, мы просто задаем в нем имена свойств. Это сокращение, которое означает то же самое: если после имени свойства в записи, заключенной в скобки, не следует значение, то оно берется из привязки с тем же именем.

Итак, каждый вечер в 10 часов — или иногда на следующее утро, — спустившись с верхней полки книжного шкафа, Жак записывает события прошедшего дня.

```
addEntry(["работал", "трогал дерево", "ел пиццу", "бегал",
          "смотрел телевизор"], false);
addEntry(["работал", "ел мороженое", "ел цветную капусту",
          "ел лазанью", "трогал дерево", "чистил зубы"], false);
addEntry(["выходной", "катался на велосипеде", "отдыхал",
          "ел арахис", "пил пиво"], true);
```

Как только у него соберется достаточно информации, он намерен использовать статистические методы, чтобы выяснить, какое из этих событий может быть связано с превращениями в белку.

Корреляция — это степень зависимости между статистическими переменными. Статистическая переменная — не совсем то же самое, что программная переменная. В статистике обычно существует множество *измерений*, и результатом каждого из них является переменная. Корреляция между переменными обычно выражается в виде значения в диапазоне от –1 до 1. Нулевая корреляция означает, что переменные не связаны. Корреляция, равная единице, указывает на то, что две переменные точно совпадают — если вы знаете одну из них, то знаете и другую. Минус единица также означает, что переменные однозначно связаны, но они противоположны — когда одна истинна, то вторая ложна.

Чтобы вычислить степень корреляции между двумя логическими переменными, можно использовать *фи-коэффициент* (φ). Это формула, вход-

ными данными которой является таблица частот, содержащая количество раз, когда наблюдались различные комбинации переменных. Результатом формулы будет число в диапазоне от −1 до 1, описывающее корреляцию.

Мы могли бы взять событие «ел пиццу» и поместить его в такую таблицу частот, где каждое число показывало бы, сколько раз эта комбинация встречалась в наших измерениях:

Если присвоить этой таблице имя n, то можно вычислить φ по следующей формуле:

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}}. \tag{4.1}$$

Если в этот момент вы собрались отложить книгу, так как вас накрыло ужасным воспоминанием об уроках математики в десятом классе, — подождите! Я не собираюсь мучить вас бесконечными страницами со страшными формулами — пока это всего одна формула. Но и ее мы лишь превратим в код на JavaScript.

Запись n_{01} обозначает количество измерений, в которых первая переменная (превращение) ложна (равна 0), а вторая (пицца) — истинна (равна 1). В таблице, посвященной поеданию пиццы, значение n_{01} равно 9.

Значение $n_{1\bullet}$ означает сумму всех измерений, где первая переменная является истинной, и в нашей таблице оно равно 5. Аналогично $n_{\bullet 0}$ соответствует сумме измерений, где вторая переменная будет ложной.

Таким образом, для таблицы про пиццу часть формулы, расположенная над дробной чертой (числитель), будет равна 1 × 76 − 4 × 9 = 40, а часть под ней (знаменатель) — квадратному корню из 5 × 85 × 10 × 80, то есть $\sqrt{340000}$.

Получается, что $\varphi \approx 0{,}069$, что очень мало. Похоже, что поедание пиццы никак не влияет на превращение в белку.

Вычисление корреляции

Таблицу два на два мы могли бы представить в JavaScript как массив из четырех элементов: `[76, 9, 4, 1]`. Мы также могли бы использовать и другие представления, такие как массив, состоящий из двух массивов по два элемента в каждом: `[[76, 9], [4, 1]]`, или как объект с именами свойств наподобие `"11"` и `"01"`. Но обычный массив проще, и выражения для доступа к таблице в нем радуют своей краткостью. Мы будем интерпретировать индексы массива как двухбитные двоичные числа, где крайняя левая (более значимая) цифра относится к переменной превращений в белку, а крайняя правая (менее значимая) цифра — к переменной события. Например, двоичное число `10` относится к случаю, когда Жак превратился в белку, но событие (в данном случае поедание пиццы) не произошло. Так случилось четыре раза. И поскольку двоичное число 10 соответствует 2 в десятичной записи, мы будем хранить это число в индексе 2 массива.

Вот функция, которая вычисляет φ-коэффициент для 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 прямой перевод φ-формулы на JavaScript. `Math.sqrt` — это функция извлечения квадратного корня, предоставляемая объектом `Math` в стандартной среде JavaScript. Чтобы получить поля типа $n_{1\bullet}$, нам нужно сложить два поля из таблицы, потому что в нашей структуре данных не хранятся готовые суммы строк и столбцов.

Жак вел свой дневник три месяца. Полученный в результате набор данных доступен в программной «песочнице» для главы (https://eloquentjavascript.net/code#4), где он хранится в привязке `JOURNAL` и в файле, выложенном онлайн.

Для того чтобы извлечь таблицу два на два для определенного события из этого журнала, нужно перебрать в цикле все записи и подсчитать, сколько раз данное событие происходило по отношению к превращениям в белку.

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

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

У массивов есть метод `include`, который проверяет, существует ли в массиве заданное значение. Функция использует этот метод, чтобы определить, входит ли имя интересующего вас события частью в список событий данного дня.

Тело цикла в функции `tableFor` определяет, в какую ячейку таблицы входит каждая запись журнала, проверяя, содержит ли запись интересующее вас событие и происходит ли это событие одновременно с превращением в белку. Затем цикл добавляет единицу в соответствующую ячейку таблицы.

Теперь у нас есть инструменты, необходимые для вычисления отдельных корреляций. Единственный оставшийся шаг — найти корреляцию для каждого типа события, которое было записано в дневнике, и посмотреть, выделяется ли что-нибудь на общем фоне.

Перебор массива в цикле

В функции `tableFor` есть такой цикл:

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
  // Сделать что-то с entry
}
```

Циклы этого типа широко распространены в классическом JavaScript — перебор массива по одному элементу встречается очень часто, для этого нужно лишь запустить счетчик по длине массива и перебрать все элементы по очереди.

Но в современном JavaScript есть более простой способ написания таких циклов.

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} событий.`);
}
```

Если цикл `for` выглядит подобным образом, со словом `of` после определения переменной, он будет перебирать все элементы значения, указанного после `of`. Это работает не только для массивов, но и для строк и для некоторых других структур данных. Подробнее о том, *как именно* это работает, мы поговорим в главе 6.

Окончательный анализ

Нам нужно вычислить корреляцию для каждого типа событий, присутствующего в наборе данных. Для этого сначала нужно *найти* все типы событий.

```
function journalEvents(journal) {
    let events = [];
    for (let entry of journal) {
      for (let event of entry.events) {
        if (!events.includes(event)) {
          events.push(event);
        }
      }
    }
    return events;
}

console.log(journalEvents(JOURNAL));
// → ["ел морковь", "делал зарядку", "выходной", "ел хлеб", ...]
```

Перебирая все события и добавляя в их массив те, которых там еще нет, функция собирает все типы событий.

Используя ее, мы можем найти все корреляции.

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → ел морковь:      0.0140970969
// → делал зарядку:   0.0685994341
// → выходной:        0.1371988681
// → ел хлеб:        -0.0757554019
// → ел пудинг:      -0.0648203724
// и так далее...
```

Похоже, большинство корреляций близки к нулю. Употребление в пищу моркови, хлеба или пудинга, по-видимому, не вызывает обострения беличьей ликантропии. *Кажется*, в выходные дни это случается несколько чаще. Отфильтруем результаты, чтобы видеть только корреляции больше 0,1 или меньше –0,1.

```
for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
}
// → выходной:     0.1371988681
// → чистил зубы: -0.3805211953
// → ел конфеты:   0.1296407447
// → работал:     -0.1371988681
// → ел спагетти:  0.2425356250
// → читал:        0.1106828054
// → ел арахис:    0.5902679812
```

Ага! Есть два фактора с корреляцией, которая явно больше других. Употребление в пищу арахиса оказывает сильное положительное воздействие на вероятность превращения в белку, тогда как чистка зубов имеет существенный отрицательный эффект.

Интересно. Проверим еще один вариант.

```
for (let entry of JOURNAL) {
  if (entry.events.includes("ел арахис") &&
      !entry.events.includes("чистил зубы")) {
    entry.events.push("арахис-зубы");
  }
}
console.log(phi(tableFor("арахис-зубы", JOURNAL)));
// → 1
```

Вот оно! Превращение происходит именно тогда, когда Жак ест арахис и не чистит зубы. Если бы он лучше заботился о зубной гигиене, то никогда не заметил бы своего недуга.

Зная это, Жак совершенно перестает есть арахис и обнаруживает, что его превращения прекращаются.

В течение нескольких лет дела Жака идут отлично. Но в какой-то момент он теряет работу. Поскольку он жил в скверной стране, где отсутствие работы означало отсутствие медицинских услуг, ему пришлось устроиться на работу в цирк, где он выступал в роли «супербелки», набивая рот арахисовым маслом перед каждым представлением.

Однажды, будучи сытым по горло этим жалким существованием, Жак не сумел вернуться в человеческий облик, выпрыгнул в щель циркового шатра и скрылся в лесу. Больше его никогда не видели.

Дальнейшая массивология

Прежде чем закончить главу, я хочу познакомить вас еще с несколькими концепциями, связанными с объектами. Для начала я покажу вам ряд широко используемых методов для работы с массивами.

Ранее в этой главе мы уже познакомились с методами `push` и `pop`, которые добавляют и удаляют элементы в конце массива. Соответствующие методы для добавления и удаления объектов в начале массива называются `unshift` и `shift`.

```
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

Эта программа управляет очередью задач. Чтобы добавить задачу в конец очереди, нужно вызвать функцию `remember("купить продукты")`, а когда вы будете готовы что-то сделать, нужно вызвать `getTask()`, чтобы получить

(и удалить) первый элемент из очереди. Функция `rememberUrgently` также добавляет задачу, но не в конец, а в начало очереди.

Для поиска определенного значения у массивов есть метод `indexOf`. Этот метод просматривает массив от начала до конца и возвращает индекс, по которому было найдено заданное значение, или –1, если оно не было найдено. Для поиска с конца, а не с начала существует аналогичный метод, который называется `lastIndexOf`.

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

И `indexOf`, и `lastIndexOf` принимают необязательный второй аргумент, который указывает, откуда начать поиск.

Еще один фундаментальный метод для работы с массивами — это `slice`. Он принимает начальный и конечный индексы и возвращает массив, содержащий только элементы, расположенные между ними, включая начальный, но исключая конечный индекс.

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

Если конечный индекс не задан, `slice` вернет все элементы, идущие после начального индекса. Можно также опустить начальный индекс, чтобы скопировать весь массив.

С помощью метода `concat` можно склеивать массивы, чтобы получить новый, аналогично тому, что делает оператор `+` для строк.

В следующем примере показано применение `concat` и `slice`. Функция принимает массив и индекс и возвращает новый массив, являющийся копией исходного массива, из которого удален элемент, расположенный по указанному индексу.

```
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"]
```

Если передать функции `concat` аргумент, который не является массивом, значение будет добавлено в новый массив, как если бы это был массив из одного элемента.

Строки и их свойства

Мы можем прочитать такие свойства строковых значений, как `length` и `toUpperCase`. Но добавить к строке новое свойство не получится.

```
let kim = "Ким";
kim.age = 88;
console.log(kim.age);
// → undefined
```

Строковые, числовые и логические значения не являются объектами, и хотя язык не будет жаловаться, если вы попытаетесь назначить им новые свойства, фактически он эти свойства не сохранит. Как упоминалось ранее, такие значения неизменяемы и не могут быть изменены.

Но у таких типов есть встроенные свойства. Каждое строковое значение имеет ряд методов. Из них особенно полезными являются `slice` и `indexOf`, которые подобны одноименным методам для массивов.

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

Одно из отличий состоит в том, что `indexOf` для строки может искать строку, содержащую более одного символа, тогда как соответствующий метод массива ищет только один элемент.

```
console.log("один два три".indexOf("ри"));
// → 10
```

Метод `trim` удаляет пробельные символы (пробелы, символы перевода строки, табуляции и аналогичные символы) в начале и конце строки.

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

Функция `zeroPad` из предыдущей главы также существует в виде метода. Он называется `padStart` и принимает в качестве аргументов желаемую длину и символ заполнения.

```
console.log(String(6).padStart(3, "0"));
// → 006
```

С помощью функции `split` можно разделить строку на части в местах, где встречаются заданные фрагменты, а затем снова ее соединить с помощью функции `join`.

```
let sentence = "Птицы-секретари умеют громко топать";
let words = sentence.split(" ");
console.log(words);
// → ["Птицы-секретари", "умеют", "громко", "топать"]
console.log(words.join(". "));
// → Птицы-секретари. умеют. громко. топать
```

С помощью метода `repeat` можно повторить строку несколько раз — будет создана новая строка, содержащая несколько склеенных вместе копий исходной строки.

```
console.log("ЛА".repeat(3));
// → ЛАЛАЛА
```

Нам уже встречалось свойство `length` для строк. Доступ к отдельным символам в строке осуществляется так же, как доступ к элементам массива (с оговоркой, которую мы обсудим в главе 5).

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

Дополнительные параметры

Возможность функции принимать любое количество аргументов может быть полезной. Например, функция `Math.max` находит максимальный из *всех* переданных ей аргументов.

Для того чтобы написать такую функцию, нужно поставить три точки перед ее последним параметром, например, так:

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
```

```
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

Когда вызывается такая функция, *остаточный параметр* привязывается к массиву, содержащему все остальные аргументы. Если перед ним есть другие параметры, то их значения не являются частью этого массива. Если, как в случае с `max`, это единственный параметр, то он будет содержать все аргументы.

Подобную нотацию с многоточием можно использовать для *вызова* функции с массивом аргументов.

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

Таким образом, массив «развертывается» в вызове функции, передавая элементы в виде отдельных аргументов. Такой массив можно передать вместе с другими аргументами, например: `max (9, ... numbers, 2)`.

Подобная запись массива в квадратных скобках позволяет оператору «многоточие» развернуть один массив внутри другого.

```
let words = ["никогда", "полностью"];
console.log(["я это", ...words, "не пойму"]);
// → ["я это", "никогда", "полностью", "не пойму"]
```

Объект Math

Как мы уже видели, `Math` — это сборная солянка из всевозможных полезных функций для работы с числами, таких как `Math.max` (максимум), `Math.min` (минимум) и `Math.sqrt` (квадратный корень).

Объект `Math` играет роль контейнера, в котором собрана вся эта взаимосвязанная функциональность. Существует только один объект `Math`, и он почти никогда не используется в качестве значения. Скорее, это *пространство имен*, необходимое, чтобы все эти функции и значения не были глобальными привязками.

Наличие слишком большого количества глобальных привязок «загрязняет» пространство имен. Чем больше имен занято, тем выше вероятность того, что

вы случайно перезапишете значение какой-либо из уже существующих привязок. Например, весьма вероятно, что вы захотите присвоить чему-нибудь в одной из своих программ имя `max`. Но поскольку функция `max`, встроенная в JavaScript, надежно скрыта внутри объекта `Math`, нам не приходится беспокоиться о том, что мы можем ее перезаписать.

Многие языки остановят вас или по крайней мере предупредят, если вы определите привязку с именем, которое уже занято. JavaScript делает это для привязок, объявленных с помощью `let` или `const`, но не для стандартных привязок и не для привязок, объявленных с помощью `var` или `function`.

Вернемся к объекту `Math`. Когда нужны тригонометрические вычисления, `Math` приходит на помощь. Здесь есть функции `cos` (косинус), `sin` (синус) и `tan` (тангенс), а также их обратные функции — `acos`, `asin` и `atan` соответственно. Число π (`pi`) — или по крайней мере наиболее точное приближение, которое вписывается в числовой формат JavaScript, — доступно как `Math.PI`. Писать имена констант строчными буквами — старая традиция программирования.

```
function randomPointOnCircle(radius) {
  let 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 не знакомы с синусами и косинусами, не беспокойтесь. Перед тем как использовать их в этой книге, в главе 14 я объясню их значение.

В предыдущем примере была задействована функция `Math.random`. При каждом вызове она возвращает новое псевдослучайное число в диапазоне от нуля (включительно) до единицы (исключительно).

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

Компьютеры являются детерминистическими машинами — они всегда реагируют одинаково на одни и те же входные данные, — однако они способны

генерировать числа, кажущиеся случайными. С такой целью машина сохраняет некоторое скрытое значение и всякий раз, когда вы запрашиваете новое случайное число, выполняет сложные вычисления для этого скрытого значения, чтобы создать новое значение. Компьютер сохраняет его и возвращает некоторое число, полученное на его основе. Таким образом, компьютер может генерировать каждый раз новые, труднопредсказуемые числа способом, *кажущимся* случайным.

Если мы хотим получить целое, а не дробное случайное число, то мы можем применить к результату `Math.random` функцию `Math.floor` (которая округляет число в меньшую сторону до ближайшего целого).

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

Умножая случайное число на 10, мы получим число, которое больше или равно 0 и меньше 10. Поскольку `Math.floor` округляет в меньшую сторону, то это выражение будет с равной вероятностью генерировать числа в диапазоне от 0 до 9.

Есть также функции `Math.ceil` (от слова *ceiling*, округляет до целого числа в бóльшую сторону), `Math.round` (округляет до ближайшего целого числа) и `Math.abs` (которая возвращает абсолютное значение числа, то есть вычисляет противоположное значение для отрицательных чисел и оставляет положительные числа без изменений).

Деструктурирование

Давайте ненадолго вернемся к функции `phi`.

```
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]));
}
```

Одна из причин, по которой эту функцию неудобно читать, — то, что привязка указывает на весь массив целиком. Мы бы предпочли иметь отдельные привязки для *элементов* массива: `let n00 = table[0]` и т. д. К счастью, в JavaScript предусмотрен краткий способ это сделать.

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

Указанный способ также работает для привязок, созданных с помощью `let`, `var` или `const`. Если вы знаете, что значение, которое назначается привязке, является массивом, то вы можете использовать квадратные скобки, чтобы «заглянуть внутрь» этого значения, назначив привязки его содержимому.

Подобный трюк работает и для объектов, только вместо квадратных скобок нужно использовать фигурные.

```
let {name} = {name: "Фараджи", age: 23};
console.log(name);
// → Фараджи
```

Обратите внимание, что если вы попытаетесь деструктурировать `null` или `undefined`, то получите ошибку, как если бы вы попытались напрямую получить доступ к свойству этих значений.

JSON

Поскольку свойства лишь захватывают свои значения, а не содержат их, объекты и массивы хранятся в памяти компьютера как последовательности битов с *адресами* тех участков памяти, где находится их содержимое. Таким образом, массив, внутри которого находится другой массив, состоит по крайней мере из адреса одной области памяти для внутреннего массива и второй — для внешнего и содержит (кроме прочих данных) двоичное число, представляющее позицию внутреннего массива.

Если вы хотите сохранить данные в файле для дальнейшего использования или передать их по сети на другой компьютер, вам нужно каким-то образом преобразовать эти перепутанные адреса памяти в описание, которое можно сохранить или переслать. Вероятно, вы *могли бы* переслать всю память компьютера вместе с адресом интересующего вас значения, но едва ли это лучший вариант.

Вместо этого мы можем *сериализовать* данные — преобразовать их, представив в виде иерархического описания. Существует популярный формат сериализации под названием *JSON* (произносится как «Джейсон»), что

означает JavaScript Object Notation — «объектная нотация JavaScript». Этот формат широко используется для хранения данных и обмена ими в Интернете не только на JavaScript, но и на других языках.

JSON — с некоторыми ограничениями — похож на способ записи массивов и объектов в JavaScript. Все имена свойств должны быть заключены в двойные кавычки, и допускаются только простые выражения данных — без вызовов функций, привязок или чего-либо другого, что связано с фактическими вычислениями. Комментарии в JSON не допускаются.

Дневниковая запись, представленная в виде данных JSON, может выглядеть так:

```
{
   "squirrel": false,
   "events": ["работал", "трогал дерево", "ел пиццу", "бегал"]
}
```

В JavaScript есть функции `JSON.stringify` и `JSON.parse`, предназначенные для преобразования данных в этот формат и обратно. Первая из них принимает значение JavaScript и возвращает строку в формате JSON. Вторая берет такую строку и преобразует ее в значение, которое было закодировано в этой строке.

```
let string = JSON.stringify({squirrel: false,
                             events: ["выходной"]});
console.log(string);
// → {"squirrel":false,"events":["выходной"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

Резюме

Объекты и массивы (которые являются частным случаем объектов) предоставляют способы объединить несколько значений в одно. Это похоже на возможность сложить несколько логически взаимосвязанных вещей в одну сумку и ходить с ней, вместо того чтобы нести все вещи в охапке, пытаясь удержать каждую в отдельности.

У большинства значений JavaScript есть свойства; исключения составляют `null` и `undefined`. Доступ к свойствам осуществляется посредством записи типа `value.prop` или `value["prop"]`. Как правило, у свойств объектов есть

имена, и количество этих свойств более или менее фиксировано. Массивы, напротив, обычно содержат разное количество концептуально идентичных значений, а в качестве имен их свойств используются числа (начиная с 0).

Тем не менее у массивов *есть* несколько именованных свойств, таких как `length`, и несколько методов. Методы — это функции, относящиеся к свойствам и (обычно) воздействующие на значение, свойством которого они являются.

Для перебора массива можно использовать специальную разновидность цикла `for` — `for (let element of array)`.

Упражнения

Сумма диапазона

Во введении к этой книге упоминался следующий хороший способ вычислить сумму диапазона чисел:

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

Напишите функцию `range`, которая принимает два аргумента, `start` и `end`, и возвращает массив, содержащий все числа от `start` до `end` включительно.

Затем напишите функцию `sum`, которая принимает массив чисел и возвращает их сумму. Запустите пример программы и посмотрите, действительно ли она возвращает 55.

В качестве дополнительного задания: измените функцию `range` так, чтобы она принимала необязательный третий аргумент, который указывал бы значение шага, используемое при построении массива. Если шаг не задан, элементы увеличиваются на единицу, что соответствует старому поведению. Вызов функции `range(1, 10, 2)` должен возвращать `[1, 3, 5, 7, 9]`. Убедитесь, что функция также работает и с отрицательными значениями шага, так что результатом `range(5, 2, -1)` является `[5, 4, 3, 2]`.

Массив в обратном порядке

У массивов есть метод `reverse`, который изменяет порядок следования элементов в массиве. Для выполнения этого упражнения напишите две

функции: `reverseArray` и `reverseArrayInPlace`. Первая функция, `reverseArray`, принимает массив в качестве аргумента и создает *новый* массив, содержащий те же элементы в обратном порядке. Вторая, `reverseArrayInPlace`, делает то же, что и метод `reverse`: *преобразовывает* массив, заданный в качестве аргумента, меняя порядок следования его элементов на обратный. Не используйте для этого стандартный метод `reverse`.

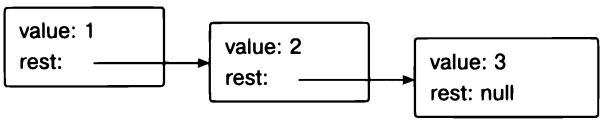
Вспомните, что мы говорили о побочных эффектах и чистых функциях в предыдущей главе, и ответьте на вопрос: какой из этих вариантов, по вашему мнению, будет полезен в большинстве случаев? Какой из них быстрее работает?

Список

Объекты как обобщенные скопления значений можно использовать для построения любых структур данных. Одной из таких распространенных структур данных является *список* (не путать с массивом). Список — это иерархический набор объектов, где первый объект содержит ссылку на второй, второй — на третий и т. д.

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

В результате получается примерно такая цепочка объектов:



Одно из приятных свойств списков — то, что они могут совместно использовать часть собственной структуры. Например, если я создам два новых значения, `{value: 0, rest: list}` и `{value: -1, rest: list}` (где `list` указывает на определенную ранее привязку), то эти значения будут независимыми спи-

сками, но оба они будут задействовать одну и ту же структуру, образующую последние три их элемента. Исходный список также является корректным списком, состоящим из трех элементов.

Напишите функцию `arrayToList`, которая строит список, чья структура подобна показанной, если передать функции массив `[1, 2, 3]` в качестве аргумента. Напишите также функцию `listToArray`, создающую массив из списка. Затем добавьте вспомогательную функцию `prepend`, принимающую элемент и список и создающую новый список, в котором заданный элемент добавлен в начало исходного списка. Кроме того, создайте функцию `nth`, принимающую список и число и возвращающую элемент, находящийся в заданной позиции в этом списке (где ноль соответствует первому элементу), или `undefined`, если элемента в заданной позиции не существует.

Если вам этого все еще недостаточно, напишите рекурсивную версию функции `nth`.

Глубокое сравнение

Оператор `==` сравнивает объекты по их тождественности. Но иногда желательно сравнить значения их реальных свойств.

Напишите функцию `deepEqual`, которая принимает два значения и возвращает `true`, только если эти объекты имеют одинаковое значение или являются объектами с одинаковыми свойствами и значения свойств равны при сравнении с рекурсивным вызовом `deepEqual`.

Чтобы выяснить, нужно сравнивать значения напрямую (для этого используйте оператор `===`) или их свойства, можете использовать оператор `typeof`. Если его результатом для обоих значений является `"object"`, то требуется выполнить глубокое сравнение. Но вам следует принять во внимание одно глупое исключение: исторически сложилось так, что результатом `typeof null` также будет `"object"`.

Для перебора и сравнения свойств объекта вам также пригодится функция `Object.keys`.

5 Функции высшего порядка

> 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.
>
> *Чарльз Энтони Ричард Хоар, 1980, лекция на вручении премии Тьюринга*

Большая программа — это дорогая программа, и не только из-за времени, необходимого для ее разработки. Чем программа больше, тем она сложнее и тем сильнее сбивает с толку программистов. Сбитые с толку программисты, в свою очередь, вносят в программу ошибки *(баги)*. Чем больше программа, тем больше в ней места, где могут скрываться ошибки, что затрудняет их поиск.

Ненадолго вернемся к последним двум примерам программ из введения. Первый из них является автономным и состоит из шести строк.

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

Второй пример опирается на две внешние функции, и его длина составляет всего одну строку.

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

В каком из них выше вероятность допустить ошибку?

Если учесть размер определений функций `sum` и `range`, то вторая программа тоже будет большой — даже больше, чем первая. Но тем не менее я бы сказал, что именно во втором примере меньше вероятность допустить ошибку.

Скорее всего, этот код будет правильным, потому что для решения использован словарь, который соответствует решаемой задаче. При суммировании диапазона чисел речь идет не о циклах и счетчиках, а о диапазонах и суммах.

Определения терминов из данного словаря (функции `sum` и `range`) попрежнему будут включать в себя циклы, счетчики и другие несущественные подробности. Но поскольку эти определения описывают более простые концепции, чем программа в целом, их легче понять.

Абстракции

В контексте программирования такие словари обычно называют *абстракциями*. Абстракции скрывают детали и позволяют обсуждать проблемы на более высоком (более абстрактном) уровне.

В порядке аналогии сравните следующие два рецепта горохового супа. Первый из них выглядит так.

> Положите в емкость сушеный горох из расчета один стакан на порцию. Добавьте воду, чтобы она полностью покрывала горох. Оставьте горох в воде как минимум на 12 часов. Выньте горох из воды и поместите его в кастрюлю. Добавьте воду, четыре стакана на порцию. Накройте кастрюлю крышкой и дайте гороху покипеть в течение двух часов. Возьмите лук, половину луковицы на порцию. Нарежьте лук на кусочки ножом. Добавьте его к гороху. Возьмите сельдерей, один стебель на порцию. Нарежьте его на кусочки ножом. Добавьте сельдерей к гороху. Возьмите морковь, одну штуку на порцию. Нарежьте морковь на кусочки. Ножом! Добавьте морковь к гороху. Держите на огне еще 10 минут.

А вот второй рецепт.

> На одну порцию: один стакан сушеного гороха, половина нарезанного лука, стебель сельдерея и одна морковь.
>
> Замочите горох на 12 часов. Варите на медленном огне два часа в четырех стаканах воды (на порцию). Нарежьте и добавьте овощи. Готовьте еще 10 минут.

Второй рецепт короче, и его проще понять. Но для этого нужно понимать еще несколько связанных с кулинарией слов, таких как *«замочить»*, *«тушить»*, *«нарезать»* и, наверное, *«овощ»*.

В программировании мы не можем рассчитывать, что все необходимые нам слова найдутся в словаре. Из-за этого мы можем пойти по пути первого рецепта — точно описать все шаги, которые должен выполнить компьютер, один за другим, не обращая внимания на понятия более высокого уровня, которые они выражают.

Это полезный навык программирования — замечать, когда вы работаете на слишком низком уровне абстракции.

Абстрагирование повторов

Как мы уже видели, простые функции являются хорошим способом построения абстракций. Но иногда они не оправдывают ожиданий.

В программах часто приходится делать одно и то же несколько раз. Для этого можно написать цикл `for`, например:

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

Можем ли мы абстрагировать операцию «сделать что-то *n* раз» в виде функции? Вполне — легко написать функцию, которая вызывает `console.log` *n* раз.

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

Но что делать, если мы хотим повторить что-то еще, кроме вывода чисел? Поскольку «сделать что-то» можно представить в виде функции, а функции — это просто значения, то можно представить наше действие как функциональное значение.

```
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
```

```
}

repeat(3, console.log);
// → 0
// → 1
// → 2
```

Мы не обязаны передавать в `repeat` заранее определенную функцию. Часто вместо этого проще создать функциональное значение непосредственно в вызове.

```
let labels = [];
repeat(5, i => {
  labels.push(`Блок ${i + 1}`);
});
console.log(labels);
// → ["Блок 1", "Блок 2", "Блок 3", "Блок 4", "Блок 5"]
```

Эта конструкция немного похожа на цикл `for`: сначала описывается тип цикла, а затем предоставляется его тело. Однако теперь тело записывается как значение функции, заключенное в скобки вызова `repeat`. Именно поэтому запись заканчивается закрывающими фигурной *и* обычной скобками. В случаях, подобных данному примеру, когда тело функции представляет собой одно небольшое выражение, можно опустить фигурные скобки и записать цикл в одну строку.

Функции высшего порядка

Функции, которые работают с другими функциями, либо принимая их в качестве аргументов, либо возвращая их, называются *функциями высшего порядка*. Поскольку, как мы уже видели, функции являются обычными значениями, в существовании функций высшего порядка нет ничего особенно примечательного. Сам термин пришел из математики, где различие между функциями и другими значениями более явное.

Функции высшего порядка позволяют абстрагироваться не только от значений, но и от *действий*. Функции высшего порядка бывают разных видов. Например, существуют функции, которые создают новые функции.

```
function greaterThan(n) {
  return m => m > n;
}
let greaterThan10 = greaterThan(10);
```

```
console.log(greaterThan10(11));
// → true
```

И существуют функции, которые изменяют другие функции.

```
function noisy(f) {
  return (...args) => {
    console.log("вызов для", args);
    let result = f(...args);
    console.log("вызов для", args, " возвращает", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → вызов для [3, 2, 1]
// → вызов для [3, 2, 1] возвращает 1
```

Можно даже написать функции, которые создают новые типы последовательности выполнения.

```
function unless(test, then) {
  if (!test) then();
}
repeat(3, n => {
  unless(n % 2 == 1, () => {
    console.log(n, " - четное число");
  });
});
// → 0 - четное число
// → 2 - четное число
```

Существует встроенный метод для массивов `forEach`, который представляет собой нечто вроде цикла `for/of`, реализованного в виде функции высшего порядка.

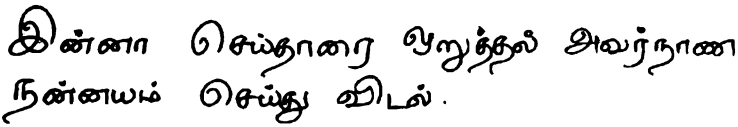
```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

Набор данных о шрифтах

Одной из областей успешного применения функций высшего порядка является обработка данных. Для нее нам понадобятся некоторые настоящие данные. В этой главе мы будем использовать набор данных о шрифтах — системах записи, таких как латиница, кириллица или арабский язык.

Помните Unicode из главы 1, систему, которая присваивает номер каждому символу письменного языка? Большинство из этих символов связаны с определенным шрифтом. Стандартный набор содержит 140 шрифтов — из них 81 используется до сих пор, а 59 уже устарели.

Я свободно читаю только латинские буквы, но понимаю, что люди пишут тексты как минимум в 80 других системах письма, многие из которых мне вообще незнакомы. Например, вот образец рукописного текста на тамильском:



Образец набора данных содержит информацию о 140 шрифтах, определенных в Unicode. Он доступен в изолированной среде программирования для этой главы (https://eloquentjavascript.net/code#5) в виде привязки `SCRIPTS`. Данная привязка содержит массив объектов, каждый из которых описывает определенный шрифт.

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

Такой объект сообщает нам имя шрифта, выделенные для него диапазоны Unicode, направление письма, время создания (приблизительное), используется ли он в настоящее время и ссылку на дополнительную информацию. Направление может быть `"ltr"` (left to right, «слева направо»), `"rtl"` (right to left, «справа налево», как пишут на арабском и иврите) или `"ttb"` (top to bottom, сверху вниз, как в монгольском языке).

Свойство `range` содержит массив диапазонов символов Unicode. Каждый элемент такого массива представляет собой двухэлементный массив, содержащий нижнюю и верхнюю границы диапазона. Все коды символов, находящиеся в этих диапазонах, присваиваются данному шрифту. Нижняя граница включается в диапазон (код 994 является символом коптского алфавита), а верхняя — не включается (код 1008 — не является).

Фильтрация массивов

Для того чтобы найти в наборе данных шрифты, которые все еще используются, нам может пригодиться следующая функция. Она отфильтровывает из массива элементы, не прошедшие тест.

```
function filter(array, test) {
  let passed = [];
  for (let element of array) {
    if (test(element)) {
      passed.push(element);
    }
  }
  return passed;
}

console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", ...}, ...]
```

Эта функция использует аргумент с именем `test` — функциональное значение, позволяющее заполнить пробел в вычислениях — процесс принятия решения о том, какие элементы следует отобрать.

Обратите внимание, что функция `filter` не удаляет элементы из существующего массива, а создает новый массив, содержащий только те элементы, которые проходят тест. Эта функция *чистая*. Она не изменяет переданный ей массив.

Подобно `forEach`, `filter` является стандартным методом массива. В примере я привел определение данной функции только для того, чтобы показать, что происходит внутри нее. С этого момента мы будем использовать ее так:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));
// → [{name: "Mongolian", ...}, ...]
```

Преобразование и отображение

Предположим, что у нас есть массив объектов, представляющих шрифты, созданный путем некой фильтрации массива `SCRIPTS`. Но нам нужен массив имен, который проще просматривать.

Метод `map` преобразует массив, применяя функцию ко всем его элементам и создавая новый массив из возвращаемых значений. Новый массив будет

иметь ту же длину, что и исходный, но его содержимое будет *отображено* функцией в новую форму.

```
function map(array, transform) {
  let mapped = [];
  for (let element of array) {
    mapped.push(transform(element));
  }
  return mapped;
}

let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", ...]
```

Подобно `forEach` и `filter`, `map` является стандартным методом для массивов.

Суммирование с помощью reduce

Еще одна распространенная вещь, которую часто делают с массивами, — вычисление одного значения на их основе. Частный случай этого — уже использованный нами пример с суммированием набора чисел. Другой пример — поиск шрифта, содержащего наибольшее количество символов.

Операция высшего порядка, которая реализует этот шаблон, называется *сокращением* (иногда ее также называют *сверткой*). Данная операция строит значение путем многократного получения одного элемента из массива и комбинирования его с текущим значением. При суммировании чисел мы начинаем с нуля и затем прибавляем к сумме каждый следующий элемент.

Параметрами функции `reduce`, кроме массива, являются комбинирующая функция и начальное значение. Эта функция немного сложнее, чем `filter` и `map`, поэтому присмотритесь к ней внимательно:

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

Стандартный метод для работы с массивами `reduce`, который, конечно же, соответствует этой функции, имеет дополнительное удобство. Если массив содержит хотя бы один элемент, можно не указывать аргумент `start`. Метод выберет в качестве начального значения первый элемент массива и начнет сокращение со второго элемента.

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

Чтобы использовать `reduce` (дважды) для поиска шрифта с наибольшим количеством символов, мы можем написать нечто вроде этого:

```
function characterCount(script) {
  return script.ranges.reduce((count, [from, to]) => {
    return count + (to - from);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return characterCount(a) < characterCount(b) ? b : a;
}));
// → {name: "Han", ...}
```

Функция `characterCount` сокращает диапазоны, назначенные данному шрифту, вычисляя сумму их размеров. Обратите внимание на использование деструктуризации в списке параметров функции сокращения. Затем второй вызов `reduce` задействует предыдущий результат, чтобы найти самый большой шрифт, многократно сравнивая два шрифта и возвращая больший из них.

Шрифт Han насчитывает более 89 000 символов, назначенных ему в стандарте Unicode, что делает его самой большой системой письма в нашем наборе данных. Han — это шрифт, иногда применяемый для китайских, японских и корейских текстов. В их языках много общих символов, хотя они и пишутся по-разному. Консорциум Unicode (расположенный в США) принял решение рассматривать подобнее символы как единую систему записи в целях экономии кодов символов. Это называется *объединением Хань* и до сих пор очень раздражает некоторых людей.

Компонуемость

Подумаем: как можно было бы переписать предыдущий пример (найти самый большой шрифт) без функций высшего порядка? Следующий код не намного хуже.

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", ...}
```

Появилось несколько дополнительных привязок, и программа стала на четыре строки длиннее. Но этот код все еще весьма понятен.

Функции высшего порядка начинают быть по-настоящему полезны, когда нужно *скомпоновать* операции. В качестве примера напишем код, который вычисляет средний год создания шрифтов живых и мертвых языков в наборе данных.

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

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1188
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 188
```

Таким образом, скрипты мертвых языков в Unicode в среднем старше, чем скрипты живых языков.

Это не особенно значимая или удивительная статистика. Но вы, надеюсь, согласитесь, что код, используемый для ее вычисления, нетрудно читать. Это можно представить себе как конвейер: мы начинаем с анализа всех шрифтов, отфильтровываем живые (или мертвые), берем годы их создания, вычисляем среднее значение и округляем результат.

Данное вычисление также можно было бы представить в виде одного большого цикла.

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
```

```
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1188
```

Но в этом коде сложнее понять, что и как вычисляется. А поскольку промежуточные результаты не представлены в виде согласованных значений, пришлось бы проделать гораздо больше работы, чтобы выделить что-нибудь вроде `average` в отдельную функцию.

С точки зрения того, что на самом деле делает компьютер, эти два подхода также принципиально различаются. Первый создает новые массивы при запуске `filter` и `map`, тогда как второй вычисляет только некоторые числа, выполняя меньше работы. Обычно вы можете позволить себе более легко читаемый вариант, но если приходится обрабатывать очень большие массивы и делать это многократно, то менее абстрактный стиль может дать вам дополнительный выигрыш в скорости.

Строки и коды символов

Одно из применений наборов данных — определить, каким шрифтом набран заданный фрагмент текста. Давайте рассмотрим программу, которая это делает.

Вспомним, что для каждого шрифта существует массив диапазонов кодов символов. Поэтому, зная код символа, мы могли бы использовать следующую функцию, чтобы найти соответствующий шрифт (если он есть):

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    })) {
      return script;
    }
  }
    return null;
}

console.log(characterScript(121));
// → {name: "Latin", ...}
```

Метод `some` — это еще одна функция высшего порядка. Он принимает тестовую функцию и сообщает, возвращает ли она `true` для любого элемента массива.

Но как мы получим коды символов в виде строки?

В главе 1 я упоминал, что в JavaScript строки представляются как последовательности 16-битных чисел. Это так называемые *кодовые единицы*. Первоначально предполагалось, что в Unicode код символа будет помещаться в таком блоке (что дает чуть больше 65 000 символов). Когда стало ясно, что этого недостаточно, многие стали возражать против необходимости использовать больше памяти для хранения одного символа. Чтобы решить эту проблему, был изобретен формат UTF-16, используемый в строках JavaScript. В нем наиболее распространенные символы занимают одну 16-битную кодовую единицу, а остальные — две кодовые единицы.

Сегодня принято считать, что UTF-16 был плохой идеей. Кажется, он создан, чтобы плодить ошибки. Можно легко написать программу, для которой кодовые единицы и символы — одно и то же. И если для вашего родного языка не используются символы, занимающие две кодовые единицы, эта программа будет нормально работать. Но, как только кто-нибудь попытается использовать такую программу для менее распространенного алфавита, например для китайских иероглифов, она сразу сломается. К счастью, после появления смайликов для кодирования символов стали повсеместно использоваться две кодовые единицы и бремя решения таких проблем распределилось более справедливо.

К сожалению, очевидные операции со строками JavaScript, такие как получение их длины через свойство `length` и доступ к их содержимому с помощью квадратных скобок, имеют дело только с кодовыми единицами.

```
// Два символа смайликов — лошадь и ботинок
let horseShoe = "🐴👟";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Invalid half-character)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Code of the half-character)
console.log(horseShoe.codePointAt(0));
// → 128052 (Actual code for horse emoji)
```

Метод JavaScript `charCodeAt` возвращает не полный код символа, а кодовую единицу. Появившийся позже метод `codePointAt` возвращает полный символ Unicode. Таким образом, мы могли бы использовать это, чтобы получить

символы из строки. Но аргумент, переданный в `codePointAt`, все еще является индексом в последовательности кодовых единиц. Таким образом, чтобы перебрать все символы в строке, нам все равно нужно решить вопрос о том, занимает символ одну или две кодовые единицы.

В предыдущей главе я упоминал, что цикл `for/of` можно использовать в том числе и для строк. Подобно `codePointAt`, этот тип цикла появился в то время, когда программисты четко осознали проблемы UTF-16. Когда вы применяете этот цикл для строки, он дает реальные символы, а не кодовые единицы.

```
let roseDragon = " 🌹🐲 ";
for (let char of roseDragon) {
  console.log(char);
}
// → 🌹
// → 🐲
```

Если у вас есть символ (который представляет собой строку из одной или двух кодовых единиц), то для того, чтобы получить его код, можно использовать `codePointAt(0)`.

Распознавание текста

У нас есть функция `characterScript` и способ корректного перебора символов в цикле. Следующий шаг — подсчитать количество символов, принадлежащих каждому шрифту. Здесь нам пригодится такая счетная абстракция:

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.findIndex(c => c.name == name);
    if (known == -1) {
      counts.push({name, count: 1});
    } else {
      counts[known].count++;
    }
  }
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

Функция `countBy` принимает на вход коллекцию (все, что можно перебрать в цикле `for/of`) и функцию, вычисляющую имя группы для данного элемента. Функция `countBy` возвращает массив объектов, каждый из которых содержит имя группы и количество найденных для нее элементов.

В этой функции использован еще один метод работы с массивами — `findIndex`. Данный метод чем-то похож на `indexOf`, но вместо поиска конкретного значения он находит первое значение, для которого заданная функция возвращает `true`. В случае если элемент не найден, `findIndex`, как и `indexOf`, возвращает –1.

Используя `countBy`, мы можем написать функцию, сообщающую, какие шрифты были задействованы в данном фрагменте текста.

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "none";
  }).filter(({name}) => name != "none");

  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No scripts found";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说 "woof", 俄罗斯的狗说 "тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

Функция сначала подсчитывает символы по имени шрифта, используя `characterScript`, чтобы назначить им имя, и возвращает строку `"none"` для символов, которые не относятся ни к какому шрифту. Вызов `filter` удаляет запись `"none"` из полученного массива, поскольку эти символы нас не интересуют.

Чтобы иметь возможность вычислять процентные соотношения, нам сначала нужно получить общее количество символов, принадлежащих шрифту, которое мы можем вычислить с помощью метода `reduce`. Если такие символы не найдены, то функция возвращает конкретную строку. В противном случае она преобразует результаты подсчета в удобно читаемые строки с помощью `map`, а затем объединяет их с помощью `join`.

Резюме

Возможность передавать функциональные значения другим функциям является очень полезным аспектом JavaScript. Это позволяет создавать функции, которые моделируют вычисления с «пробелами». Впоследствии при вызове таких функций в коде данные «пробелы» заполняются функциональными значениями.

Для массивов существует ряд полезных методов высшего порядка. Метод `forEach` можно использовать для циклического перебора элементов массива. Метод `filter` возвращает новый массив, содержащий только элементы, удовлетворяющие условию предикативной функции. Преобразование массива посредством выполнения функции для каждого элемента выполняется с помощью `map`. Чтобы объединить все элементы массива в одно значение, можно использовать `reduce`. Метод `some` проверяет, соответствует ли какой-либо элемент заданной предикативной функции. Наконец, метод `findIndex` находит позицию первого элемента, который соответствует предикату.

Упражнения

Свертка

Используйте метод `reduce` в сочетании с методом `concat` для свертки массива, состоящего из нескольких массивов, в один массив, у которого есть все элементы входных массивов.

Ваш собственный цикл

Напишите функцию высшего порядка `loop`, которая представляет собой аналог оператора цикла `for`. Она принимает значение, функцию условия, функцию обновления и функцию тела. На каждой итерации сначала выполняется функция условия для текущего значения цикла. Если эта функция возвращает `false`, то выполнение цикла прекращается. Затем вызывается функция тела, которой передается текущее значение цикла. Наконец, вызывается функция обновления для создания нового значения, и цикл запускается сначала.

При определении функции вы можете использовать обычный цикл для перебора значений.

Метод every

Для массивов существует метод `every`, аналогичный методу `some`. Этот метод возвращает `true`, когда заданная функция возвращает `true` для *каждого* элемента массива. В некотором смысле `some` — это версия оператора `||` для массивов, а метод `every` подобен оператору `&&`.

Реализуйте метод `every`, принимающий в качестве параметров массив и предикативную функцию. Напишите две версии: одну с использованием цикла, а вторую — с применением метода `some`.

Доминирующее направление письма

Напишите функцию, которая вычисляет доминирующее направление письма в строке текста. Помните, что у каждого объекта шрифта есть свойство `direction`, принимающее значения `"ltr"` (left to right, «слева направо»), `"rtl"` (right to left, «справа налево») или `"ttb"` (top to bottom, «сверху вниз»).

Доминирующее направление — это направление большинства символов, которые принадлежат какому-либо шрифту. Вероятно, вам пригодятся описанные в этой главе функции `characterScript` и `countBy`.

6 Тайная жизнь объектов

> Абстрактный тип данных создается путем написания специальной программы [...], определяющей этот тип в терминах операций, которые могут быть выполнены с ним.
>
> *Барбара Лисков. Абстрактные типы данных в программировании*

В главе 4 мы познакомились с объектами JavaScript. В культуре программирования есть раздел, называемый *объектно-ориентированным программированием*, — это набор техник, для которых объекты (и связанные с ними понятия) являются центральным принципом организации программы.

Несмотря на то что до сих пор есть несогласные с точным определением данной концепции, объектно-ориентированное программирование стало основой для многих языков программирования, включая JavaScript. В этой главе будет показано, как идеи объектно-ориентированного программирования могут быть реализованы в JavaScript.

Инкапсуляция

Основная идея объектно-ориентированного программирования состоит в том, чтобы разделить программу на более мелкие части и сделать так, чтобы каждая из них отвечала за управление своим состоянием.

Таким образом, некоторые знания о том, как работает данная часть программы, могут храниться в ней *локально*. Те, кто работает над остальной

частью программы, не должны помнить или даже иметь представление об этих знаниях. Всякий раз, когда локальные детали изменяются, должен обновляться только код, непосредственно имеющий к ним отношение.

Части такой программы взаимодействуют между собой посредством *интерфейсов* — ограниченных наборов функций — или привязок, которые предоставляют полезную функциональность на более абстрактном уровне, скрывая подробности реализации.

Такие части программы моделируются с использованием объектов. Их интерфейс представляет собой определенный набор методов и свойств. Свойства, являющиеся частью интерфейса, называются *общедоступными*. Другие, которые не должны быть доступны для внешнего кода, называются *закрытыми*.

Во многих языках предусмотрен способ различать общедоступные и закрытые свойства, так чтобы закрытые свойства были недоступны для внешнего кода. В JavaScript с его минималистским подходом это не сделано — по крайней мере пока. Идет работа над тем, чтобы добавить в язык такую возможность.

Однако, несмотря на то что в языке не предусмотрено такого различия, программисты на JavaScript успешно используют данную идею. Как правило, доступный интерфейс описывается в документации или в комментариях. Кроме того, имена закрытых свойств принято начинать с символа подчеркивания (_), чтобы отличать их от общедоступных.

Отделение интерфейса от реализации — отличная идея. Обычно это называется *инкапсуляцией*.

Методы

Методы — это не что иное, как свойства, которые содержат функциональные значения. Вот пример простого метода:

```
let rabbit = {};
rabbit.speak = function(line) {
  console.log(`Кролик говорит: '${line}'`);
};

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

Обычно метод должен что-то делать с объектом, для которого он был вызван. Когда функция вызывается как метод, она выглядит как свойство и вызывается сразу, как в `object.method()`: в ее теле существует привязка, называемая `this`, автоматически указывающая на объект, для которого была вызвана функция.

```
function speak(line) {
  console.log(`${this.type} кролик говорит: '${line}'`);
}
let whiteRabbit = {type: "Белый", speak};
let hungryRabbit = {type: "Голодный", speak};

whiteRabbit.speak("Ах вы ушки-усики мои!" +
                  "Как я опаздываю!");
// → Белый кролик говорит: 'Ах вы ушки-усики мои! Как я опаздываю!'
hungryRabbit.speak("Я бы съел сейчас морковку.");
// → Голодный кролик говорит: 'Я бы съел сейчас морковку.'
```

Привязку `this` можно представить как дополнительный параметр, передаваемый другим способом. Если вы хотите передать его явно, то можете использовать метод функции `call`, который принимает значение `this` в качестве первого аргумента и обрабатывает остальные аргументы как обычные параметры.

```
speak.call(hungryRabbit, "Я объелся!");
// → Голодный кролик говорит: 'Я объелся!'
```

Поскольку у каждой функции есть своя привязка `this`, значение которой зависит от способа ее вызова, вы не можете ссылаться на `this` в области видимости обычной функции, определенной с помощью ключевого слова `function`.

Стрелочные функции ведут себя иначе: у них нет собственных объектов `this`, но они могут видеть привязку `this`, принадлежащую области видимости, внутри которой они находятся. Таким образом, можно выполнять что-то вроде следующего кода, ссылающегося на `this` внутри локальной функции:

```
function normalize() {
  console.log(this.coords.map(n => n / this.length));
}
normalize.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

Если бы я написал аргумент для `map`, используя ключевое слово `function`, этот код не работал бы.

Прототипы

Внимательно присмотритесь к этому коду.

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

Я вытащил свойство из пустого объекта. Волшебство!

Ну хорошо, не совсем волшебство. Я просто скрывал от вас информацию о том, как работают объекты JavaScript. Кроме набора свойств, у большинства объектов также есть *прототип*. Прототип — это другой объект, используемый в качестве запасного источника свойств. Когда объект получает запрос на свойство, которого у него нет, будет выполняться поиск такого свойства в его прототипе, затем в прототипе прототипа и т. д.

А что является прототипом данного пустого объекта? Это великий предок всех прототипов, сущность, от которой происходят почти все объекты, — `Object.prototype`.

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

Как вы уже, вероятно, догадались, `Object.getPrototypeOf` возвращает прототип объекта.

Взаимосвязи между прототипами объектов в JavaScript образуют древовидную структуру, и корнем этого дерева является `Object.prototype`. Данный объект предоставляет несколько методов, присущих всем объектам, таких как метод `toString`, который преобразует объект в строковое представление.

Для многих объектов `Object.prototype` не выступает непосредственным прототипом — вместо этого существует другой объект, который предоставляет по умолчанию другой набор свойств. Функции наследуются от `Function.prototype`, а массивы — от `Array.prototype`.

```
console.log(Object.getPrototypeOf(Math.max) ==
            Function.prototype);
```

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

Каждый подобный объект-прототип, в свою очередь, имеет свой прототип, и часто это `Object.prototype`, который все равно неявно предоставляет такие методы, как `toString`.

Для создания объекта с конкретным прототипом можно использовать `Object.create`.

```
let protoRabbit = {
  speak(line) {
    console.log(`${this.type} кролик говорит: '${line}'`);
  }
};
let killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "Боевой";
killerRabbit.speak("ПИФ-ПАФ!");
// → Боевой кролик говорит 'ПИФ-ПАФ!'
```

Свойства, подобные `speak(line)`, в объектном выражении являются кратким способом определения метода. При этом создается свойство с именем `speak`, значением которого будет функция.

«Протокролик» играет роль контейнера для свойств, общих для всех кроликов. Отдельный объект-кролик, такой как `killerRabbit`, содержит свойства, присущие только ему (в данном случае его тип), и наследует общие свойства от своего прототипа.

Классы

Система прототипов JavaScript может быть интерпретирована как несколько неформальная реализация объектно-ориентированной концепции, называемой *классами*. Класс — вариант типа объекта, описывающий, какие методы и свойства имеет данный объект. Такой объект называется *экземпляром* класса.

Прототипы полезны для определения свойств, имеющих одинаковое значение для всех экземпляров класса, таких как методы. Свойства, которые отличаются в каждом конкретном случае, такие как свойство `type` для наших кроликов, должны храниться непосредственно в самих объектах.

Таким образом, чтобы создать экземпляр данного класса, нужно создать объект, производный от выбранного прототипа, но, *кроме того,* следует убедиться, что сам этот объект обладает свойствами, которые должны иметь экземпляры данного класса. Для этого используется функция *конструктора.*

```
function makeRabbit(type) {
  let rabbit = Object.create(protoRabbit);
  rabbit.type = type;
  return rabbit;
}
```

JavaScript позволяет упростить определение функций этого типа. Если поставить перед вызовом функции ключевое слово `new`, то функция будет считаться конструктором. Это означает, что в ней будет автоматически создан объект с выбранным прототипом, он будет привязан к `this` и возвращен в конце выполнения функции.

Объект-прототип, используемый при построении объектов, определяется путем получения свойства `prototype` функции-конструктора.

```
function Rabbit(type) {
  this.type = type;
}
Rabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} кролик говорит: '${line}'`);
};

let weirdRabbit = new Rabbit("Странный");
```

Конструкторы (в сущности, как и остальные функции) автоматически получают свойство с именем `prototype`, которое по умолчанию содержит просто пустой объект, производный от `Object.prototype`. При желании его можно перезаписать новым объектом. Или же добавить свойства к существующему объекту, как в данном примере.

По соглашению имена конструкторов пишутся с большой буквы, чтобы легко отличать их от других функций.

Важно понимать различие между тем, как прототип связан с конструктором (через свойство `prototype`), и тем, что у объектов *есть* прототип (который можно узнать с помощью `Object.getPrototypeOf`).

Настоящим прототипом конструктора выступает `Function.prototype`, поскольку конструкторы являются функциями. Его *свойство* `prototype`

содержит прототип, используемый для создаваемых с его помощью экземпляров.

```
console.log(Object.getPrototypeOf(Rabbit) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(weirdRabbit) ==
            Rabbit.prototype);
// → true
```

Запись классов

Таким образом, классы в JavaScript — это функции-конструкторы со свойством прототипа. Так они работают, и именно так до 2015 года их следовало писать. В наши дни появилась менее неуклюжая запись.

```
class Rabbit {
  constructor(type) {
    this.type = type;
  }
  speak(line) {
    console.log(`${this.type} кролик говорит: '${line}'`);
  }
}

let killerRabbit = new Rabbit("Убийственный");
let blackRabbit = new Rabbit("Черный");
```

Ключевое слово `class` означает начало описания класса. В этом описании содержатся определение конструктора и набор методов — все в одном месте. В скобках описания класса может содержаться любое количество методов. Один из таких методов, с именем `constructor`, имеет специальное назначение. Он представляет собой функцию конструктора, которая будет связана с именем `Rabbit`. Остальные методы будут упакованы в прототип данного конструктора. Таким образом, это определение класса эквивалентно определению конструктора из предыдущего раздела, но выглядит лучше.

В настоящее время определения классов позволяют создавать в прототипе только *методы* — свойства, которые содержат функции. Это бывает не очень удобно, если нужно сохранить в классе нефункциональные значения. Вероятно, в следующей версии языка это будет улучшено. Пока что мы можем создавать такие свойства, непосредственно управляя прототипом после того, как определили класс.

Подобно ключевому слову `function`, ключевое слово `class` может применяться как в инструкциях, так и в выражениях. При использовании в качестве выражения оно не определяет привязку, а лишь создает конструктор как значение. В выражении класса имя класса разрешается опустить.

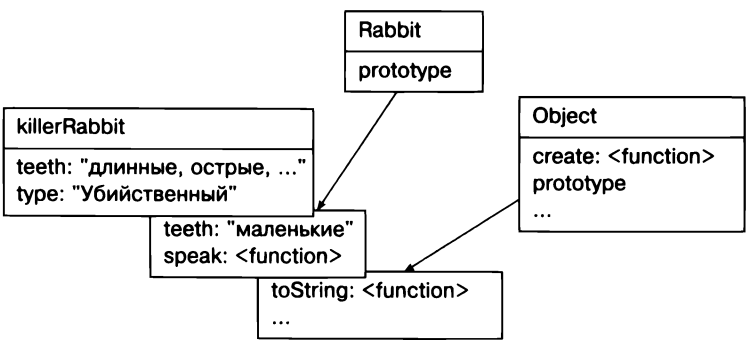
```
let object = new class { getWord() { return "привет"; } };
console.log(object.getWord());
// → привет
```

Переопределение производных свойств

Когда вы добавляете свойство к объекту, независимо от того, присутствует оно в прототипе или нет, свойство добавляется к *самому объекту*. Если в прототипе уже было свойство с таким именем, оно больше не будет влиять на объект, будучи скрыто за собственным свойством объекта.

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

На следующей схеме показана ситуация после выполнения этого кода. В основе объекта `killerRabbit` лежат прототипы `Rabbit` и `Object` — как своего рода фон, откуда можно получить свойства, которых нет в самом объекте.



Переопределение свойств, уже существующих в прототипе, может быть полезным. Как показывает пример с зубами кролика, переопределение может использоваться для выражения особых свойств экземпляра в случае наличия более общего класса, позволяя неисключительным объектам принимать стандартные значения из их прототипа.

Переопределение также применяется для того, чтобы придать стандартным прототипам функции и массива новый метод `toString`, отличный от прототипа базового объекта.

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

Вызов `toString` для массива дает результат, аналогичный вызову `.join(",")`, — между значениями в массиве появляются запятые. При непосредственном вызове `Object.prototype.toString` для массива создается другая строка. Эта функция ничего не знает о массивах, поэтому выдает просто слово *object* и имя типа, заключенные в квадратные скобки.

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

Словари

Мы уже видели *отображение* слов (map), использованное в предыдущей главе, где этим словом описывалась определенная операция, которая преобразовывала структуру данных путем применения функции к ее элементам. Как ни странно, в программировании то же слово *map* применяется для связанного с этим, но совсем другого понятия.

Словарь — также структура данных, которая связывает значения (ключи) с другими значениями. Например, можно сопоставить имена и возраст. Для этого можно использовать объекты.

```
let ages = {
  Борису: 39,
  Ли: 22,
  Юлии: 62
};
```

```
console.log(`Юлии ${ages["Юлии"]}`);
// → Юлии 62
console.log("Знаем ли мы, сколько лет Джеку?", "Джеку" in ages);
// → Знаем ли мы, сколько лет Джеку? false
console.log("Знаем ли мы, сколько лет toString?", "toString" in ages);
// → Знаем ли мы, сколько лет toString? true
```

Здесь имена свойств объекта — это имена людей, а значения свойств — их возраст. Но мы определенно не указали в словаре никого с именем `toString`. Тем не менее, поскольку простые объекты являются производными от `Object.prototype`, похоже, что такое свойство есть.

Поэтому использовать простые объекты в качестве словарей опасно. Есть несколько способов избежать подобной проблемы. Во-первых, можно создавать объекты *без* прототипа. Если передать `null` в `Object.create`, то полученный объект не будет производным от `Object.prototype` и его можно будет безопасно задействовать в качестве словаря.

```
console.log("toString" in Object.create(null));
// → false
```

Имена свойств объекта должны быть строками. Если нам нужен словарь, ключи которого нельзя легко преобразовать в строки — например, потому, что это объекты, — мы не можем использовать объект в качестве словаря.

К счастью, в JavaScript есть класс `Map`, который написан специально на такой случай. Он хранит отображение и позволяет задействовать любые типы ключей.

```
let ages = new Map();
ages.set("Борису", 39);
ages.set("Ли", 22);
ages.set("Юлии", 62);

console.log(`Юлии ${ages.get("Юлии")}`);
// → Юлии 62
console.log("Знаем ли мы, сколько лет Джеку?", ages.has("Джеку"));
// → Знаем ли мы, сколько лет Джеку? False
console.log(ages.has("toString"));
// → false
```

Методы `set`, `get` и `has` являются частью интерфейса объекта `Map`. Создать структуру данных, которая позволяла бы быстро обновлять и выполнять поиск в большом наборе значений, нелегко, но нам можно не беспокоиться.

Это уже сделано до нас, и мы можем воспользоваться данным простым интерфейсом, чтобы использовать готовое.

Если у нас все же есть простой объект, который по какой-то причине нужно задействовать как словарь, то будет полезно знать, что `Object.keys` возвращает только *собственные* ключи объекта, а не ключи из прототипа. В качестве альтернативы оператору `in` можно использовать метод `hasOwnProperty`, который игнорирует прототип объекта.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

Полиморфизм

Когда вы вызываете функцию `String` (которая преобразует значение в строку) для объекта, он вызывает метод `toString` для этого объекта, чтобы попытаться сформировать из него осмысленную строку. Я упомянул, что в отдельных стандартных прототипах определена собственная версия `toString`, поэтому они позволяют создать строку, которая содержит больше полезной информации, чем `"[object Object]"`. Вы также можете сделать это самостоятельно.

```
Rabbit.prototype.toString = function() {
  return `${this.type} кролик`;
};

console.log(String(blackRabbit));
// → Черный кролик
```

Это простой пример мощной идеи. Если часть кода написана для работы с объектами, имеющими определенный интерфейс — в данном случае метод `toString`, — то любой объект, который поддерживает указанный интерфейс, может быть вставлен в этот код и он будет работать.

Подобная методика называется *полиморфизмом*. Полиморфный код может работать со значениями разных видов, если они поддерживают ожидаемый интерфейс.

В главе 4 я упомянул, что цикл `for/of` позволяет перебирать разные виды структур данных. Это еще один случай полиморфизма — такие циклы ожидают, что структура данных предоставит определенный интерфейс,

как у массивов и строк. Но ведь и мы можем снабдить таким интерфейсом наши объекты! Однако прежде, чем это сделать, нам нужно узнать, что такое символы.

Символы

Несколько интерфейсов могут использовать одно и то же имя свойства для разных целей. Например, я мог бы определить интерфейс, в котором метод `toString` преобразовывал бы объект в кусок пряжи. Тогда для объекта было бы невозможным и соответствовать такому интерфейсу, и выполнять стандартный `toString`.

Это было бы плохой идеей, но данная проблема не так уж сильно распространена. Большинство программистов на JavaScript просто не думают о ней. Однако разработчики языка, чья работа заключается в том, чтобы помнить о таких вещах в любом случае, предоставили нам решение.

Когда я заявил, что имена свойств являются строками, это было не совсем точно. Обычно так и есть, но имена могут быть и *символами*. Символы есть значения, созданные с помощью функции `Symbol`. В отличие от строк, вновь созданные символы уникальны — нельзя создать один и тот же символ дважды.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

Строка, передаваемая в `Symbol`, используется при преобразовании ее в строку и может упростить распознавание символа, например, при отображении в консоли. Но она не имеет никакого другого значения — несколько символов могут иметь одно и то же имя.

Благодаря уникальности и удобству применения в качестве имен свойств символы хорошо подходят для определения интерфейсов, которые могут мирно сосуществовать с другими свойствами, независимо от их имен.

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
```

```
  return `${this.length} см голубой шерсти`;
};

console.log([1, 2].toString());
// → 1,2
console.log([1, 2][toStringSymbol]());
// → 2 см голубой шерсти
```

Чтобы включить свойства символа в выражения объекта и класса, нужно заключить имя свойства в квадратные скобки. В результате благодаря нотации заключения в квадратные скобки имя свойства будет вычисляться, что позволяет ссылаться на привязку, которая содержит символ.

```
let stringObject = {
  [toStringSymbol]() { return "джутовая веревка"; }
};
console.log(stringObject[toStringSymbol]());
// → джутовая веревка
```

Интерфейс итератора

Ожидается, что объект, переданный в цикл `for/of`, будет *итерируемым*. Это означает, что у него есть метод, названный символом `Symbol.iterator` (значение символа, определенное языком и сохраненное как свойство функции `Symbol`).

При вызове данный метод должен возвращать объект, предоставляющий другой интерфейс — *итератор*. Это именно та штука, которая выполняет перебор. У итератора есть метод `next`, возвращающий следующий результат. Такой результат должен быть объектом со свойством `value`, которое содержит следующее значение, если оно есть. Также у него должно быть свойство `done`, являющееся истинным, если результатов больше нет, и ложным в противном случае.

Обратите внимание, что имена свойств `next`, `value` и `done` представляют собой не символы, а простые строки. Только метод `Symbol.iterator`, который может добавляться к *самым разным* объектам, действительно является символом.

Мы можем использовать этот интерфейс напрямую.

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
```

```
// → {value: "O", done: false}
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

Реализуем итерируемую структуру данных. Мы создадим класс *матрицы*, действующий как двумерный массив.

```
class Matrix {
  constructor(width, height, element = (x, y) => undefined) {
    this.width = width;
    this.height = height;
    this.content = [];

    for (let y = 0; y < height; y++) {
      for (let x = 0; x < width; x++) {
        this.content[y * width + x] = element(x, y);
      }
    }
  }

  get(x, y) {
    return this.content[y * this.width + x];
  }
  set(x, y, value) {
    this.content[y * this.width + x] = value;
  }
}
```

Содержимое этого класса хранится в общем массиве элементов размером `width` × `height`. Элементы хранятся построчно, так что, например, третий элемент пятой строки (с отсчетом индексов от нуля) хранится в позиции 4 × `width` + 2.

Функция конструктора принимает ширину, высоту и необязательную функцию содержимого, которая используется для заполнения начальных значений. Определены также методы `get` и `set` для извлечения и изменения элементов матрицы.

При переборе элементов матрицы нас обычно интересуют положение элементов и сами элементы, поэтому у нас будет итератор, создающий объекты со свойствами `x`, `y` и `value`.

```
class MatrixIterator {
  constructor(matrix) {
    this.x = 0;
    this.y = 0;
    this.matrix = matrix;
  }

  next() {
    if (this.y == this.matrix.height) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matrix.get(this.x, this.y)};
    this.x++;
    if (this.x == this.matrix.width) {
      this.x = 0;
      this.y++;
    }
    return {value, done: false};
  }
}
```

Класс отслеживает процесс перебора матрицы по ее свойствам x и y. Метод next начинается с проверки, достигнут ли конец матрицы. Если этого не произошло, он *сначала* создает объект, содержащий текущее значение, а *затем* изменяет его позицию, при необходимости переходя к следующей строке.

Настроим итерируемый класс Matrix. В этой книге я иногда буду использовать манипуляции с прототипами постфактум при добавлении методов в классы, чтобы приводимые в качестве примеров фрагменты кода были небольшими и самодостаточными. В обычной программе, где нет необходимости разбивать код на маленькие порции, такие методы объявляются прямо в классе.

```
Matrix.prototype[Symbol.iterator] = function() {
  return new MatrixIterator(this);
};
```

Теперь мы можем перебрать матрицу в цикле for/of.

```
let matrix = new Matrix(2, 2, (x, y) => `значение ${x},${y}`);
for (let {x, y, value} of matrix) {
  console.log(x, y, value);
}
```

```
// → 0 0 значение 0,0
// → 1 0 значение 1,0
// → 0 1 значение 0,1
// → 1 1 значение 1,1
```

Геттеры, сеттеры и статические методы

Интерфейсы часто состоят главным образом из методов, но они также могут включать в себя свойства, которые содержат нефункциональные значения. Например, объекты `Map` имеют свойство `size`, сообщающее нам о количестве хранящихся в них ключей.

Для подобных объектов даже нет необходимости вычислять и хранить такое свойство непосредственно в экземпляре. Даже свойства, к которым есть прямой доступ, могут скрывать вызов метода. Такие методы называются *геттерами*, и для их определения ставится слово `get` перед именем метода в выражении объекта или объявлении класса.

```
let varyingSize = {
  get size() {
    return Math.floor(Math.random() * 100);
  }
};

console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

Всякий раз, когда кто-то читает содержимое свойства `size` такого объекта, вызывается связанный метод. Для того чтобы сделать подобное для записи свойства, используем *сеттер*.

```
class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }
```

```
  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

Класс `Temperature` позволяет читать и записывать температуру в градусах Цельсия или Фаренгейта, но внутри температура хранится только в градусах Цельсия; преобразование из них в градусы Фаренгейта и обратно выполняется автоматически в геттере и сеттере.

Иногда желательно прикрепить некоторые свойства не к прототипу, а непосредственно к функции конструктора. Такие методы не будут иметь доступа к экземпляру класса, но их можно использовать, например, для предоставления дополнительных способов создания экземпляров.

Методы, описанные внутри объявления класса, перед именем которых стоит слово `static`, хранятся в конструкторе. Таким образом, класс `Temperature` позволяет написать `Temperature.fromFahrenheit(100)`, чтобы получить температуру в градусах Фаренгейта.

Наследование

Есть матрицы, называемые *симметричными*. Если зеркально отобразить симметричную матрицу относительно ее диагонали, соединяющей верхний левый и нижний правый углы, то матрица не изменится. Другими словами, значение, хранящееся в ячейке x, y, всегда совпадает со значением в ячейке y, x.

Предположим, что нам нужна структура данных, подобная `Matrix`, но такая, которая гарантировала бы, что матрица является симметричной и остается таковой. Мы могли бы написать такую структуру данных с нуля, но это потребовало бы повторения кода, очень похожего на тот, что мы уже написали.

Система прототипов JavaScript позволяет создать *новый* класс, очень похожий на старый, но с новыми определениями некоторых свойств. Прототип

нового класса происходит от старого прототипа, но в него можно добавить новое определение — например, для метода `set`.

В терминах объектно-ориентированного программирования это называется *наследованием*. Новый класс наследует свойства и поведение старого класса.

```
class SymmetricMatrix extends Matrix {
  constructor(size, element = (x, y) => undefined) {
    super(size, size, (x, y) => {
      if (x < y) return element(y, x);
      else return element(x, y);
    });
  }

  set(x, y, value) {
    super.set(x, y, value);
    if (x != y) {
      super.set(y, x, value);
    }
  }
}

let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);
console.log(matrix.get(2, 3));
// → 3,2
```

Слово `extends` указывает на то, что данный класс основан не на прототипе `Object`, как принято по умолчанию, а на каком-то другом классе, который называется *суперклассом*. Производный класс является *подклассом*.

Чтобы инициализировать экземпляр `SymmetricMatrix`, его конструктор вызывает конструктор своего суперкласса через ключевое слово `super`. Это необходимо, поскольку если новый объект должен вести себя (приблизительно) как `Matrix`, его экземпляру потребуются свойства матрицы. Для обеспечения симметричности матрицы конструктор оборачивает функцию `element`, чтобы поменять местами координаты и получить значения ниже диагонали.

В методе `set` снова применяется слово `super`, но на этот раз для вызова не конструктора, а конкретного метода из множества методов суперкласса. Мы переопределяем метод `set`, но хотим использовать исходное поведение этого метода. Поскольку `this.set` ссылается на *новый* метод `set`, такой вызов не будет работать. Слово `super`, применяемое внутри описания метода класса, позволяет вызывать методы так, как если бы они были определены в суперклассе.

Наследование позволяет создавать типы данных, слегка отличающиеся от уже существующих, с относительно небольшими затратами. Это фундаментальная часть объектно-ориентированной традиции наряду с инкапсуляцией и полиморфизмом. Но если последние две концепции в настоящее время принято считать прекрасными идеями, наследование вызывает больше споров.

Если инкапсуляция и полиморфизм применяются для того, чтобы *отделить* части кода друг от друга, делая программу в целом менее запутанной, то наследование фундаментально связывает классы вместе, создавая *больше* путаницы. При наследовании от класса обычно необходимо знать о том, как он работает, больше, чем при простом использовании. Наследование может быть полезным инструментом, и я к нему время от времени прибегаю в своих программах, но оно не должно быть вашим главным инструментом, и вам, вероятно, не следует активно искать возможности для построения иерархий (генеалогических деревьев) классов.

Оператор instanceof

Иногда полезно знать, является ли объект производным от определенного класса. Для этого в JavaScript существует бинарный оператор `instanceof`.

```
console.log(
  new SymmetricMatrix(2) instanceof SymmetricMatrix);
// → true
console.log(new SymmetricMatrix(2) instanceof Matrix);
// → true
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);
// → false
console.log([1] instanceof Array);
// → true
```

Оператор просматривает унаследованные типы, так что `SymmetricMatrix` для него является экземпляром `Matrix`. Оператор также может быть применен к стандартным конструкторам, таким как `Array`. Почти каждый объект представляет собой экземпляр `Object`.

Резюме

Итак, объекты могут нечто большее, чем просто хранить свои свойства. У них есть прототипы, также являющиеся объектами. Объекты действуют

так, как будто у них есть свойства, которых на самом деле у них нет, если эти свойства есть у их прототипа. Прототипом простых объектов выступает `Object.prototype`.

Конструкторы, представляющие собой функции, имена которых обычно начинаются с заглавной буквы, могут использоваться с оператором `new` для создания новых объектов. Прототипом нового объекта является объект, находящийся в свойстве `prototype` конструктора. Это можно эффективно применять, поместив в прототип свойства, значения которых будут общими для данного типа. Запись с использованием слова `class` позволяет ясно описать конструктор и его прототип.

Определив геттеры и сеттеры, вы создаете методы, которые будут незаметно вызываться при каждом обращении к объекту. Статические методы — это методы, хранящиеся в конструкторе класса, а не в его прототипе.

Оператор `instanceof`, вызванный для данного объекта и конструктора, позволяет узнать, является ли данный объект экземпляром этого конструктора.

Одно из полезных применений объектов — возможность определить для них интерфейс и сообщить всем, что взаимодействие с объектом допустимо только через этот интерфейс. Остальные детали реализации объекта теперь инкапсулированы — скрыты за интерфейсом.

Один и тот же интерфейс может применяться несколькими типами. Код, написанный для использования интерфейса, автоматически знает, как работать с любым количеством различных объектов, которые реализуют данный интерфейс. Этот принцип называется *полиморфизмом*.

При реализации нескольких классов, различающихся только отдельными деталями, бывает полезно создавать новые классы как *подклассы* существующего класса, *наследуя* часть его поведения.

Упражнения

Тип вектора

Напишите класс `Vec`, который представляет вектор в двумерном пространстве. Вектор принимает параметры `x` и `y` (числа) и сохраняет их в свойствах с тем же именем.

Напишите для прототипа `Vec` два метода, `plus` и `minus`, которые принимают в качестве параметра другой вектор и возвращают новый вектор, представляющий собой сумму или разность значений `x` и `y` для двух векторов (`this` и параметра).

Добавьте в прототип свойство-геттер `length`, которое вычисляет длину вектора — расстояние от точки (x, y) до начала координат $(0, 0)$.

Группы

Стандартная среда JavaScript предоставляет еще одну структуру данных, которая называется `Set`. Подобно экземпляру `Map`, `Set` содержит коллекцию значений. В отличие от `Map` класс `Set` не связывает с ними другие значения — только отслеживает, какие значения являются частью множества. Значение может входить в состав множества только один раз — попытка добавить его снова не будет иметь никакого эффекта.

Напишите класс с именем `Group` (поскольку имя `Set` уже занято). Как и `Set`, он располагает методами `add`, `delete` и `has`. Его конструктор создает пустую группу, `add` добавляет в нее значение (но только если такого значения там еще нет), метод `delete` удаляет свой аргумент из группы (если таковой там был), а `has` возвращает логическое значение, указывающее, является ли его аргумент членом группы.

Для того чтобы определить, одинаковы ли два значения, используйте оператор `===` или какой-либо его эквивалент, например `indexOf`.

Присвойте классу статический метод `from`, который принимает в качестве аргумента итерируемый объект и создает группу, содержащую все значения, полученные посредством перебора.

Итерируемые группы

Сделайте класс `Group` из предыдущего упражнения итерируемым. Если вы не помните точный вид интерфейса итератора, перечитайте раздел, посвященный этому интерфейсу, ранее в данной главе.

Если для представления членов группы вы использовали массив, не возвращайте просто итератор, созданный путем вызова метода `Symbol.iterator` для массива. Это бы сработало, но оно не соответствует цели данного упражнения.

Если ваш итератор ведет себя странно, когда группа изменяется во время итерации, — это нормально.

Заимствование метода

Ранее в этой главе я упоминал, что объект `hasOwnProperty` можно использовать как более надежную альтернативу оператору `in`, если мы хотим проигнорировать свойства прототипа. Но что, если нужно включить в словарь слово `hasOwnProperty`? Тогда вы больше не сможете вызывать этот метод, поскольку его значение будет скрыто за собственным свойством объекта.

Можете ли вы придумать способ вызова `hasOwnProperty` для объекта, у которого есть собственное свойство с таким именем?

7 Проект: робот

> [...] 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 «проектных» главах я на время перестану нагружать вас новой теорией, а вместо этого мы вместе разработаем программу. Теория необходима для обучения программированию, но не менее важно научиться читать и понимать реальные программы.

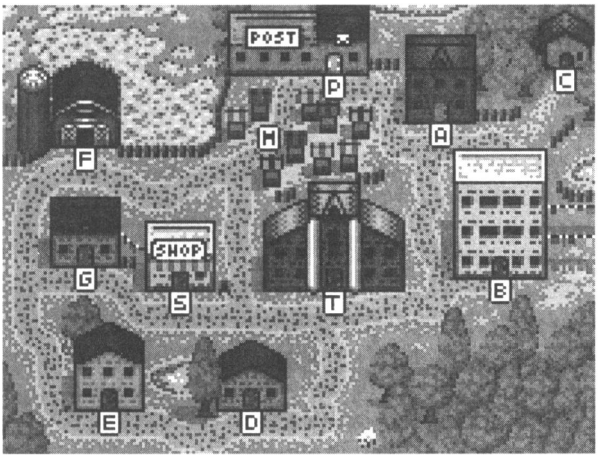
В данной главе наш проект — построить автомат, небольшую программу, которая выполняет задачу в виртуальном мире. Наш автомат будет роботом доставки почты, принимающим и отправляющим посылки.

Деревня Медоуфилд

Деревня Медоуфилд не очень велика. В ней всего 11 домов, и между ними — 14 дорог. Это можно описать таким массивом дорог:

```
const roads = [
  "Дом Алисы-Дом Боба",     "Дом Алисы-Склад",
  "Дом Алисы-Почта",        "Дом Боба-Ратуша",
  "Дом Дарии-Дом Эрни",     "Дом Дарии-Ратуша",
  "Дом Эрни-Дом Греты",     "Дом Греты-Ферма",
  "Дом Греты-Магазин",      "Рынок-Ферма",
  "Рынок-Почта",            "Рынок-Магазин",
  "Рынок-Ратуша",           "Магазин-Ратуша"

];
```



Сеть дорог в деревне образует *граф*. Граф — это множество точек (домов в деревне), соединенных линиями (дорогами). Данный граф будет миром, в котором движется наш робот.

Работать с массивом строк не очень удобно. Нас интересуют пункты назначения, к которым мы можем добраться из данного места. Преобразуем список дорог в структуру данных, позволяющую узнать, куда можно попасть из каждого места.

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
  return graph;
}

const roadGraph = buildGraph(roads);
```

Функция `buildGraph` принимает массив ребер и создает для него объект словаря, в котором каждому узлу соответствует массив связанных с ним узлов.

Метод `split` обеспечивает переход от строк, описывающих дороги и имеющих форму «Начало — конец», к двухэлементным массивам, содержащим начало и конец в виде отдельных строк.

Задача

Наш робот будет перемещаться по деревне. В разных ее местах находятся посылки, каждую из которых нужно доставить по другому адресу. Когда робот приходит по определенному адресу, он забирает оттуда посылки и доставляет их, когда прибывает в место назначения.

Придя по адресу, автомат каждый раз должен принять решение, куда идти дальше. Его задача считается выполненной, когда все посылки доставлены.

Чтобы смоделировать этот процесс, нам нужно определить виртуальный мир, в котором можно описать данный процесс. Такая модель скажет нам, где находится робот в данный момент и где находятся посылки. Когда робот решит куда-либо двинуться, нужно обновить модель, чтобы она отражала новую ситуацию.

Если вы думаете в терминах объектно-ориентированного программирования, то вашим первым побуждением может стать определение объектов для различных элементов этого мира: один класс для робота, другой для посылки, возможно, еще один для адреса. Такие классы могут содержать свойства, описывающие их текущее состояние, — например, стопку посылок по данному адресу, которая бы изменялась при обновлении мира.

Это неправильно.

По крайней мере, в большинстве случаев. Тот факт, что что-то звучит как объект, не означает автоматически, что оно должно быть представлено как объект в вашей программе. Привычка писать классы для каждого концепта в приложении, как правило, приводит к тому, что у нас на руках оказывается набор взаимосвязанных объектов, каждый из которых имеет свое внутреннее, изменяющееся состояние. Такие программы обычно трудно понять и, следовательно, легко сломать.

Вместо этого сведем состояние деревни к минимальному набору значений, его определяющих. Это текущее местоположение робота и множество недоставленных посылок, каждая из которых имеет текущее местоположение и адрес назначения. Вот то, что надо.

А теперь сделаем так, чтобы при перемещениях робота нужно было бы не *изменять* состояние, а вычислять *новое* состояние, отражающее ситуацию после перемещения робота.

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place != this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place != p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

Все действия выполняются в методе `move`. Сначала он проверяет, существует ли дорога из текущего места в пункт назначения, и если нет, то возвращает старое состояние, поскольку это недопустимый ход.

Затем метод создает новое состояние с пунктом назначения в качестве нового местоположения робота. Но для этого также необходимо создать новое множество посылок — те посылки, которые несет робот (находящиеся в текущем местоположении робота), необходимо переместить на новое место.

Кроме того, посылки, адресованные данному месту, должны быть доставлены — другими словами, удалены из множества недоставленных посылок. Перемещения выполняются посредством вызова `map`, а доставка — с помощью вызова `filter`.

При перемещениях объекты посылок не изменяются, а создаются заново. Метод `move` создает новое состояние деревни, при этом не изменяя старое.

```
let first = new VillageState(
  "Почта",
  [{place: "Почта", address: "Дом Алисы"}]
);
let next = first.move("Дом Алисы");
```

```
console.log(next.place);
// → Дом Алисы
console.log(next.parcels);
// → []
console.log(first.place);
// → Почта
```

Результатом перемещения является доставка посылки, и это отражается в следующем состоянии. Но исходное состояние по-прежнему описывает ситуацию, когда робот находится на почте, а посылка не доставлена.

Постоянные данные

Структуры данных, которые не изменяются, называются *неизменяемыми* или *постоянными*. Они ведут себя как строки и числа — в том смысле, что остаются такими, как есть, вместо того чтобы содержать разные значения в разное время.

В JavaScript можно изменить практически *все*, поэтому работа со значениями, которые должны быть постоянными, требует определенных ограничений. Существует функция `Object.freeze`, изменяющая объект таким образом, что попытка записи в его свойства игнорируется. Эту функцию можно использовать, если нужно гарантировать, что объекты не будут изменены, если вы хотите соблюдать осторожность. Правда, такое «замораживание» требует от компьютера выполнения дополнительной работы, а игнорирование обновлений способно запутать человека, читающего код, и навести его на мысль, что он делает что-то не так. Поэтому я обычно предпочитаю просто сказать людям, что данный объект не нужно трогать, и надеюсь, что они это запомнят.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

Почему я всеми силами стремлюсь не менять объекты, хотя язык явно ожидает от меня этого?

Потому что отсутствие изменений помогает мне понимать мои программы. Это снова вопрос сложности управления. Когда объекты в моей системе зафиксированы и стабильны, я могу рассматривать операции над ними

изолированно — перемещение в дом Алисы из заданного начального состояния всегда будет создавать одно и то же новое состояние. Когда объекты изменяются во времени, это добавляет к рассуждениям совершенно новое измерение сложности.

Для небольшой системы, подобной той, которую мы строим в данной главе, мы могли бы справиться с таким незначительным усложнением. Но самое важное ограничение на то, какие системы мы можем построить, — это то, насколько мы можем их понять. Все, что упрощает понимание вашего кода, позволяет строить более амбициозные системы.

Понять систему, построенную на постоянных структурах данных, проще, однако, к сожалению, ее *разработка*, особенно когда выбранный язык программирования этому не способствует, может быть немного сложнее. В настоящей книге мы будем искать возможности применения постоянных структур данных, но и использовать изменяемые структуры.

Моделирование

Робот доставки смотрит на виртуальный мир и принимает решение, в каком направлении двигаться. Таким образом, можно сказать, что робот — это функция, которая принимает объект `VillageState` и возвращает имя ближайшего места назначения.

Поскольку мы хотим, чтобы роботы запоминали какие-то вещи, чтобы составлять и выполнять планы, мы также передаем им их память и позволяем возвращать новую память. Таким образом, робот возвращает объект, содержащий две вещи: направление, в котором он намерен двигаться, и значение памяти, которое будет возвращено ему при следующем вызове.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Выполнено за ${turn} ходов`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Переход в направлении ${action.direction}`);
  }
}
```

Подумайте, что должен сделать робот, чтобы «решить» данное состояние. Он должен забрать все посылки, посетив все места, в которых есть хотя бы одна из них, и доставить посылки, посетив все места, куда адресована хотя бы одна из них, но только после получения соответствующей посылки.

Какова самая нелепая стратегия, которая могла бы сработать? Робот мог бы просто каждый раз делать ход в произвольном направлении. Это означает, что, с большой вероятностью, он в итоге попадет во все места, где есть посылки, а затем также в какой-то момент достигнет места, куда они должны быть доставлены.

Вот как это может выглядеть:

```
function randomPick(array) {
  let choice = Math.floor(Math.random() * array.length);
  return array[choice];
}

function randomRobot(state) {
  return {direction: randomPick(roadGraph[state.place])};
}
```

Вспомним: `Math.random()` возвращает число от нуля до единицы, но всегда меньше единицы. Если умножить такое число на длину массива и затем применить к нему `Math.floor`, то получим случайный индекс для массива.

Поскольку этому роботу не нужно ничего запоминать, он игнорирует второй аргумент (помните, что функции JavaScript можно вызывать с дополнительными аргументами без нежелательных последствий) и пропускает свойство `memory` в возвращаемом объекте.

Чтобы этот сложный робот заработал, нам сначала понадобится способ создать новое состояние с несколькими посылками. Статический метод (здесь он создан путем непосредственного добавления свойства в конструктор) — хорошее место для реализации такой функциональности.

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
```

```
  }
  return new VillageState("Почта", parcels);
};
```

Мы не хотим, чтобы посылки были адресованы в то же место, откуда они были отправлены. По этой причине цикл do продолжает выбирать новые места, если полученное место соответствует адресу посылки.

А теперь мы начнем строить виртуальный мир.

```
runRobot(VillageState.random(), randomRobot);
// → Переход к Рынку
// → Переход к Ратуше
// →...
// → Сделано 63 хода
```

Робот не очень хорошо планирует, и поэтому ему требуется много ходов, чтобы доставить посылки. Скоро мы ими займемся.

Маршрут почтового грузовика

Мы должны уметь намного больше, чем робот со случайным выбором хода. Нам немного поможет подсказка о том, как работает доставка почты в реальном мире. Если мы найдем маршрут, который проходит через все места в деревне, то робот может дважды пройти по этому маршруту, и тогда задача будет гарантированно выполнена. Вот один из таких маршрутов (начиная с почты):

```
const mailRoute = [
  "Дом Алисы", "Сарай", "Дом Алисы", "Дом Боба",
  "Ратуша", "Дом Дарии", "Дом Эрни",
  "Дом Греты", "Магазин", "Дом Греты", "Ферма",
  "Рынок", "Почта"
];
```

Для реализации робота, следующего по маршруту, нам нужно использовать память робота. Он сохраняет остаток маршрута в своей памяти и на каждом ходу отбрасывает первый элемент.

```
function routeRobot(state, memory) {
  if (memory.length == 0) {
    memory = mailRoute;
  }
  return {direction: memory[0], memory: memory.slice(1)};
}
```

Этот робот уже гораздо быстрее. Теперь все займет максимум 26 ходов (двойной обход маршрута из 13 пунктов), но обычно меньше.

Поиск пути

Однако я бы не назвал слепое следование по фиксированному маршруту интеллектуальным поведением. Робот мог бы работать более эффективно, если бы построил свое поведение в соответствии с фактической работой, которую необходимо выполнить.

Для этого он должен быть в состоянии преднамеренно двигаться к определенной посылке или к месту, куда она должна быть доставлена. В связи с чем, особенно если цель находится более чем в одном шаге, потребуется некоторая функция поиска маршрута.

Проблема поиска маршрута по графу является типичной *проблемой поиска*. Мы можем сказать, будет ли данное решение (маршрут) допустимым, но не можем напрямую вычислить решение, как при сложении 2 и 2. Вместо этого приходится продолжать строить потенциальные решения, пока не будет найдено то, которое сработает.

Количество возможных маршрутов через граф бесконечно. Но при поиске маршрута из точки *A* в точку *B* нас интересуют только те из них, что начинаются в точке *A*. Нам также не нужны маршруты, дважды посещающие одно и то же место, — в любом случае это определенно не самый эффективный маршрут. Таким образом, количество маршрутов, которые следует найти при поиске, сокращается.

На самом деле нас больше всего интересует *кратчайший* маршрут. Поэтому мы вначале будем рассматривать короткие маршруты и только потом перейдем к более длинным. Хорошим подходом было бы «наращивать» маршруты из начальной точки, исследуя каждое достижимое место, которое еще не посещалось, пока маршрут не достигнет цели. Таким образом, мы будем исследовать только потенциально интересные маршруты и найдем кратчайший (или один из кратчайших, если их несколько) путь к цели.

Вот функция, делающая это:

```
function findRoute(graph, from, to) {
  let work = [{at: from, route: []}];
  for (let i = 0; i < work.length; i++) {
```

```
    let {at, route} = work[i];
    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
      if (!work.some(w => w.at == place)) {
        work.push({at: place, route: route.concat(place)});
      }
    }
  }
}
```

Исследование маршрутов должно двигаться в правильном направлении — места, которые следует посетить в первую очередь, должны быть исследованы также в первую очередь. Мы не можем сразу исследовать найденное место, потому что это будет означать, что места, до которых можно дойти *из этой точки*, также будут немедленно исследованы и т. д., несмотря на то что могут быть и другие, более короткие пути, нами еще не исследованные.

Поэтому функция ведет *рабочий список*. Это множество мест, которые должны быть исследованы в ближайшее время, а также маршрут, который нас туда доставил. Программа начинает работать с начальной точки и пустого маршрута.

Затем поиск выполняется путем выбора и исследования следующего по списку элемента, а это означает, что будут просмотрены все дороги, идущие из данного места. Если одна из них ведет к цели, то можно вернуть готовый маршрут. В противном случае, если мы еще не рассматривали это место, новый элемент добавляется в список. Если же данное место уже рассмотрено, поскольку мы ищем в первую очередь короткие маршруты, то мы либо нашли более длинный маршрут к этому месту, либо длина такого маршрута такая же, как и другого, уже существующего, и нам не нужно его исследовать.

Визуально это можно представить как сеть известных маршрутов, исходящих из начальной точки и равномерно растущих во все стороны (но никогда не пересекающих себя). Когда первый маршрут достигнет цели, он будет прослежен до начала и мы получим результирующий маршрут.

Наш код не обрабатывает ситуацию, когда в рабочем списке не осталось рабочих элементов, поскольку мы точно знаем, что наш граф является *связным*, то есть любое местоположение может быть достигнуто из всех остальных мест. Мы всегда сможем найти маршрут между двумя точками, и поиск никогда не потерпит неудачу.

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
  return {direction: route[0], memory: route.slice(1)};
}
```

Подобно роботу, который следует по маршруту, этот робот также использует свою память, представленную в виде списка направлений движения. Когда данный список пуст, робот должен выяснить, что делать дальше. Он выбирает первую из множества недоставленных посылок и, если она еще не выбиралась, прокладывает для нее маршрут. Если посылка *уже выбиралась*, ее все равно необходимо доставить, и робот вместо этого создает маршрут к адресу доставки.

Данный робот обычно выполняет задачу доставки пяти посылок примерно за 16 ходов. Это немного лучше, чем `routeRobot`, но все еще едва ли оптимально.

Упражнения

Измерение параметров робота

Трудно объективно сравнивать роботов, просто позволяя им решить несколько задач с разными сценариями. Возможно, одному роботу просто попались более легкие задачи или те, с которыми он лучше справляется, а другому — нет.

Напишите функцию `compareRobots`, которая принимала бы на входе двух роботов (и их стартовую память). Функция должна генерировать 100 задач и позволить каждому из роботов решить каждую из них. После этого должно быть вычислено среднее количество шагов, за которые каждый робот решает одну задачу.

Справедливости ради убедитесь, что вы даете каждую задачу обоим роботам, а не генерируете для них разные задачи.

Эффективность робота

Можете ли вы написать робота, который выполнял бы задачу доставки быстрее, чем `goalOrientedRobot`? Понаблюдайте за поведением такого робота: какие глупости он делает? Как это можно исправить?

Если вы выполнили предыдущее упражнение, то можете использовать функцию `compareRobots`, чтобы проверить, удалось ли вам улучшить робота.

Постоянная группа

Большинство структур данных, предоставляемых стандартной средой JavaScript, не очень хорошо подходят для хранения постоянных данных. У массивов есть методы `slice` и `concat`, которые позволяют легко создавать новые массивы, не повреждая старый. Но у `Set`, например, нет методов для создания нового множества с добавленным или удаленным элементом.

Напишите новый класс `PGroup`, аналогичный классу `Group` из главы 6, в котором хранится множество значений. Подобно `Group`, у него есть методы `add`, `delete` и `has`.

Однако его метод `add` должен возвращать *новый* экземпляр `PGroup` с добавленным заданным элементом, оставляя старый экземпляр без изменений. Аналогично `delete` создает новый экземпляр, в котором нет заданного элемента.

Класс должен работать со значениями любого типа, а не только со строками. Он *не должен* быть эффективным для большого количества значений.

Конструктор не должен быть частью интерфейса класса (хотя вы определенно захотите использовать его внутри класса). Вместо этого существует пустой экземпляр `PGroup.empty`, который можно применять в качестве начального значения.

Зачем нужно единственное значение `PGroup.empty`, если можно создать функцию, которая бы каждый раз генерировала новый пустой словарь?

8 Ошибки и дефекты

> Отлаживать программу вдвое
> сложнее, чем писать код с нуля.
> Поэтому если вы пишете код
> настолько заумный, насколько
> вы способны, то вы по определению
> не будете способны его отладить.
>
> *Брайан Керниган и Филип*
> *Джеймс Плугер. Основы стиля*
> *программирования*

Изъяны компьютерных программ обычно называют *дефектами (багами)*. Программистам нравится представлять их как мелких насекомых, случайно залетевших в механизм. На самом деле, конечно же, мы сами их туда вносим.

Если программа — это воплощенная мысль, то все дефекты можно разбить на две категории: те, что были вызваны путаницей мыслей, и те, что вызваны ошибками, возникшими в процессе преобразования мысли в код. Первый тип, как правило, сложнее диагностировать и исправить, чем второй.

Язык

Компьютер мог бы автоматически указать нам на многие ошибки, если бы он достаточно знал о том, что мы пытаемся сделать. Но этому мешает гибкость JavaScript. Концепция привязок и свойств данного языка слишком расплывчата, поэтому он редко ловит опечатки перед запуском программы. И даже в таком случае он, не жалуясь, позволяет нам делать некоторые явно бессмысленные вещи, такие как вычисление `true * "monkey"`.

Но на отдельные вещи JavaScript все же жалуется. Если написать программу, которая не соответствует грамматике языка, компьютер немедленно сообщит об этом. Другие вещи, такие как вызов чего-то, что не является функцией, или поиск свойства для неопределенного значения, приведут к сообщению об ошибке, когда программа попытается выполнить подобное действие.

Но часто бессмысленные вычисления просто приводят к появлению `NaN` (not a number — «не число») или неопределенного значения, в то время как программа успешно продолжает выполняться, убежденная, что она делает что-то осмысленное. Ошибка проявится позже, после того, как фиктивное значение пройдет через несколько функций. Оно может вообще не вызвать ошибку, а молча привести к неправильному результату программы. Найти источник таких проблем бывает сложно.

Процесс поиска дефектов в программах называется *отладкой*.

Строгий режим

JavaScript можно сделать *немного* строже, включив *строгий режим*. Для этого нужно поставить в начало файла или тела функции строку `"use strict"`. Например:

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++) {
    console.log("Супер-пупер!");
  }
}

canYouSpotTheProblem();
// → ReferenceError: счетчик не определен
```

Обычно, если вы забыли поставить `let` перед привязкой, как в данном примере перед `counter`, JavaScript тихо создает глобальную привязку и использует ее. В строгом режиме вместо этого появляется сообщение об ошибке. Что очень полезно. Следует отметить, однако, что если такая привязка уже существует как глобальная, то это не сработает. В таком случае цикл все равно будет перезаписывать значение привязки.

Другое отличие строгого режима заключается в том, что в функциях, которые не вызываются как методы, привязка `this` содержит значение `undefined`.

При выполнении такого вызова вне строгого режима `this` относится к объекту глобальной области видимости — объекту, свойства которого являются глобальными привязками. Поэтому, если вы случайно неправильно вызовете метод или конструктор в строгом режиме, JavaScript будет выдавать ошибку, как только попытается прочитать что-нибудь из `this`, вместо того чтобы спокойно записывать данные в объект из глобальной области видимости.

В качестве примера рассмотрим следующий код, который вызывает функцию конструктора без ключевого слова `new`, чтобы его привязка `this` *не ссылалась* на вновь созданный объект:

```
function Person(name) { this.name = name; }
let ferdinand = Person("Фердинанд"); // ой
console.log(name);
// → Фердинанд
```

Таким образом, фиктивный вызов `Person` завершился успешно, но вернул неопределенное значение и создал глобальную привязку `name`. В строгом режиме результат будет другим.

```
"use strict";
function Person(name) { this.name = name; }
let ferdinand = Person("Фердинанд"); // забыли поставить new
// → TypeError: Cannot set property 'name' of undefined
```

Нам сразу сообщают, что что-то не так. Это полезно.

К счастью, конструкторы, созданные с помощью нотации `class`, всегда будут сообщать об ошибке, если вызывать их без `new`, что уменьшает количество проблем даже в нестрогом режиме.

У строгого режима есть еще несколько особенностей. В нем запрещено передавать функции нескольких параметров с одинаковыми именами и полностью исключены некоторые проблемные языковые функции (такие как утверждение `with`, настолько неверное, что не обсуждается в этой книге).

Короче говоря, строка `"use strict"`, поставленная вверху программы, редко усложняет жизнь и может помочь обнаружить проблему.

Типы

Некоторые языки желают знать типы всех привязок и выражений еще до запуска программы. Они сразу сообщат вам, если тип используется не по

назначению. JavaScript учитывает типы только при фактическом запуске программы и даже тогда часто пытается неявно преобразовать значения в ожидаемый тип, так что это не сильно помогает.

Тем не менее типы служат удобным каркасом при обсуждении программ. Многие ошибки происходят из-за того, что мы запутались — какое значение подается на вход или является результатом функции. Если у вас есть такая информация, вы вряд ли запутаетесь.

Для описания типа функции перед ней можно добавить комментарий, подобный тому, что показан в следующем примере с функцией `goalOrientedRobot`:

```
// (VillageState, Array) → {direction: string, memory: Array}
function goalOrientedRobot(state, memory) {
  // ...
}
```

Существует ряд соглашений по аннотированию программ JavaScript, касающихся типов.

Один из нюансов, касающихся типов, состоит в том, что они добавляют сложности в программу, чтобы описать достаточное количество полезного кода. Как вы думаете, что будет типом функции `randomPick`, возвращающей случайный элемент массива? Приходится ввести *переменную типа — T*, которая может заменить любой тип, чтобы можно было присвоить функции `randomPick` тип наподобие `([T])→T` (функция, преобразующая массив элементов *T* в один элемент *T*).

Когда типы программ известны, компьютер может их *проверить*, сообщив вам ошибки перед запуском программы. Существует несколько диалектов JavaScript, которые добавляют в язык типы и проверяют их. Самый популярный из них называется TypeScript. Если вам хотелось бы сделать программы более строгими, то я рекомендую вам его попробовать.

В этой книге мы продолжим использовать сырой, опасный, нетипизированный код JavaScript.

Тестирование

Если язык особо не собирается помогать нам находить ошибки, придется искать их, идя более трудным путем: запустив программу и посмотрев, правильно ли она работает.

Делать это вручную снова и снова — очень плохая идея. Это не только раздражает, но и, как правило, неэффективно, поскольку для того, чтобы тщательно все протестировать, каждый раз, когда вы вносите изменения, требуется слишком много времени.

Компьютеры хорошо справляются с повторяющимися задачами, а тестирование — идеальная повторяющаяся задача. Автоматизированное тестирование — процесс написания программы, которая тестирует другую программу. Написание тестов требует немного больше работы, чем тестирование вручную, но после того, как вы это сделаете, вы получите некое подобие высшей власти: вам потребуется всего несколько секунд, чтобы убедиться, что программа по-прежнему работает правильно во всех ситуациях, для которых вы написали тесты. Когда вы что-нибудь сломаете, вы заметите это сразу, а не случайно натолкнувшись позже.

Тесты обычно имеют вид небольших программ, проверяющих те или иные аспекты кода. Например, набор тестов для метода `toUpperCase` (стандартного, возможно уже проверенного кем-то) может выглядеть следующим образом:

```
function test(label, body) {
  if (!body()) console.log(`Failed: ${label}`);
}

test("convert Latin text to uppercase", () => {
    return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΪΡΕΤΕ";
});
test("don't convert case-less characters", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

При написании подобных тестов получается довольно сильно повторяющийся и неуклюжий код. К счастью, существует программное обеспечение, которое помогает создавать и запускать наборы тестов *(тестовые наборы)*, предоставляя язык (в виде функций и методов), подходящий для представления тестов, и выводя информативные данные в случае их сбоя. Обычно такие программы называют *системами выполнения тестов*.

Одни виды кода тестировать проще, другие — сложнее. Как правило, чем больше внешних объектов взаимодействует с кодом, тем сложнее построить контекст для его тестирования. Стиль программирования, продемонстриро-

ванный в предыдущей главе, в котором используются автономные постоянные значения, а не изменяемые объекты, как правило, упрощает тестирование.

Отладка

После того как вы заметите, что с программой что-то не так, она неправильно себя ведет или выдает ошибки, ваш следующий шаг — выяснить, в чем проблема.

Иногда это очевидно. Сообщение об ошибке указывает на конкретную строку программы, и если вы посмотрите на описание ошибки и на эту строку кода, то в большинстве случаев заметите проблему.

Но не всегда. Иногда строка, которая вызвала проблему, является просто первым местом, где нестабильное значение, полученное в другом месте, используется недопустимым образом. Если вы выполнили упражнения для предыдущих глав, то, вероятно, вы уже сталкивались с подобными ситуациями.

В следующем примере программа пытается преобразовать целое число в строку в заданной системе счисления (десятичная, двоичная и т. д.) путем многократного подбора последней цифры с последующим делением числа, чтобы избавиться от этой цифры. Но, судя по странному выводу, который она сейчас производит, в данной программе есть ошибка.

```
function numberToString(n, base = 10) {
  let 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 уже видите проблему, сделайте вид, что ее не замечаете. Мы знаем, что наша программа работает со сбоями, и хотим выяснить, почему так происходит.

Именно здесь необходимо подавить желание начать вносить в код случайные изменения, чтобы посмотреть, не станет ли от этого лучше. Вместо этого *задумайтесь*. Проанализируйте, что происходит, и предложите теорию, почему так может происходить. Затем сделайте дополнительные наблюдения, чтобы проверить эту теорию, или, если теории пока нет, сделайте дополнительные наблюдения, которые помогут вам ее создать.

Хороший способ получить дополнительную информацию о том, что делает программа, — поместить в нее несколько стратегически важных вызовов `console.log`. В данном случае мы хотим, чтобы привязка `n` принимала значения `13`, `1` и затем `0`. Выведем это значение в начале цикла.

```
13
1.3
0.13
0.013...
1.5e-323
```

Вот оно. Деление 13 на 10 не дает целого числа. Чтобы число правильно сдвигалось вправо, вместо `n /= base` нам на самом деле нужно `n = Math.floor(n/base)`.

В качестве альтернативы использованию `console.log`, чтобы заглянуть внутрь поведения программы, можно воспользоваться возможностями *отладчика* браузера. В браузеры встроена возможность устанавливать *контрольные точки* в выбранных строках кода. Когда выполнение программы доходит до строки с контрольной точкой, выполнение приостанавливается и можно проверить значения привязок в этой точке. Я не буду вдаваться в подробности, так как в разных браузерах отладчики различаются, но посмотрите на инструменты разработчика вашего браузера или поищите в Интернете дополнительную информацию.

Другой способ установить контрольную точку — включить в программу инструкцию `debugger` (состоящую только из этого ключевого слова). Если инструменты разработчика в вашем браузере активны, то программа будет останавливаться всякий раз, когда будет встречать такую инструкцию.

Распространение ошибок

Программист, к сожалению, не может предотвратить все проблемы. Если программа каким-либо образом взаимодействует с внешним миром, она

может получить искаженный ввод, оказаться перегруженной работой или пострадать от сбоя сети.

Если вы программируете только для себя, то можете себе позволить просто игнорировать подобные проблемы, пока они не возникнут. Но если то, что вы создаете, будет использоваться кем-то еще, вы наверняка захотите от программы чего-то лучшего, чем постоянные сбои. Иногда лучшее, что можно сделать, — это получить неверные входные данные и продолжить работу. В других случаях предпочтительнее сообщить пользователю, что пошло не так, и сдаться. Но, как бы то ни было, программа должна активно реагировать на проблему.

Предположим, у нас есть функция `promptNumber`, которая запрашивает у пользователя число и возвращает его. Что она должна вернуть, если пользователь введет слово «оранжевый»?

Один из вариантов — возвращать специальное значение. Распространенными вариантами для таких значений являются `null`, `undefined` или `-1`.

```
function promptNumber(question) {
  let result = Number(prompt(question));
  if (Number.isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("Сколько деревьев вы видите?"));
```

Теперь любой код, который вызывает `promptNumber`, должен проверить, действительно ли было прочитано число, и в случае неудачи как-то исправиться — возможно, запросить число снова или ввести значение по умолчанию. Или же код может, в свою очередь, вернуть в *свою* точку вызова специальное значение, указывающее на то, что не удалось выполнить задачу.

Во многих ситуациях — в основном для распространенных ошибок, которые должны быть явно учтены в точке вызова, возвращение специального значения является хорошим способом указать на ошибку. Однако у этого способа есть свои недостатки. Во-первых, как быть, если функция и без того может возвращать все возможные значения? В такой функции приходится, например, обернуть результат в объект, чтобы отличать успешное завершение от неудачного.

```
function lastElement(array) {
  if (array.length == 0) {
```

```
    return {failed: true};
  } else {
    return {element: array[array.length - 1]};
  }
}
```

Вторая проблема, связанная с возвратом специальных значений, заключается в том, что это может привести к неудобочитаемому коду. Если `promptNumber` вызывается в коде десять раз, то нужно десять раз проверить, не было ли возвращено значение `null`. И если в ответ на `null` нужно просто вернуть `null`, то в точках вызова придется тоже это проверить, и т. д.

Исключения

В случае, когда функция не может работать нормально, было бы удобно просто все прекратить и немедленно перейти к тому месту, которое знает, как решить проблему. Это делается посредством *обработки исключений*.

Исключения — механизм, благодаря которому код, столкнувшись с проблемой, может генерировать (или выбрасывать) исключение. Им может быть любое значение. Генерирование исключения напоминает перегруженное возвращаемое значение функции: оно прерывает выполнение не только текущей функции, но и всех вызывающих ее функций, вплоть до первого вызова, запустившего текущее выполнение. Это называется *раскруткой стека*. Это тот самый стек вызовов функций, который, как вы, возможно, помните, упоминался в главе 3. Исключение уменьшает данный стек, отбрасывая все контексты вызовов, которые его касаются.

Если бы исключения всегда достигали самого низа стека, то они бы не были особенно полезны. Это был бы просто еще один способ взорвать программу. Сила исключений состоит в том, что вы можете устанавливать «препятствия» вдоль стека, чтобы перехватить исключение по мере его продвижения. Перехватив исключение, вы можете что-то с ним сделать для решения проблемы, а затем продолжить выполнение программы.

Вот пример такого поведения:

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "Л";
  if (result.toLowerCase() == "right") return "П";
```

```
  throw new Error("Неверное направление: " + result);
}

function look() {
  if (promptDirection("Куда двигаться?") == "L") {
    return "дом";
  } else {
    return "два злых медведя";
  }
}

try {
  console.log("Перед вами", look());
} catch (error) {
  console.log("Что-то пошло не так: " + error);
}
```

Ключевое слово `throw` используется для генерирования исключения. Чтобы перехватить исключение, нужно поместить фрагмент кода в блок `try`, после которого стоит ключевое слово `catch`. Если код в блоке `try` вызывает исключение, то выполняется блок `catch`, у которого в скобках стоит имя, связанное со значением исключения. Когда завершится блок `catch` или если блок `try` выполнится без проблем, программа переходит к инструкциям, стоящим под `try` / `catch`.

В данном случае для создания значения исключения мы использовали конструктор `Error`. Это стандартный конструктор JavaScript, который создает объект со свойством `message`. В большинстве сред выполнения JavaScript экземпляры данного конструктора также собирают информацию о стеке вызовов, существовавшем во время создания исключения, — так называемую *трассировку стека*. Эта информация хранится в свойстве `stack` и может быть полезна при попытке отследить проблему: она сообщает, в какой функции возникла проблема и какие функции сделали вызов, который привел к сбою.

Обратите внимание, что функция `look` полностью игнорирует возможность ошибки в функции `promptDirection`. В этом заключается большое преимущество исключений: код обработки ошибок нужен только в том месте, где происходит ошибка, и в точке, где она обрабатывается. Все функции, расположенные между ними, могут спокойно забыть о проблеме.

Ну хорошо, почти забыть...

Подчищаем за исключениями

Эффект от исключений — это еще один способ изменить последовательность выполнения программы. Каждое действие, которое может вызвать исключение — то есть почти каждый вызов функции и доступ к свойству, — может привести к тому, что управление программой внезапно покинет ваш код.

Если у кода есть побочные эффекты и «обычная» последовательность выполнения выглядит так, как будто они всегда будут наблюдаться, это приводит к тому, что исключение может помешать выполнению некоторых из данных эффектов.

Вот по-настоящему плохой код для банковской программы.

```
const accounts = {
  a: 100,
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Введите имя учетной записи");
  if (!accounts.hasOwnProperty(accountName)) {
    throw new Error(`Нет такой учетной записи: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

Функция `transfer` переводит некоторую сумму с одного счета на другой, в процессе запрашивая имя этого другого счета. Если задано неверное имя учетной записи, то `getAccount` выдает исключение.

Но `transfer` *сначала* снимает деньги со счета, а *затем* вызывает `getAccount`, прежде чем внести их на другой счет. Если в этот момент выполнение функции будет прервано исключением, то деньги просто исчезнут.

Данный код можно было бы написать немного более разумно — например, вызвав `getAccount`, прежде чем начинать перевод денег. Но такие проблемы

часто возникают в более тонких ситуациях. Даже функции, которые на вид не генерируют исключение, могут делать это в особо редких случаях или если в них содержится ошибка программиста.

Один из способов решения указанной проблемы — уменьшить количество используемых побочных эффектов. Здесь нам снова помогает стиль программирования, при котором вместо изменения существующих данных вычисляются новые значения. Если фрагмент кода перестает работать в процессе создания нового значения — не проблема, это наполовину созданное значение никто никогда не увидит.

Но подобное не всегда практично. Поэтому у инструкций `try` есть еще одна особенность. После них, в дополнение к блоку `catch` или вместо него, может стоять блок `finally`. Блок `finally` как бы говорит: «Что бы ни случилось, выполните этот код после попытки выполнить код, заключенный в блоке `try`».

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
      accounts[from] -= amount;
      progress = 1;
      accounts[getAccount()] += amount;
      progress = 2;
    } finally {
      if (progress == 1) {
        accounts[from] += amount;
      }
    }
}
```

Эта версия функции отслеживает ее выполнение, и, если при выходе окажется, что функция была прервана в момент, когда было создано несовместимое состояние программы, она исправляет нанесенный ущерб.

Обратите внимание, что, хотя код `finally` и выполняется, когда в блоке `try` создается исключение, это не мешает самому исключению. После выполнения блока `finally` стек продолжает раскручиваться.

Писать программы, которые продолжают надежно работать, несмотря на появление исключений в неожиданных местах, нелегко. Многие просто не заботятся об этом, и, поскольку исключения, как правило, предназначены для исключительных обстоятельств, проблема может возникать настолько

редко, что ее даже не замечают. Хорошо это или плохо, зависит от того, какой ущерб может причинить программное обеспечение в случае сбоя.

Выборочный перехват исключений

Если исключение без перехвата проходит до самого конца стека, оно обрабатывается средой выполнения. Что это означает, зависит от конкретной среды. В браузерах обычно появляется описание ошибки в консоли JavaScript (эта консоль доступна через меню Инструменты или Разработка). В Node.js, безбраузерной среде JavaScript, которую мы обсудим в главе 20, реализован более осторожный подход в отношении повреждения данных. Там при возникновении необработанного исключения прерывается весь процесс.

В случае ошибок программиста зачастую лучшее, что можно сделать, — это просто пропустить ошибку. Необработанное исключение — разумный способ сообщить о сломанной программе, а консоль JavaScript в современных браузерах предоставит отдельную информацию о том, какие вызовы функций находились в стеке при возникновении проблемы.

Для проблем, которые *могут* случиться при использовании программы в штатном режиме, сбой с необработанным исключением — ужасная стратегия.

Неправильное применение языка, такое как ссылка на несуществующую привязку, поиск свойства в `null` или вызов чего-то, что не является функцией, также приведет к возникновению исключений. Такие исключения тоже могут быть перехвачены.

Все, что мы знаем, создавая тело `catch`, — это то, что в нашем теле `try` *что-то* вызвало исключение. Но мы не знаем, *что именно* произошло или *какое* исключение оно вызвало.

JavaScript не обеспечивает (явное упущение!) непосредственную поддержку для выборочного перехвата исключений: вы либо перехватываете их все, либо не перехватываете ничего. Из-за этого возникает соблазн *предположить*, что полученное исключение — именно то, которое вы имели в виду, когда писали блок `catch`.

Но это может быть не так. Могут быть нарушены какие-то другие допущения, или вы сами могли ввести дефект, вызвавший исключение. Вот пример,

в котором *предпринята попытка* продолжить вызывать `promptDirection`, пока не будет получен корректный ответ:

```
for (;;) {
  try {
    let dir = promtDirection("Где?"); // ← typo!
    console.log("Перед вами ", dir);
    break;
  } catch (e) {
    console.log("Неверное направление. Попробуйте снова.");
  }
}
```

Конструкция `for (;;)` — это способ намеренно создать цикл, который сам по себе не завершится никогда. Принудительный выход из цикла произойдет только тогда, когда будет задано правильное направление. *Однако* мы неправильно написали `promptDirection`, что приведет к ошибке «неопределенная переменная». Поскольку в блоке `catch` значение исключения (`e`) полностью игнорируется, так как предполагается, что проблема известна, ошибка привязки ошибочно воспринимается как указание на неверный ввод. Это не только приводит к бесконечному циклу, но и скрывает полезное сообщение об ошибке с неправильной привязкой.

В общем случае не следует использовать исключения для комплексного перехвата, если только перехват не предназначен для перенаправления еще куда-нибудь — например, по сети, чтобы сообщить другой системе о сбое нашей программы. И даже в таком случае хорошо продумайте, как можно было бы случайно скрыть информацию.

Поэтому мы хотели бы перехватывать *конкретные* исключения. Мы можем это сделать, проверив в блоке `catch`, является ли полученное нами исключение именно тем, в котором мы заинтересованы, а если нет — то перебросить его дальше. Но как распознать исключение?

Мы могли бы сравнить его свойство `message` с сообщением об ошибке, которое мы ожидаем. Но это ненадежный способ написания кода: мы будем использовать информацию, предназначенную для людей (сообщение), чтобы принимать решение программно. Как только кто-нибудь изменит сообщение (или переведет его на другой язык), код перестанет работать.

Вместо этого определим новый тип ошибки и используем `instanceof` для ее идентификации.

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "Л";
  if (result.toLowerCase() == "right") return "П";
  throw new InputError("Неверное направление: " + result);
}
```

Новый класс ошибок является расширением `Error`. Он не определяет собственный конструктор, и это означает, что он наследует конструктор `Error`, который ожидает в качестве аргумента строковое сообщение. На самом деле в нем вообще ничего не определено — класс пуст. Объекты `InputError` ведут себя как объекты `Error`, за исключением того, что они принадлежат к другому классу, по которому их можно распознать.

Теперь цикл может перехватывать исключения более точно.

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("Перед вами ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Неверное направление. Попробуйте снова.");
    } else {
      throw e;
    }
  }
}
```

Теперь будут перехватываться только экземпляры `InputError`, а остальные несвязанные исключения будут пропускаться. Если снова допустить опечатку, то, как и должно быть, появится сообщение об ошибке неопределенной привязки.

Утверждения

Утверждения — это проверки внутри программы, которые удостоверяются, что нечто является тем, чем оно должно быть. Утверждения используются не для обработки ситуаций, способных возникнуть при нормальной работе, а для обнаружения ошибок программиста.

Если, например, `firstElement` описана как функция, которая никогда не должна вызываться для пустых массивов, ее можно было бы написать так:

```
function firstElement(array) {
  if (array.length == 0) {
    throw new Error("Вызов firstElement с []");
  }
  return array[0];
}
```

Теперь вместо того, чтобы тихо возвращать значение `undefined` (которое получается при чтении несуществующего свойства массива), функция будет громко взрывать программу при каждом неправильном использовании. Это снижает вероятность того, что подобные ошибки могут остаться незамеченными, и упрощает поиск причин их возникновения.

Я не рекомендую пытаться писать утверждения для всех возможных вариантов некорректных исходных данных. Это потребовало бы много работы и привело к слишком шумному коду. Лучше зарезервировать утверждения для ошибок, которые легко допустить (или которые вы частенько допускаете сами).

Резюме

Ошибки и некорректные исходные данные — часть жизни. Поиск, диагностика и исправление дефектов являются важным аспектом программирования. Чтобы было легче заметить проблемы, применяются автоматизированные тестовые наборы или утверждения, добавленные в программы.

Проблемы, вызванные не зависящими от программы факторами, обычно должны решаться изящно. Иногда, когда проблему можно решить локально, хорошим способом ее отследить являются специальные возвращаемые значения. В противном случае обычно предпочтительнее использовать исключения.

Генерирование исключения приводит к раскрутке стека вызовов до первого блока `try / catch`, перехватывающего это исключение, или же до дна стека. Значение исключения будет передано в перехватывающий его блок `catch`, который должен проверить, что это действительно ожидаемый тип исключения, а затем что-то с ним сделать. Чтобы справиться с непредсказуемой передачей управления, вызванной исключениями, можно использовать блоки `finally`, которые гарантируют, что заключенный в них фрагмент кода *всегда* будет выполнен после блока исключения.

Упражнения

Повторная попытка

Представьте, что у вас есть функция `primitiveMultiply`, которая в 20 % случаев умножает два числа, а в остальных 80 % случаев возникает исключение типа `MultiplicatorUnitFailure`. Напишите функцию, оборачивающую эту неуклюжую функцию и просто продолжающую попытки до тех пор, пока вызов не завершится успешно, после чего возвращающую результат.

Убедитесь, что вы обрабатываете только те исключения, которые рассчитываете обработать.

Запертый ящик

Рассмотрим следующий (довольно надуманный) объект:

```
const box = {
  locked: true,
  unlock() { this.locked = false; },
  lock() { this.locked = true; },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Заперто!");
    return this._content;
  }
};
```

Это ящик с замком. В ящике есть массив, но его можно получить, только если отпереть ящик. Прямой доступ к частному свойству `_content` запрещен.

Напишите функцию `withBoxUnlocked`, которая принимает в качестве аргумента функциональное значение, отпирает ящик, запускает функцию, а затем гарантирует, что прежде, чем завершить работу, ящик снова будет заперт независимо от того, возвратила функция-аргумент нормальный результат или вызвала исключение.

Если хотите заработать дополнительные баллы, убедитесь, что при вызове `withBoxUnlocked`, когда ящик уже открыт, он остается открытым.

9 Регулярные выражения

> Некоторые люди, столкнувшись с проблемой, думают: «О, а использую-ка я регулярные выражения». Теперь у них есть две проблемы.
>
> *Джейми Завински*

Инструменты и техники программирования выживают и распространяются в хаотичной, эволюционной манере. Побеждают не самые красивые или блестящие, а те, которые достаточно хорошо функционируют в нужной нише или оказываются интегрированными с другой успешной технологией.

В этой главе я расскажу об одном таком инструменте — *регулярных выражениях*. Это способ описания шаблонов в виде строковых данных. Регулярные выражения образуют небольшой самостоятельный язык, который является частью JavaScript и многих других языков и систем.

Регулярные выражения ужасно неудобны и чрезвычайно полезны. Их синтаксис загадочен, а программный интерфейс, предоставляемый для них в JavaScript, неуклюж. Но они представляют собой мощный инструмент для проверки и обработки строк. Хорошее понимание регулярных выражений сделает вас более эффективным программистом.

Создание регулярных выражений

Регулярное выражение — это тип объекта. Оно может быть создано с помощью конструктора `RegExp` или записано как литеральное значение, в котором шаблон заключен между символами косой черты (`/`).

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Оба этих объекта регулярных выражений реализуют один и тот же шаблон: символ `a`, за которым следует `b`, а за ним — `c`.

При использовании конструктора `RegExp` шаблон записывается как обычная строка, поэтому в ней для обратной косой черты действуют обычные правила.

Во второй нотации, где шаблон заключен между символами косой черты, обратная косая черта трактуется несколько иначе. Во-первых, поскольку косая черта — признак завершения шаблона, необходимо поставить обратную косую черту перед любой прямой косой чертой, которая является *частью* шаблона. Кроме того, обратные косые, которые не являются частью специальных кодов символов (таких как `\n`), будут *сохранены*, а не проигнорированы, поскольку находятся в строках, и будут изменять значение шаблона. Отдельные символы, такие как вопросительные знаки и знаки «плюс», в регулярных выражениях имеют специальные значения, и если они предназначены для представления самого символа, то перед ними нужно ставить обратную косую черту.

```
let eighteenPlus = /eighteen\+/;
```

Проверка на соответствия

У объектов регулярных выражений есть несколько методов. Самый простой из них — `test`. Если передать ему строку, то он вернет логическое значение, сообщающее, содержит ли данная строка шаблон, указанный в выражении.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

Регулярное выражение, состоящее только из нестандартных символов, представляет просто эту последовательность символов. Если `abc` встречается где-либо в строке, которую мы тестируем (не обязательно в начале), то `test` вернет `true`.

Множества символов

Чтобы выяснить, содержит ли строка фрагмент `abc`, можно также вызвать `indexOf`. Регулярные выражения позволяют выражать более сложные шаблоны.

Предположим, что мы хотим проверить, содержит ли срока какое-нибудь число. Если в регулярном выражении разместить несколько символов в квадратных скобках, то данная часть выражения будет соответствовать любому из символов, находящихся в квадратных скобках.

Оба следующих выражения соответствуют всем строкам, которые содержат цифры:

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

Тире (`-`) между двумя символами, заключенными в квадратные скобки, может использоваться для обозначения диапазона символов, порядок следования которых определяется номером символа в кодировке Unicode. Символы от 0 до 9 идут друг за другом в этом порядке (коды от 48 до 57), поэтому `[0-9]` охватывает их все и соответствует любой цифре.

Некоторые стандартные группы символов имеют собственные встроенные комбинации. Цифры являются одной из них: `\d` означает то же самое, что и `[0-9]`:

- `\d` — любая цифра;
- `\w` — любой алфавитно-цифровой символ (словообразующий символ);
- `\s` — любой пробельный символ (пробел, табуляция, новая строка и т. п.);
- `\D` — символ, который *не является* цифрой;
- `\W` — не алфавитно-цифровой символ;
- `\S` — не пробельный символ;
- `.` — любой символ, кроме новой строки.

Таким образом, формат даты и времени, такой как 01-30-2003 15:20, можно представить в виде следующего выражения:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("01-30-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

Выглядит чудовищно, не правда ли? Половина этих символов — обратные косые, создающие фоновый шум, из-за которого трудно понять настоящий

смысл шаблона. Позже мы познакомимся с несколько улучшенной версией данного выражения.

Эти коды с обратной косой чертой также могут использоваться в квадратных скобках. Например, `[\d.]` означает любую цифру или символ точки. Но сама точка, заключенная в квадратные скобки, теряет свое особое значение. То же касается других специальных символов, таких как `+`.

Чтобы *инвертировать* набор символов, то есть сказать, что вы хотите получить любой символ, кроме символов, входящих в данное множество, можно поставить после открывающей скобки символ вставки (`^`).

```
let notBinary = /[^01]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

Повторяющиеся части шаблона

Теперь мы знаем, как обозначить в шаблоне одну любую цифру. А что, если мы хотим обозначить целое число — последовательность из одной или нескольких цифр?

Если после чего-либо в регулярном выражении поставить знак «плюс» (`+`), это будет означать, что данный элемент может повторяться несколько раз. Так, `/\d+/` соответствует одной или нескольким цифрам.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

Звездочка (`*`) имеет аналогичное значение, но шаблон с ее использованием соответствует также нулю повторений. Если после чего-то стоит звездочка, данная часть шаблона не мешает соответствию — если не будет найден подходящий текст, то шаблон просто посчитает это нулевым количеством найденных соответствий.

Вопросительный знак делает часть шаблона *необязательной*, то есть она может встретиться один раз или не встретиться ни разу. В следующем примере допускается наличие символа u, но если его не будет, шаблон все равно считается соответствующим.

```
let neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

Чтобы показать, что фрагмент должен встречаться точное количество раз, применяются фигурные скобки. Например, если после элемента поставить {4}, то этот элемент должен повториться именно четыре раза. Можно также указать диапазон: {2,4} означает, что элемент должен повториться минимум дважды и максимум четыре раза.

Вот еще одна версия шаблона даты и времени, допускающая как однозначные, так и двузначные цифры для обозначения дней, месяцев и часов. Кроме того, ее немного легче расшифровать.

```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("1-30-2003 8:45"));
// → true
```

Кроме того, при использовании фигурных скобок можно указать открытые диапазоны, пропустив число после запятой. Например, {5,} означает повторение пять и более раз.

Группировка подвыражений

Чтобы использовать операторы типа * или + для нескольких повторений элемента, нужно задействовать скобки. Часть регулярного выражения, заключенная в круглые скобки, считается одним элементом по отношению к следующим за ней операторам.

```
let cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

Первый и второй знаки «плюс» применяются только ко второй букве o во фрагментах boo и hoo соответственно. Третий плюс применяется ко всей группе (hoo+), так что ей соответствует одна или несколько таких последовательностей.

Буква `i` в конце выражения в данном примере делает это регулярное выражение нечувствительным к регистру, так что ему подходит прописная буква `B` во входной строке, даже если сам шаблон написан строчными буквами.

Соответствия и группы

Метод `test` — это простейший способ проверить соответствие регулярного выражения. Данный метод сообщает нам только о том, обнаружено ли соответствие, и ничего больше. Для регулярных выражений также существует метод `exec` (от слова *execute* — «выполнить»), который возвращает `null`, если совпадение не найдено, и объект с информацией о совпадении в противном случае.

```
let match = /\d+/.exec("один два 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

Объект, который возвращает метод `exec`, имеет свойство `index`, сообщающее нам, *где именно* в строке найдено совпадение с шаблоном. Кроме этого, объект выглядит как массив строк (и фактически им является), первый элемент которого — строка, соответствующая шаблону. В предыдущем примере это искомая последовательность цифр.

Для строковых значений существует метод `match`, который ведет себя аналогичным образом.

```
console.log("один два 100".match(/\d+/));
// → ["100"]
```

Если регулярное выражение содержит подвыражения, заключенные в скобки, то текст, соответствующий этим группам, также будет представлен в массиве. Первым элементом массива всегда является полное совпадение. Следующий элемент — та часть, которая соответствует первой группе (открывающая скобка которой стоит первой в выражении), затем идет соответствие второй группе и т. д.

```
let quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

Если соответствий для группы вообще не найдено (например, когда после нее стоит вопросительный знак), в выходном массиве в позиции этой группы будет стоять `undefined`. Аналогично, если для группы найдено несколько соответствий, в массив попадает только последнее из них.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

Группы бывают полезны для извлечения частей строки. Если нам нужно не просто проверить, содержит ли строка определенную дату, но и извлечь эту дату и построить объект, который ее представляет, мы можем заключить шаблоны цифр в круглые скобки и напрямую извлечь дату из результата `exec`.

Но сначала мы сделаем небольшое отступление, в котором обсудим встроенный в JavaScript способ представления значений даты и времени.

Класс Date

В JavaScript существует стандартный класс для представления дат, или, точнее, моментов времени. Этот класс называется `Date`. Если просто создать объект даты, используя `new`, то получатся текущая дата и время.

```
console.log(new Date());
// → Mon Nov 13 2017 16:19:11 GMT+0100 (CET)
```

Можно также создать объект для определенного времени.

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

В JavaScript используется соглашение, согласно которому номера месяцев начинаются с нуля (поэтому декабрь имеет номер 11), а номера дней — с единицы. Это сбивает с толку и вообще глупо. Будьте внимательны.

Последние четыре аргумента (часы, минуты, секунды и миллисекунды) являются необязательными, и если не заданы, то считаются равными нулю.

Метки времени хранятся в виде количества миллисекунд с начала 1970 года в часовом поясе UTC — в соответствии с заключенным в то время

соглашением о Unix-времени. Для описания времени до 1970 года можно использовать отрицательные числа. Такое число возвращает метод `getTime` объекта даты. Как нетрудно догадаться, это большое число.

```
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 передать конструктору `Date` только один аргумент, он будет считаться временем в миллисекундах. Чтобы получить текущее значение времени в миллисекундах, нужно создать новый объект `Date` и вызвать для него `getTime` либо вызвать функцию `Date.now`.

Для извлечения компонентов даты объекты `Date` имеют такие методы, как `getFullYear`, `getMonth`, `getDate`, `getHours`, `getMinutes` и `getSeconds`. Помимо `getFullYear`, также есть метод `getYear`, который возвращает год минус 1900 (`98` или `119`) и, как правило, бесполезен.

Заключив интересующие нас части выражения в скобки, мы теперь можем создать объект даты из строки.

```
function getDate(string) {
  let [_, month, day, year] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(year, month - 1, day);
}
console.log(getDate("1-30-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

Привязка `_` (подчеркивание) игнорируется и используется только для того, чтобы пропустить элемент полного совпадения в массиве, возвращаемом методом `exec`.

Границы слов и строк

К сожалению, `getDate` так же радостно извлекает бессмысленную дату 00-1-3000 из строки `"100-1-30000"`. Соответствие может быть найдено в любом месте строки; в данном случае оно будет начинаться со второго символа и заканчиваться вторым символом с конца.

Для того чтобы принудительно потребовать совпадения всей строки, можно добавить маркеры `^` и `$`. Символ `^` соответствует началу входной строки,

а знак доллара — ее концу. Таким образом, `/^\d+$/` означает строку, полностью состоящую из одной или нескольких цифр; `/^!/` соответствует любой строке, которая начинается с восклицательного знака, а `/x^/` не соответствует ни одной строке (`x` не может стоять перед началом строки).

Если же мы лишь хотим гарантировать, что дата начинается и заканчивается на границе слова, то можно использовать маркер `\b`. Граница слова может быть началом или концом строки или находиться в любой другой точке строки, у которой с одной стороны стоит словообразующий символ (как в `\w`), а с другой — несловообразующий символ.

```
console.log(/кот/.test("антрекот"));
// → true
console.log(/\bкот\b/.test("антрекот"));
// → false
```

Обратите внимание, что маркер границы не соответствует какому-либо реальному символу. Он просто обеспечивает соответствие регулярного выражения только тогда, когда в том месте шаблона, где стоит этот символ, выполняется определенное условие.

Выбор шаблонов

Предположим, что мы хотим знать, содержит ли фрагмент текста не только число, но и число, за которым следует одно из слов: `pig`, `cow` или `chicken` — или эти же слова во множественном числе.

Мы могли бы написать три регулярных выражения и проверить их все по очереди, но есть способ получше. Символ вертикальной черты (`|`) обозначает выбор между шаблонами, расположенными слева и справа от него. Так что можно написать так:

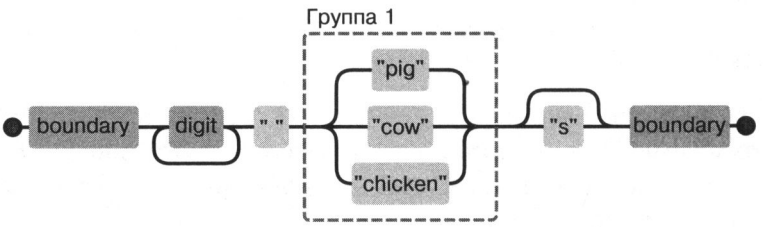
```
let animalCount = /\b\d+ (pig|cow|chicken)s?\b/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

Чтобы ограничить часть шаблона, к которой должен применяться оператор `|`, можно заключить ее в скобки; для описания выбора из более чем двух альтернатив можно использовать несколько таких операторов последовательно.

Механизм поиска соответствия

Концептуально при использовании `exec` или `test` механизм регулярных выражений ищет совпадения в заданной строке, пытаясь сопоставить выражение с начала строки, затем со второго символа и т. д., пока не будет найдено совпадение или не будет достигнут конец строки. Механизм либо вернет первое найденное совпадение, либо не найдет совпадения вообще.

На практике, чтобы выполнить такое сопоставление, регулярное выражение рассматривается как блок-схема. Ниже представлена схема регулярного выражения для рассмотренного ранее примера с животными.



Наше регулярное выражение находит соответствие, если удается найти путь от левой части схемы к правой. Мы запоминаем текущую позицию в строке и каждый раз, когда проходим через блок, проверяем, соответствует ли этому блоку часть строки после текущей позиции.

Таким образом, если попытаться сопоставить с шаблоном строку `"the 3 pigs"` из позиции 4, то продвижение по блок-схеме будет выглядеть следующим образом.

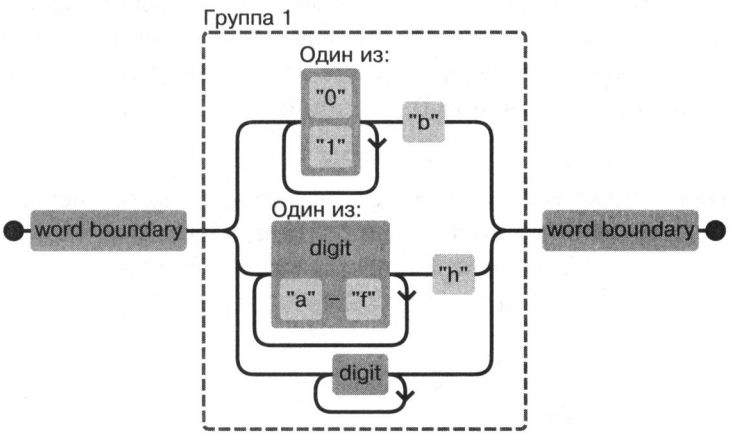
- В позиции 4 есть граница слова, поэтому первый блок пройден.
- Там же, в позиции 4, находим цифру, поэтому второй блок пройден.
- В позиции 5 один путь возвращается к предыдущему (второму) блоку, а второй ведет вперед через блок, содержащий один символ. У нас здесь стоит пробел, а не цифра, поэтому мы должны выбрать второй путь.
- Теперь мы находимся в позиции 6 (начало слова `pigs`) и перед тройным разветвлением на схеме. Варианты `cow` и `chicken` не подходят, но зато подходит `pig`, поэтому выбираем данную ветвь.
- В позиции 9 после ветвления один путь пропускает блок `s` и идет прямо к конечной границе слова, тогда как другой путь соответствует букве `s`.

В нашей строке в этом месте находится символ `s`, а не граница слова, поэтому проходим через блок `s`.

- Теперь мы находимся в позиции 10 (конец строки) и можем сопоставить только границу слова. Конец строки считается границей слова, поэтому мы проходим последний блок и получаем успешное соответствие строки шаблону.

Поиск с возвратом

Регулярное выражение `/\b([01]+b|[\da-f]+h|\d+)\b/` соответствует двоичному числу, после которого идет буква `b`, или шестнадцатеричному числу (то есть с основанием 16, с буквами от `a` до `f`, обозначающими цифры от 10 до 15), после которого стоит буква `h`, или обычному десятичному числу без суффиксного символа. Вот соответствующая схема:



При сопоставлении этого выражения часто случается, что верхняя (двоичная) ветвь выбирается даже в том случае, если на самом деле входные данные не содержат двоичное число. Например, при сопоставлении строки `"103"` только в точке 3 становится ясно, что мы выбрали не ту ветвь. Строка *соответствует* регулярному выражению, но не той ветке, в которой мы находимся.

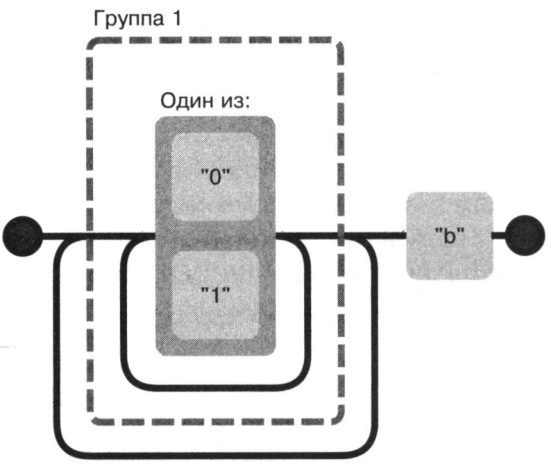
В таких случаях механизм поиска соответствия выполняет *возврат*. При входе в ветку она запоминает текущую позицию (в данном случае в начале строки, сразу после первого блока на схеме, обозначающего границу слова),

чтобы можно было вернуться и попробовать другую ветку, если пройти текущую не удалось. Для строки `"103"`, дойдя до цифры 3, механизм поиска попробует пройти ветвь для шестнадцатеричных чисел, что снова не удастся, потому что после числа нет буквы `h`. Тогда он попытается пройти ветвь десятичного числа. Это удастся, и в итоге механизм поиска сообщит о том, что соответствие найдено.

Поиск останавливается, как только находит полное совпадение. Это означает, что, если несколько ветвей потенциально могут соответствовать строке, используется только первая из них (в том порядке, в котором ветви идут в регулярном выражении).

Возврат относится также к операторам повторения, таким как `+` и `*`. Если сравнить `/^.*x/` и `"abcxe"`, то часть `.*` сначала пытается захватить всю строку. Затем механизм поиска понимает, что для соответствия шаблону ему нужен `x`. Поскольку после конца строки `x` отсутствует, оператор `*` пытается найти соответствие на один символ меньше. Но механизм поиска не находит `x` и после `abcx`, поэтому он снова возвращается, сопоставляя оператор `*` только с `abc`. *Только теперь* он находит `x` там, где нужно, и сообщает об успешно найденном соответствии с позиции от 0 до 4.

Можно написать регулярные выражения, которые будут делать *много* возвратов. Эта проблема возникает тогда, когда шаблон может соответствовать исходному фрагменту различными способами. Например, если бы мы запутались при написании регулярного выражения для двоичного числа, то могли бы случайно написать что-то вроде `/([01]+)+b/`.



Если механизм поиска попытается сопоставить несколько длинных серий нулей и единиц без завершающего символа b, то он сначала будет проходить через внутренний цикл, пока не закончатся цифры. Затем он заметит, что в конце нет буквы b, и возвратится на одну позицию, один раз пройдет через внешний цикл и снова сдастся, пытаясь выйти из внутреннего цикла. Он продолжит перебирать все возможные маршруты через эти две петли. Подобное означает, что объем работы удваивается с каждым новым символом. Даже для нескольких десятков символов поиск соответствий займет вечность.

Метод replace

У строковых значений есть метод `replace`, который можно использовать для замены части строки на другую строку.

```
console.log("papa".replace("p", "m"));
// → mapa
```

Первый аргумент может быть регулярным выражением, и тогда заменяется первое соответствие регулярного выражения. Если добавить к регулярному выражению опцию `g` (от слова *global*), то будут заменены все соответствия в строке, а не только первое из них.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

Было бы разумно, если бы выбор между заменой одного или всех соответствий определялся дополнительным аргументом метода `replace` или если бы существовал отдельный метод `replaceAll`. Но по какой-то несчастливой причине вместо этого выбор определяется свойством регулярного выражения.

По-настоящему возможности использования регулярных выражений в `replace` проявляются в том, что с их помощью можно ссылаться на совпадающие группы в строке замены. Например, предположим, что у нас есть длинная строка, содержащая имена людей, по одному имени в строке в формате `Lastname, Firstname` (фамилия, имя). Мы хотим поменять местами имя и фамилию и удалить запятую между ними, чтобы получить формат `Firstname Lastname`. Для этого можно применить следующий код:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nWadler, Philip"
```

```
    .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Philip Wadler
```

`$1` и `$2` в строке замены относятся к группам, которые в шаблоне заключены в скобки. `$1` заменяется текстом, соответствующим первой группе, `$2` — второй и так далее до `$9`. Для того чтобы сослаться на полное совпадение, используется комбинация `$&`.

Вместо строки в качестве второго аргумента можно передать функцию. Тогда при каждой замене будет вызываться эта функция, которой в качестве аргументов будут передаваться найденные соответствия для групп (а также полное соответствие), а ее возвращаемое значение будет вставлено в новую строку.

Вот небольшой пример:

```
let s = "the cia and fbi";
console.log(s.replace(/\b(fbi|cia)\b/g,
            str => str.toUpperCase()));
// → the CIA and FBI
```

Вот более интересный пример:

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) { // остался только один, убрать 's'
    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
```

Эта функция принимает строку, находит все вхождения числа, за которым следует алфавитно-цифровое слово, и возвращает строку, где каждое такое вхождение уменьшается на единицу.

Группа `(\d+)` передается функции как аргумент функции `amount`, а группа `(\w+)` — как аргумент `unit`. Функция преобразует `amount` в число — что всегда возможно, поскольку оно соответствует шаблону `\d+`, — и вносит некоторые коррективы, если остается только один элемент или ноль.

О жадности

Используя метод `replace`, можно написать функцию, которая бы удаляла все комментарии из фрагмента кода JavaScript. Вот первая попытка написать такую функцию:

```
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
```

То, что стоит перед оператором ИЛИ, соответствует двум символам косой черты, за которыми следует любое количество символов, не являющихся символами новой строки. Фрагмент, описывающий многострочные комментарии, более сложен. Мы применили код `[^]` (любой символ, которого нет в пустом наборе символов), чтобы обозначить любой символ. Здесь мы не можем просто использовать точку, потому что блочные комментарии могут продолжаться на новой строке, а символ точки не соответствует символу новой строки.

Но результат для последней строки, похоже, неверен. Почему?

Фрагмент `[^]*` в регулярном выражении, как было описано в разделе о поиске с возвратами, сначала захватит максимально возможную часть строки. Если это приведет к несоответствию следующей части шаблона, то механизм поиска соответствий переместится назад на один символ и предпримет новую попытку. В этом примере механизм поиска соответствий сначала пытается сопоставить всю оставшуюся часть строки, а затем возвращается назад. Вернувшись на четыре символа назад, он обнаружит вхождение `*/` и посчитает его соответствием. Но это не то, что мы хотели, — мы хотели найти отдельный комментарий, а не дойти до конца кода и обнаружить конец последнего блочного комментария.

Из-за такого поведения операторы повторения (`+`, `*`, `?` и `{}`) называют *жадными*, имея в виду, что они захватывают столько, сколько могут, а затем выполняют возврат. Если после них поставить вопросительный знак (`+?`, `*?`, `??`, `{}?`), то они перестанут быть жадными и начнут с сопоставления как

можно меньшей части строки, забирая больше только в том случае, если оставшийся шаблон не соответствует меньшему фрагменту.

И это именно то, что мы хотим в данном случае. Если звездочка соответствует наименьшему фрагменту строки, после которого идет `*/`, то мы выбираем один блочный комментарий и ничего больше.

```
function stripComments(code) {
  return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

Многие дефекты при программировании регулярных выражений могут быть вызваны непреднамеренным использованием жадных операторов там, где нежадный оператор работал бы более эффективно. При применении оператора повтора вначале проверяйте нежадный вариант.

Динамическое создание объектов RegExp

Бывает так, что точный шаблон, с которым нужно сопоставлять строку, при написании кода неизвестен. Предположим, вы хотите найти имя пользователя во фрагменте текста и заключить это имя между символами подчеркивания, чтобы выделить его. Поскольку имя будет известно только после запуска программы, использовать нотацию с косыми не удастся.

Но можно построить строку и затем использовать конструктор `RegExp`, например:

```
let name = "гарри";
let text = "Гарри — подозрительный персонаж.";
let regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Гарри_ - подозрительный персонаж.
```

При создании маркеров границы `\b` нам пришлось использовать две обратные косые черты, потому что это обычная строка, а не регулярное выражение, заключенное между двумя косыми. Второй аргумент конструктора `RegExp` содержит параметры регулярного выражения — в данном случае `"gi"`, что означает глобальную замену *(global)* без учета регистра *(case insensitive)*.

Но что, если окажется, что наш пользователь — подросток-зануда и он введет в качестве имени `"dea+hl[]rd"`? Это приведет к бессмысленному

регулярному выражению, которое не будет соответствовать имени пользователя.

Чтобы исключить такую возможность, можно добавить обратную косую черту перед любым символом, который имеет особое значение.

```
let name = "dea+hl[]rd";
let text = "Этот dea+hl[]rd раздражает сверх меры.";
let escaped = name.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("\\b" + escaped + "\\b", "gi");
console.log(text.replace(regexp, "_$&_"));
// → Этот _dea+hl[]rd_ раздражает сверх меры.
```

Метод search

Метод `indexOf` предназначен для строк, и его нельзя использовать с регулярными выражениями. Но есть другой метод, `search`, аргументом которого может быть регулярное выражение. Как и `indexOf`, он возвращает первый индекс, по которому было найдено выражение, или –1, если оно не было найдено.

```
console.log("    слово".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

К сожалению, не существует способа указать, что соответствие следует искать с заданным смещением (как это делается с помощью второго аргумента `indexOf`), что часто бывает полезно.

Свойство lastIndex

Метод `exec`, как и `search`, не предоставляет удобного способа начать поиск с заданной позиции в строке. Зато он предоставляет *неудобный* способ это сделать.

У объектов регулярных выражений есть свойства. Одним из них является `source`, где содержится строка, на основе которой было создано выражение. Другим свойством выступает `lastIndex`, в отдельных ограниченных обстоятельствах определяющее начало поиска следующего соответствия.

Эти обстоятельства таковы: в регулярном выражении должны присутствовать параметры глобального поиска (`g`) или «приклеивания» (`y`), а сопоставление

должно выполняться с помощью метода exec. Опять же было бы проще передать методу exec дополнительный аргумент, но путаница является неотъемлемым свойством интерфейса регулярных выражений JavaScript.

```
let pattern = /y/g;
pattern.lastIndex = 3;
let match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

Если поиск был успешен, то вызов exec автоматически обновляет свойство lastIndex, чтобы оно указывало на следующую позицию после найденного соответствия. Если же совпадений не было найдено, то lastIndex присваивается ноль — что также является значением, которое этот параметр имеет во вновь созданном объекте регулярного выражения.

Разница между параметрами глобального поиска и «приклеивания» заключается в том, что в режиме «приклеивания» поиск соответствий будет вестись, только начиная непосредственно с lastIndex, тогда как при глобальном поиске поиск будет считаться успешным для любого найденного соответствия.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

При использовании одного и того же регулярного выражения для нескольких вызовов exec эти автоматические изменения свойства lastIndex могут вызвать проблемы. Регулярное выражение может случайно начинаться с индекса, оставшегося после предыдущего вызова.

```
let digit = /\d/g;
console.log(digit.exec("вот единица: 1"));
// → ["1"]
console.log(digit.exec("а теперь: 1"));
// → null
```

Другой интересный эффект с параметром глобального поиска заключается в том, что он меняет способ работы метода match для строк. При вызове этого метода для глобального поиска соответствий вместо массива, подобного

тому, который возвращается exec, match найдет *все* совпадения шаблона в строке и вернет массив, содержащий соответствующие строки.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

Так что будьте осторожны с глобальными регулярными выражениями. Обычно они необходимы только при вызовах replace и там, где вы хотите явно использовать lastIndex.

Циклический поиск соответствий

Часто требуется найти в строке все соответствия шаблону таким образом, чтобы получить доступ к объекту соответствия в теле цикла. Для этого можно использовать lastIndex и exec.

```
let input = "Строка с 3 числами... 42 и 88.";
let number = /\b\d+\b/g;
let match;
while (match = number.exec(input)) {
  console.log("Найдено число", match[0], "в позиции", match.index);
}
// Найдено число 3 в позиции 14
// Найдено число 42 в позиции 33
// Найдено число 88 в позиции 40
```

Здесь использован тот факт, что значением выражения присваивания (=) является присвоенное значение. Таким образом, применяя match = number.exec(input) в качестве условия в инструкции while, мы выполняем поиск соответствий в начале каждой итерации, сохраняем его результат в привязке и прекращаем цикл, когда совпадений больше не найдено.

Анализ INI-файла

В заключение главы рассмотрим проблему, для решения которой требуются регулярные выражения. Предположим, что мы пишем программу для автоматического сбора информации о наших врагах в Интернете. (На самом деле мы не будем писать эту программу целиком, только ту часть, которая читает файл конфигурации. Уж простите.) Файл конфигурации выглядит следующим образом:

```
searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7

; строка комментария начинается с точки с запятой...
; каждый раздел касается отдельного врага
[larry]
fullname=Ларри Доу
type=хулиган в детском саду
website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
fullname=Девеорн
type=злой волшебник
outputdir=/home/marijn/enemies/davaeorn
```

Точные правила для этого формата (очень популярного, обычно называемого INI-файлом) таковы.

- Пустые строки и строки, начинающиеся с точки с запятой, игнорируются.
- Строки, заключенные в квадратные скобки [ ], открывают новый раздел.
- Строки, содержащие алфавитно-цифровой идентификатор, после которого стоит знак =, добавляют параметр в текущий раздел.
- Любые другие строки являются некорректными.

Наша задача — преобразовать подобную строку в объект, свойства которого содержат строки для параметров, записанных перед первым заголовком раздела, и подобъектов для разделов, причем каждый подобъект содержит параметры соответствующего раздела.

Поскольку формат должен обрабатываться построчно, было бы логично начать с разделения файла на строки. В главе 4 мы узнали о методе `split`. Однако в отдельных операционных системах для разделения строк используется не просто символ новой строки, а символ возврата каретки, за которым следует символ новой строки (`"\r\n"`). Учитывая, что метод `split` также допускает применение регулярного выражения в качестве аргумента, мы можем задействовать для разделения строк регулярное выражение типа `/\r?\n/`, допускающее наличие между строками как `"\n"`, так и `"\r\n"`.

```
function parseINI(string) {
  // Начинаем с объекта для хранения полей верхнего уровня
  let result = {};
  let section = result;
```

```
  string.split(/\r?\n/).forEach(line => {
    let match;
    if (match = line.match(/^(\w+)=(.*)$/)) {
      section[match[1]] = match[2];
    } else if (match = line.match(/^\[(.*)\]$/)) {
      section = result[match[1]] = {};
    } else if (!/^\s*(;.*)?$/.test(line)) {
      throw new Error("Строка '" + line + "' некорректна.");
    }
  });
  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

Этот код проходит по всем строкам файла и создает объект. Свойства верхнего уровня хранятся непосредственно в данном объекте, а свойства, найденные в разделах, — в отдельных объектах разделов. Привязка `section` указывает на объект для текущего раздела.

Существует два вида значимых строк — заголовки разделов и строки свойств. Если строка является обычным свойством, то оно сохраняется в текущем разделе. Если же строка — заголовок раздела, то создается новый объект раздела, и привязка `section` устанавливается так, чтобы указывать на этот раздел.

Обратите внимание на повторяющееся использование символов `^` и `$`, позволяющих убедиться, что выражение соответствует всей строке, а не только ее части. Если их убрать, то получим код, в основном работающий для некоторых вариантов ввода, но ведущий себя странно, и этот дефект может быть трудно отследить.

Шаблон `if (match = string.match(...))` аналогичен приему с использованием присваивания в качестве условия для `while`. Мы не всегда уверены, что запрос на совпадение будет успешным, поэтому можем получить доступ к полученному объекту только внутри оператора `if`, который проверяет это. Чтобы не нарушать приятную цепочку форм `else if`, мы присваиваем результат сопоставления привязке и сразу же используем его в качестве условия в операторе `if`.

Если строка не заголовок раздела или свойство, то функция проверяет, является она комментарием или пустой строкой, применяя выражение `/^\s*(;.*)?$/`. Видите, как это работает? Часть в скобках будет соответствовать комментариям, а вопросительный знак позволяет проверить на соответствие строкам, содержащим только пробел. Если строка не соответствует ни одному из ожидаемых форматов, функция выдает исключение.

Интернациональные символы

Из-за первоначальной упрощенной реализации JavaScript и того факта, что этот упрощенный подход впоследствии стал основой стандартного поведения, регулярные выражения JavaScript весьма беспомощны в отношении символов, не относящихся к английскому языку. Например, в регулярных выражениях JavaScript «словообразующий символ» может быть только одним из 26 символов латинского алфавита (верхнего или нижнего регистра), десятичной цифрой или, по некоторым причинам, символом подчеркивания. Такие знаки, как *é* или *ß*, определенно являющиеся словообразующими символами, не будут соответствовать `\w` (и будут соответствовать варианту с заглавной буквой `\W`, то есть считаться несловообразующим символом).

По странной исторической случайности у комбинации `\s` (пробельный символ) нет подобной проблемы, она соответствует всем символам, которые стандарт Unicode рассматривает как пробельный, включая такие вещи, как неразрывный пробел и монгольский разделитель гласных.

Еще одна проблема заключается в том, что, как обсуждалось в главе 5, по умолчанию регулярные выражения работают не с реальными символами, а с кодовыми единицами. Это означает, что символы, состоящие из двух кодовых единиц, будут вести себя странно.

```
console.log(/🍎{3}/.test("🍎🍎🍎"));
// → false
console.log(/<.>/.test("<🌹>"));
// → false
console.log(/<.>/u.test("<🌹>"));
// → true
```

Проблема в том, что символ 🍎 в первой строке рассматривается как две кодовые единицы и часть `{3}` применяется только ко второй из них. Ана-

логично точка соответствует одной кодовой единице, а не двум, которые образуют эмоджи в виде розочки.

Для того чтобы регулярное выражение правильно обрабатывало такие символы, в него нужно добавить опцию `u` (от слова Unicode). К сожалению, по умолчанию выбирается неправильное поведение, потому что изменение поведения может вызвать проблемы в уже существующем, зависящем от него коде.

Несмотря на то что это только недавно было стандартизировано и на момент написания данной книги еще не получило широкой поддержки, уже можно использовать в регулярных выражениях параметр `\p` (требующий включения режима Unicode), которому соответствуют все символы, имеющие заданное свойство в стандарте Unicode.

```
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

Unicode определяет ряд полезных свойств, хотя найти то, что нужно, иногда бывает нелегко. Используя нотацию `\p{свойство=значение}`, можно найти любые символы, у которых данное свойство имеет данное значение. Если свойство не указано, как в `\p{имя}`, то предполагается, что `имя` является либо двоичным свойством, таким как `Alphabetic`, либо категорией, такой как `Number`.

Резюме

Регулярные выражения — это объекты, которые представляют собой шаблоны в виде строк. Для описания таких шаблонов используется специальный язык:

- `/abc/` — последовательность символов;
- `/[abc]/` — любой символ из данного набора;
- `/[^abc]/` — любой символ, *не входящий* в данный набор;

- `/[0-9]/` — любой символ из данного диапазона;
- `/x+/` — одно или несколько вхождений шаблона `x`;
- `/x+?/` — одно или несколько вхождений шаблона, нежадный вариант;
- `/x*/` — ноль и более вхождений;
- `/x?/` — ноль или одно вхождение;
- `/x{2,4}/` — от двух до четырех вхождений;
- `/(abc)/` — группа;
- `/a|b|c/` — любой из нескольких шаблонов;
- `/\d/` — любой цифровой символ;
- `/\w/` — алфавитно-цифровой символ (словообразующий символ);
- `/\s/` — любой пробельный символ;
- `/./` — любой символ, кроме перевода строки;
- `/\b/` — граница слова;
- `/^/` — начало ввода;
- `/$/` — конец ввода.

У регулярных выражений есть метод `test`, позволяющий проверить, соответствует ли данная строка данному выражению. Существует также метод `exec`, который при обнаружении соответствия возвращает массив, содержащий все найденные группы. У этого массива есть свойство `index`, указывающее, с какой позиции начинается соответствие.

У строк есть метод `match`, позволяющий сопоставить строку с регулярным выражением, и метод `search`, помогающий найти соответствие и возвращающий только начальную позицию совпадения. Метод `replace` позволяет заменить в строке совпадения с шаблоном на строку или выполнить для них заданную функцию.

Регулярные выражения могут иметь параметры, которые ставятся после закрывающей косой черты. Параметр `i` дает шаблону нечувствительность к регистру. Параметр `g` делает выражение *глобальным*, что, помимо прочего, заставляет метод `replace` заменить все найденные соответствия, а не только первое из них. Параметр `y` делает выражение «приклеенным», что означает, что после найденного первого соответствия дальнейшая часть строки будет пропущена — поиск в оставшейся части строки производиться не будет.

Параметр u включает режим Unicode, который устраняет ряд проблем, связанных с обработкой символов, занимающих две кодовые единицы.

Регулярные выражения напоминают острый инструмент с неудобной ручкой. Они сильно упрощают решение некоторых задач, но легко могут стать неуправляемыми в применении к сложным проблемам. Отчасти знания об их использовании призваны оградить вас от стремления применять регулярные выражения там, где с их помощью нельзя четко описать проблему.

Упражнения

В ходе работы над этими упражнениями вы почти наверняка придете в замешательство и испытаете разочарование из-за необъяснимого поведения некоторых регулярных выражений. Иногда помогает анализ таких выражений с помощью онлайн-инструментов, таких как https://debuggex.com, где можно увидеть, соответствует ли визуализация тому, что вы хотели, и поэкспериментировать с реакцией регулярного выражения на различные входные строки.

Regexp-гольф. Термином *«кодовый гольф»* называют игру, цель которой — описать определенную программу как можно меньшим количеством символов. Аналогично *regexp-гольф* — это практика создания как можно более коротких регулярных выражений, описывающих данный шаблон и только его.

Для каждого из следующих элементов напишите регулярное выражение, позволяющее проверить, встречается ли в строке какая-либо из указанных подстрок. Регулярное выражение должно соответствовать только строкам, содержащим одну из описанных подстрок. Не беспокойтесь о границах слов, если явно не указано иное. Когда ваше выражение сработает, проверьте, нельзя ли сделать его еще короче.

1. `car` и `cat`.
2. `pop` и `prop`.
3. `ferret`, `ferry` и `ferrari`.
4. Любое слово, оканчивающееся на `ious`.
5. Пробельный символ, за которым следуют точка, запятая, двоеточие или точка с запятой.

6. Слово длиннее шести букв.
7. Слово без буквы *e* (или *E*).

В качестве справочного пособия используйте список в разделе «Резюме» этой главы. Проверьте каждое решение с помощью нескольких тестовых строк.

Стиль цитирования

Представьте, что вы написали рассказ и использовали одинарные кавычки для обозначения прямой речи. Теперь вы хотите заменить все кавычки в диалогах двойными кавычками, сохраняя при этом одинарные кавычки, примененные в качестве апострофов, как в слове *Д'Артаньян*.

Подумайте, как сделать шаблон, который бы различал эти два вида использования кавычек, и создайте вызов метода `replace`, который бы выполнял правильную замену.

Снова числа

Напишите выражение, которое выбирало бы только числа в стиле JavaScript. В нем должен учитываться необязательный знак «минус» или «плюс» перед числом, десятичная точка и обозначение показателя степени, как в `5e-3` или `1E10`, также с необязательным знаком перед показателем степени. Кроме того, обратите внимание, что в числе могут отсутствовать цифры перед точкой или после нее, но число не может представлять собой только одну точку. Другими словами, `.5` и `5.` представляют собой правильную запись чисел в JavaScript, а одиночная точка — нет.

10 Модули

> Пишите код, который легко удалить,
> но трудно растянуть.
>
> *Tef. Программирование — ужасная вещь*

Идеальная программа имеет кристально ясную структуру. Можно легко объяснить, как она работает, и роль каждой ее части четко определена.

Типичная реальная программа растет как дерево. По мере того как появляются новые потребности, добавляются функциональные возможности. Структурирование — и сохранение структуры — дополнительная работа, которая окупится только в будущем, когда кто-нибудь в *очередной* раз будет работать над программой. А пока есть сильный соблазн пренебречь этим и позволить частям программы сильно запутаться.

На практике подобное приводит к двум проблемам. Во-первых, понять такую систему непросто. Если любые изменения могут затронуть все остальное, трудно рассмотреть какой-то фрагмент в отдельности. Приходится разбираться во всей программе целиком. Во-вторых, если вы захотите использовать какую-либо функциональность такой программы в другой ситуации, может оказаться, что ее проще переписать заново, чем пытаться отделить от контекста.

Подобные большие, бесструктурные программы часто описывают фразой «большой ком грязи»: все слиплось, и стоит попытаться выделить что-то одно, как оно разваливается и вы стоите с грязными руками.

Зачем нужны модули

Модули дают возможность избежать подобных проблем. Модулем называется часть программы, которая определяет, на какие другие компоненты она

опирается и какие функциональные возможности предоставляет другим модулям (*интерфейс* модуля).

Интерфейсы модулей имеют много общего с интерфейсами объектов, описанными в главе 6. Интерфейс делает часть модуля доступной для внешнего мира и закрывает остальное. Ограничивая способы взаимодействия модулей между собой, система превращается в подобие конструктора «Лего», части которого соединяются посредством четко определенных разъемов, и становится уже менее похожей на кусок грязи, где все смешалось в один большой ком.

Отношения между модулями называются *зависимостями*. Когда модулю требуется фрагмент из другого модуля, говорят, что он зависит от этого модуля. Когда данный факт четко описан в самом модуле, его можно использовать с целью выяснить, какие другие модули необходимы для того, чтобы можно было применять данный модуль и автоматически загружать зависимости.

Для того чтобы разделить модули подобным образом, каждому из них нужна своя собственная область видимости.

Простое размещение кода JavaScript в разных файлах не удовлетворяет этим требованиям. Файлы по-прежнему используют общее глобальное пространство имен. Они могут, намеренно или случайно, изменять привязки друг друга. Остается также неясной структура зависимостей. Как мы увидим в данной главе, это можно улучшить.

Разработать подходящую структуру модулей программы бывает непросто. На этапе предварительного исследования проблемы, подбирая для нее различные варианты решения, об этом можно особо не беспокоиться, поскольку это может сильно отвлекать. Но, когда у вас уже есть что-то, что представляется надежным решением, самое время сделать шаг назад и привести это в порядок.

Пакеты

Одним из преимуществ построения программы из отдельных частей и возможности запускать их по отдельности является то, что вы можете использовать один и тот же фрагмент в разных программах.

Но как это сделать? Предположим, я хочу применить функцию `parseINI` из главы 9 в другой программе. Если известно, от чего зависит эта функция

(в данном случае ни от чего), то я могу просто скопировать весь необходимый код в мой новый проект и использовать его. Но потом, если я найду ошибку в этом коде, я, вероятно, исправлю ее в той программе, с которой сейчас работаю, но забуду исправить ее в другой программе.

Как только вы начнете дублировать код, вы быстро обнаружите, что тратите время и силы на перемещение копий и поддержание их в актуальном состоянии.

Именно здесь нам помогут *пакеты*. Пакет — это кусок кода, который можно распространять (копировать и устанавливать). Пакет может содержать один или несколько модулей, а также информацию о том, от каких других пакетов он зависит. Пакет обычно поставляется в комплекте с документацией, объясняющей, что он делает, чтобы тот, кто его не писал, мог его использовать.

Если в пакете обнаружена проблема или добавляется новая функция, пакет обновляется. После этого программы, которые от него зависят (и которые также могут быть пакетами), могут обновиться до новой версии.

Такой стиль работы требует инфраструктуры. Нам нужно место для хранения и поиска пакетов, а также удобный способ их установки и обновления. В мире JavaScript данная инфраструктура предоставляется NPM (https://www.npmjs.com/).

NPM — это два в одном: онлайн-сервис, откуда можно загружать (и где можно размещать) пакеты, и программа (поставляется в комплекте с Node.js), которая помогает устанавливать пакеты и управлять ими.

На момент написания этой книги в NPM было доступно более полумиллиона различных пакетов. Следует признать, что большая часть из них — просто мусор. Но почти все полезные, общедоступные пакеты там также можно найти. Например, в пакете с именем `ini` есть синтаксический анализатор INI-файлов, аналогичный тому, который мы создали в главе 9.

В главе 20 вы узнаете, как устанавливать такие пакеты локально с помощью программы командной строки `npm`.

Наличие качественных пакетов, доступных для скачивания, чрезвычайно важно. Это зачастую означает возможность не переписывать заново программу, которую уже написали до нас 100 человек, вместо чего получить надежную, проверенную реализацию, нажав всего нескольких клавиш.

Программное обеспечение легко копируется, поэтому распространение того, что кто-то уже написал ранее, среди других людей является эффективным процессом. Но написать исходную программу — *немалый* труд, а реагировать на сообщения от людей, которые обнаружили проблемы в коде или хотят предложить новые функции, — труд еще больший.

По умолчанию авторские права на написанный вами код принадлежат вам, и другие люди могут использовать его только с вашего разрешения. Но, поскольку существуют просто щедрые люди, а публикация хорошего программного обеспечения способна сделать вас немного популярнее среди программистов, многие пакеты публикуются по лицензии, которая явно позволяет другим применять эти пакеты.

Именно так лицензируется бóльшая часть кода на NPM. Некоторые лицензии требуют, чтобы вы тоже публиковали код, созданный на основе пакета, под той же лицензией. Другие менее требовательны, они лишь требуют сохранять лицензию вместе с кодом при его распространении. Сообщество JavaScript в основном использует последний тип лицензии. При применении пакетов других разработчиков ознакомьтесь с их лицензией.

Импровизированные модули

До 2015 года в языке JavaScript не было встроенной системы модулей. Тем не менее к тому времени программисты уже более десяти лет создавали большие системы на JavaScript, и им были *необходимы* модули.

Поэтому они разработали собственные модульные системы как надстройку над языком. Вы можете использовать функции JavaScript для создания локальных областей видимости и объектов, имитирующих интерфейсы модулей.

Ниже показан модуль для преобразования названий дней недели в их номера (как возвращает метод `Date getDay`). Его интерфейс состоит из `weekDay.name` и `weekDay.number`, а локальная привязка `names` скрыта внутри области видимости функции, которая вызывается немедленно.

```
const weekDay = function() {
  const names = ["Понедельник", "Вторник", "Среда", "Четверг",
                 "Пятница", "Суббота", "Воскресенье"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
```

```
}());

console.log(weekDay.name(weekDay.number("Воскресенье")));
// → Воскресенье
```

Такой стиль модулей в определенной степени обеспечивает изоляцию, но не объявляет зависимости. Вместо чего он просто размещает свой интерфейс в глобальной области и ожидает, что его зависимости, если таковые имеются, будут поступать так же. Долгое время данный подход был основным в веб-программировании, но сейчас он практически полностью устарел.

Если мы хотим сделать отношения зависимостей частью кода, то мы должны взять загрузку зависимостей под контроль. Для этого необходимо уметь выполнять строки как код. JavaScript позволяет это сделать.

Выполнение данных как кода

Существует несколько способов взять данные (строку кода) и выполнить их как часть текущей программы.

Наиболее очевидным способом является специальный оператор `eval`, который выполняет строку в *текущей* области видимости. Обычно это плохая идея, потому что при этом нарушаются отдельные свойства, которыми обычно обладают области видимости, — например, предсказуемость того, какое имя относится к какой привязке.

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1
```

Менее неудачный способ интерпретации данных как кода — использовать конструктор `Function`. Он принимает два аргумента: строку, содержащую список имен аргументов через запятую, и строку, содержащую тело функции; он обертывает код в функцию-значение, чтобы получить собственную область видимости и не делать странных вещей с другими областями.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

Это именно то, что нам нужно для модульной системы. Мы можем обернуть код модуля в функцию и использовать область видимости этой функции в качестве области видимости модуля.

CommonJS

Наиболее широко применяемый подход для построения системы модулей JavaScript называется *модулями CommonJS*. Эта система используется в Node.js, а также в большинстве NPM-пакетов.

Основная концепция модулей CommonJS — функция, именуемая `require`. Если вызвать ее с именем модуля зависимости, то функция проверит, загружен ли этот модуль, и вернет его интерфейс.

Поскольку загрузчик обертывает код модуля в функцию, модули автоматически получают собственную локальную область видимости. Для того чтобы получить доступ к их зависимостям и поместить их интерфейс в объект, связанный с `exports`, остается лишь вызвать `require`.

В следующем примере показан модуль, содержащий функцию форматирования даты. Он использует два пакета из NPM — `ordinal` для преобразования чисел в строки, такие как `"1st"` и `"2nd"`, и `date-names`, чтобы получить английские названия дней недели и месяцев. Модуль экспортирует отдельную функцию `formatDate`, которая принимает объект `Date` и строку шаблона.

Строка шаблона может содержать коды, определяющие формат, такие как `YYYY` для полного года и `Do` для номера дня месяца. Например, если задать в качестве формата строку `"MMMM Do YYYY"`, то получим результат вида November 22nd 2017.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
```

```
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

Интерфейс `ordinal` — это отдельная функция, тогда как `date-names` экспортирует объект, содержащий несколько свойств: `days` и `months` являются массивами имен. Деструктуризация очень удобна при создании привязок для импортируемых интерфейсов.

Модуль добавляет к `exports` свою функцию интерфейса, чтобы модули, зависящие от него, получали к ней доступ. Мы могли бы использовать этот модуль следующим образом:

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

Мы можем определить `require` в самой минималистической форме, например так:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = {exports: {}};
    equire.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

В этом коде `readFile` представляет собой готовую функцию, которая читает файл и возвращает его содержимое в виде строки. В стандартном JavaScript такой функциональности нет, но различные среды JavaScript, такие как браузеры и Node.js, предоставляют собственные способы доступа к файлам. В этом примере просто предполагается, что `readFile` существует.

Во избежание загрузки одного и того же модуля несколько раз функция `require` хранит (кэширует) уже загруженные модули. При вызове она сначала

проверяет, был ли уже загружен запрошенный модуль, и если нет, то загружает его — читает код модуля, обертывает его в функцию и вызывает эту функцию.

Интерфейс пакета `ordinal`, который нам уже встречался, — это не объект, а функция. У модулей CommonJS есть одна странность: система модулей создает пустой интерфейсный объект (связанный с `exports`), но его можно заменить любым значением, перезаписав `module.exports`. Во многих модулях так и сделано, чтобы экспортировать одно значение вместо объекта интерфейса.

Определяя `require`, `export` и `module` в качестве параметров для генерируемой функции-оболочки (и передавая соответствующие значения при ее вызове), загрузчик обеспечивает доступность этих привязок в области видимости модуля.

Строка, переданная в `require`, преобразуется в фактическое имя файла или в веб-адрес разными способами, в зависимости от системы. Если строка начинается с `"./"` или `"../"`, то она обычно интерпретируется как путь относительно имени файла текущего модуля. Таким образом, `"./format-date"` — это файл с именем `format-date.js` в той же директории.

Если же имя не является относительным, Node.js будет искать установленный пакет с таким именем. В примере кода, рассмотренном в данной главе, мы будем интерпретировать такие имена как ссылки на NPM-пакеты. Подробнее о том, как устанавливать и использовать NPM-модули, я расскажу в главе 20.

Теперь, вместо того чтобы писать собственный анализатор INI-файлов, мы можем задействовать соответствующий NPM-пакет.

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

Модули ECMAScript

Модули CommonJS работают вполне приемлемо — в сочетании с NPM они позволили сообществу JavaScript начать широкомасштабное распространение кода.

Но они все еще остаются чем-то вроде решения «на коленке». Их нотация не очень удобна — например, то, что вы добавляете в `exports`, недоступно

в локальной области видимости. А поскольку `require` — это обычный вызов функции, принимающей любой тип аргумента, а не только строковый литерал, бывает сложно определить зависимости модуля, не выполняя его код.

Поэтому в стандарте JavaScript от 2015 года появилась собственная специальная система модулей. Обычно ее называют *ES-модулями*, где ES — сокращение от ECMAScript. Основные понятия зависимостей и интерфейсов остаются прежними, но детали различаются. С одной стороны, теперь нотация интегрирована в язык. Вместо вызова функции для доступа к зависимости можно использовать специальное ключевое слово `import`.

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

Аналогично для экспорта применяется ключевое слово `export`. Его можно ставить перед определением функции, класса или привязки (`let`, `const` или `var`).

Интерфейс ES-модуля — это не одно значение, а набор именованных привязок. Приведенный выше модуль связывает `formatDate` с функцией. При импорте из другого модуля импортируется не значение, а *привязка*, так что экспортирующий модуль может в любое время изменить значение привязки и модули, которые его импортируют, увидят это новое значение.

Если у модуля есть привязка с именем `default`, то она рассматривается как основное экспортируемое значение модуля. Так, при импортировании модуля `ordinal`, как показано в примере, без скобок вокруг имени привязки мы получим его привязку `default`. Наряду со своими значениями по умолчанию такие модули могут по-прежнему экспортировать другие привязки под другими именами.

Для того чтобы создать экспорт по умолчанию, нужно поставить перед выражением, объявлением функции или класса слова `export default`.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

Импортированные привязки можно переименовать, для этого нужно использовать ключевое слово `as`.

```
import {days as dayNames} from "date-names";

console.log(dayNames.length);
// → 7
```

Еще одно важное отличие заключается в том, что импорт ES-модуля происходит до выполнения кода модуля. Это означает, что объявления `import` не могут появляться внутри функций или блоков, а имена зависимостей должны быть представлены строками в кавычках, а не произвольными выражениями.

На момент написания настоящей книги сообщество JavaScript находилось в процессе принятия данного стиля модульной организации. Но это медленный процесс. Потребовалось несколько лет после принятия формата, прежде чем его стали поддерживать браузеры и Node.js. И несмотря на то, что в настоящее время они его в целом поддерживают, у этой поддержки все еще остаются проблемы и все еще продолжается обсуждение того, как такие модули должны распространяться через NPM.

Многие проекты пишутся с использованием ES-модулей, а затем автоматически преобразуются в какой-либо другой формат для публикации. Мы переживаем переходный период, когда две разные модульные системы существуют бок о бок, поэтому полезно уметь читать и писать код в любой из них.

Сборка и комплектация

На практике многие проекты JavaScript технически написаны не на JavaScript. Существуют распространенные расширения, такие как упоминавшийся в главе 8 диалект с проверкой типов. Люди также часто начинают использовать запланированные расширения языка задолго до того, как они встраиваются в платформы, которые в действительности поддерживают JavaScript.

Чтобы это стало возможным, код *компилируют*, переводя его с выбранного диалекта JavaScript на обычный добрый старый JavaScript — или даже в предыдущую версию JavaScript, — чтобы он выполнялся в старых браузерах.

Размещение на веб-странице модульной программы, которая состоит из 200 различных файлов, создает дополнительные проблемы. Если загрузка по сети одного файла занимает 50 миллисекунд, то загрузка всей программы займет десять секунд или в лучшем случае вдвое меньше, если удастся загрузить несколько файлов одновременно. Это много потерянного времени. Поскольку один большой файл обычно открывается быстрее, чем много

мелких, веб-программисты стали использовать инструменты, которые сворачивают программы (изначально кропотливо разбитые на модули) обратно в один большой файл, прежде чем публиковать его в Интернете. Такие инструменты называются *упаковщиками*.

И это еще не все. Помимо количества файлов, их *размер* также определяет, как быстро они могут быть переданы по сети. Поэтому сообщество JavaScript изобрело *минификаторы*. Они представляют собой инструменты, которые берут программу JavaScript и уменьшают ее, автоматически удаляя комментарии и пробелы, переименовывая привязки и заменяя фрагменты кода эквивалентным кодом, занимающим меньше места.

Поэтому код, который вы найдете в NPM-пакете или который запускается на веб-странице, нередко проходит *несколько* этапов преобразования — из современного JavaScript в устаревший, из ES-модуля в CommonJS, упаковку и минимизацию. В книге мы не будем углубляться в подробности работы этих инструментов, поскольку они довольно скучны и быстро изменяются. Просто помните, что код JavaScript, который вы запускаете, — часто не тот код, который был написан изначально.

Структура модулей

Структурирование программ — один из тонких аспектов программирования. Любая нетривиальная часть функциональности может быть представлена разными способами.

Понятие хорошо структурированной программы субъективно — приходится идти на компромиссы, и многое зависит от личных вкусов. Лучший способ понять важность хорошо структурированной программы — это читать большое количество программ или работать с ними, обращая внимание на то, что работает, а что — нет. Не думайте, что неприятный беспорядок — то, с чем остается только смириться. Вы можете улучшить структуру почти всего, если как следует подумать над ней.

Одним из аспектов модульной структуры является удобство применения. Если вы разрабатываете что-то, что предназначено для использования несколькими людьми — или даже только вами, но через три месяца, когда вы уже не будете помнить детали того, что сделали, — будет полезно, если ваш интерфейс окажется прост и предсказуем.

Это подразумевает следование существующим соглашениям. Хороший пример — пакет `ini`. Данный модуль имитирует стандартный объект `JSON`, предоставляя функции `parse` и `stringify` (для записи INI-файла) и, подобно `JSON`, выполняет преобразование строк в простые объекты и обратно. Получается краткий и знакомый интерфейс — поработав с ним один раз, вы, скорее всего, запомните, как им пользоваться.

Даже если не существует стандартной функции или широко используемого пакета, который можно было бы имитировать, все равно можно делать модули предсказуемыми, задействуя простые структуры данных и выполняя какую-то одну определенную задачу. Многие модули синтаксического анализа INI-файлов в NPM предоставляют функцию, которая, например, непосредственно считывает такой файл с жесткого диска и анализирует его. Это делает невозможным использование таких модулей в браузере, где нет прямого доступа к файловой системе, и усложняет модуль — вместо этого его можно было бы *скомпоновать* с модулем, реализующим функцию чтения файлов.

Это знакомит нас еще с одним полезным аспектом проектирования модулей — удобство компоновки с другим кодом. Модули, которые сосредоточены на одной задаче и вычисляют конкретные значения, применимы в более широком диапазоне программ, чем более крупные модули, выполняющие сложные действия с побочными эффектами. Функция чтения INI-файлов, которая требует чтения файла с диска, будет бесполезна в сценарии, где содержимое файла поступает из какого-то другого источника.

Вместе с этим объекты с сохранением состояния бывают полезны и даже необходимы, но если что-то можно сделать с помощью функции, используйте функцию. Некоторые из программ чтения INI-файлов в виде NPM-пакетов предоставляют стиль интерфейса, требующий, чтобы вы сначала создали объект, затем загрузили в него файл и только потом использовали специальные методы для получения результатов. Подобные вещи распространены в объектно-ориентированной традиции, но они просто ужасны. Вместо того чтобы всего один раз вызвать функцию и двигаться дальше, от вас требуют выполнить ритуал перемещения объекта через различные состояния. И поскольку данные теперь обернуты в специализированный тип объекта, весь код, который с ним взаимодействует, должен знать об этом типе, что создает ненужные взаимозависимости.

Часто невозможно избежать определения новых структур данных — языковой стандарт предоставляет лишь несколько базовых структур, и мно-

гие типы данных должны быть более сложными, чем массив или словарь. Но если массива достаточно, используйте его.

Примером немного более сложной структуры данных является граф из главы 7. В JavaScript не существует единого очевидного способа представления графов. В этой главе мы задействовали объект, свойства которого содержат массивы строк — другие узлы, доступные из данного узла.

В NPM есть несколько пакетов для поиска пути, но ни один из них не использует этот формат графов. Обычно они присваивают ребрам графа вес, который является стоимостью или расстоянием, соответствующим данному ребру. В нашем представлении подобное невозможно.

Например, есть пакет `dijkstrajs`. Хорошо известный подход к поиску пути, очень похожий на нашу функцию `findRoute`, называется *алгоритмом Дейкстры* в честь Эдсгера Дейкстры, который первым его записал. К именам пакетов часто добавляется суффикс `js`, чтобы указать, что они написаны на JavaScript. В этом пакете `dijkstrajs` применяется формат графа, похожий на наш, но вместо массивов в нем использованы объекты, значениями свойств которых являются числа — веса ребер.

Поэтому, если мы хотим использовать такой пакет, нам нужно убедиться, что наш граф сохранен в том формате, который ожидает данный пакет. Поскольку в нашей упрощенной модели все дороги имеют одинаковую стоимость (один ход), в данном случае все ребра имеют одинаковый вес.

```
const {find_path} = require("dijkstrajs");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}
console.log(find_path(graph, "Почта", "Хижина"));
// → ["Почта", "Дом Алисы", "Хижина"]
```

То, что различные пакеты используют разные структуры данных для описания сходных вещей, может быть препятствием для компоновки, так как их сложно объединить. Поэтому, если вы хотите построить программу с возможностью компоновки, выясните, какие структуры данных применяют другие разработчики, и по возможности следуйте их примеру.

Резюме

Модули обеспечивают структуру для больших программ, позволяя разделить код на части с понятными интерфейсами и зависимостями. Интерфейс — это часть модуля, видимая из других модулей, а зависимости — другие модули, которые использует данный модуль.

Поскольку изначально в JavaScript не было предусмотрено модульной системы, система CommonJS была построена как надстройка языка. Затем в какой-то момент встроенная поддержка модулей *появилась*, и теперь она — не без сложностей — сосуществует с системой CommonJS.

Пакет — это кусок кода, который может распространяться самостоятельно. NPM является хранилищем пакетов JavaScript. Оттуда можно загружать самые разные пакеты — как полезные, так и бесполезные.

Упражнения

Модульный робот

Вот привязки, созданные в проекте из главы 7:

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

Если бы вы написали данный проект в виде модульной программы, какие модули вы бы создали? Как бы эти модули зависели друг от друга и как бы выглядели их интерфейсы?

Какие фрагменты кода вы, скорее всего, обнаружили бы на NPM? Вы бы предпочли использовать NPM-пакеты или написать их самостоятельно?

Модуль Roads

Основываясь на примере из главы 7, напишите модуль CommonJS, который содержит массив дорог и экспортирует структуру данных графа, представленную как `roadGraph`. Модуль должен зависеть от модуля `./graph`, экспортирующего функцию `buildGraph`, которая используется для построения графа. Эта функция ожидает массива, состоящего из двухэлементных массивов (начальная и конечная точки для каждой дороги).

Циклические зависимости

Циклическая зависимость — это ситуация, когда модуль A зависит от модуля B, а B, в свою очередь, прямо или косвенно зависит от A. Во многих модульных системах подобное просто запрещено, поскольку, в каком бы порядке вы ни загружали эти модули, вы не сможете гарантировать загрузку всех зависимостей каждого модуля до его запуска.

Модули CommonJS допускают ограниченную форму циклических зависимостей. Если в модулях их объекты по умолчанию не заменяются `exports` и у них нет доступа к интерфейсам друг друга до завершения загрузки, циклические зависимости возможны.

Функция `require`, описанная ранее в этой главе, поддерживает данный тип циклических зависимостей. Видите, как она обрабатывает циклы? Что может пойти не так, если модуль в цикле *заменяет* свой объект по умолчанию `exports`?

11 Асинхронное программирование

> Кто способен спокойно ждать, пока грязь осядет?
>
> Кто способен оставаться на месте до момента, когда придет пора действовать?
>
> *Лао-цзы. Дао дэ цзин. Книга пути и благодати*

Центральная часть компьютера, та, что выполняет отдельные шаги, из которых состоят программы, называется *процессором*. Программы, встретившиеся нам до сих пор, построены так, чтобы процессор все время был занят, пока они не закончат свою работу. Скорость, с которой выполняется, к примеру, цикл, манипулирующий числами, в значительной степени зависит от скорости процессора.

Но многие программы взаимодействуют с устройствами, находящимися вне процессора. Например, они могут обмениваться данными по компьютерной сети или запрашивать их с жесткого диска, что намного медленнее, чем получать их из памяти.

Когда такое происходит, было бы неверно оставлять процессор без работы — может быть, в это время он мог бы сделать что-то полезное. Отчасти эту задачу решает операционная система, которая переключает процессор между несколькими работающими программами. Но это не помогает, если мы хотим, чтобы *одна* программа могла продолжать выполнение, пока ожидается ответ на запрос по сети.

Асинхронность

В модели *синхронного* программирования все операции выполняются по одной. Если вы вызвали функцию, которая выполняет какое-то длительное действие, она завершит работу и сможет вернуть результат только после того, как данное действие будет завершено. Все это время программа будет простаивать.

Асинхронная модель позволяет делать несколько вещей одновременно. Когда вы запускаете действие, программа продолжает работать. Когда действие заканчивается, программа получает уведомление об этом и получает доступ к результату (например, к данным, прочитанным с диска).

Синхронное и асинхронное программирование можно сравнить на простом примере — программы, которая получает по сети два ресурса, а затем объединяет результаты.

В синхронной среде, где функция запроса возвращает результат только после того, как она завершила свою работу, самый простой способ выполнить эту задачу — сделать запросы один за другим. У такого подхода есть недостаток: выполнение второго запроса начнется только после завершения первого. Общее время будет равно как минимум сумме времени выполнения обоих запросов.

Решение данной проблемы в синхронной системе заключается в запуске дополнительных управляющих потоков. *Поток* — это еще одна запущенная программа, так что операционная система может чередовать ее выполнение с другими программами. Это возможно, поскольку большинство современных компьютеров имеют несколько процессоров, так что несколько потоков могут выполняться одновременно на разных процессорах. Второй поток может запустить второй запрос, а затем оба потока будут ожидать возвращения результатов, после чего будут повторно синхронизированы для объединения последних.

На следующей схеме жирные линии представляют время, которое программа тратит на обычную работу, а тонкие — время, затраченное на ожидание ответа по сети. В синхронной модели время, затраченное на сеть, является *частью* временной шкалы для данного управляющего потока. В асинхронной модели начало сетевой операции концептуально вызывает *разделение* временной шкалы. Программа, которая инициировала действие, продолжает выполняться, а действие выполняется независимо от этого, уведомляя программу о своем завершении.

Еще один способ описать разницу заключается в том, что в синхронной модели ожидание завершения действий подразумевается *неявно*, а в асинхронной — *явно*, и мы его контролируем.

Асинхронность — обоюдоострый инструмент. Она упрощает описание программ, не вписывающихся в прямолинейную модель управления, но в то же время может сильно усложнить описание программ, которые хорошо выполняются последовательно. Далее в этой главе мы рассмотрим несколько способов решения этой проблемы.

Обе основные платформы программирования на JavaScript — браузеры и Node.js — выполняют операции, которые могут занять какое-то время, асинхронно, вместо того чтобы полагаться на потоки. Поскольку программирование с потоками печально известно своей сложностью (понять, что делает программа, гораздо сложнее, если она делает несколько вещей одновременно), это обычно считается хорошей идеей.

Технологии воронов

Большинство людей знают, что вороны — очень умные птицы. Они умеют использовать инструменты, планировать, запоминать разные вещи и даже общаться между собой.

Но мало кто знает, что вороны умеют и многое другое, и они это хорошо скрывают от нас. Один авторитетный (хотя и несколько эксцентричный) эксперт по врановым рассказал мне, что технологии воронов не сильно отстают от человеческих, и они догоняют нас.

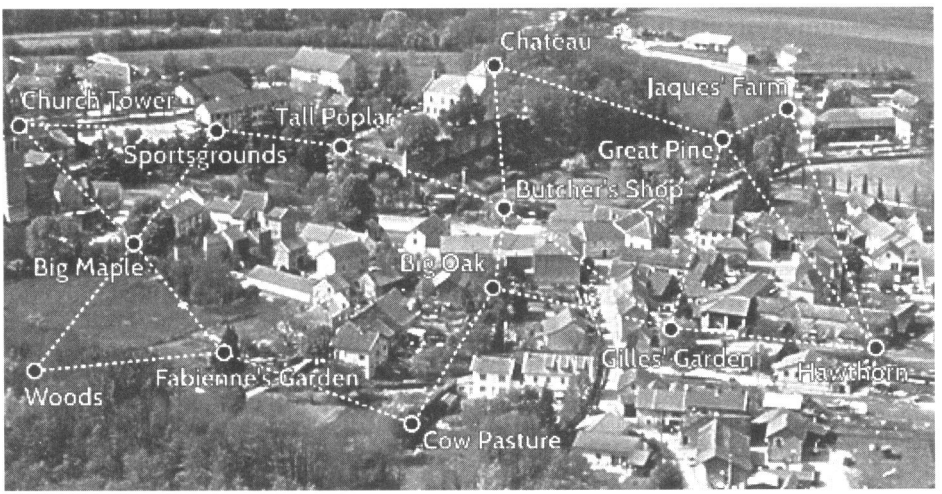
Например, многие сообщества воронов умеют конструировать вычислительные устройства. Это не электронные устройства, как у людей, — они работают за счет взаимодействия крошечных насекомых, близкородствен-

ных термитам, у которых развились симбиотические отношения с воронами. Птицы снабжают насекомых пищей, а те взамен строят сложные колонии и управляют ими. Эти колонии и выполняют вычисления посредством расположенных внутри них живых существ.

Такие колонии обычно расположены в больших долгоживущих гнездах. Птицы и насекомые вместе строят сеть луковицеобразных глиняных конструкций, спрятанных между ветками гнезда, — именно в них живут и работают насекомые.

Для коммуникации с другими устройствами эти машины используют световые сигналы. Вороны встраивают кусочки отражающего материала в специальные коммуникационные стебли, и насекомые направляют отраженный свет на другое гнездо, передавая данные в форме последовательности быстрых вспышек. Это означает, что обмен данными возможен только между теми гнездами, которые расположены в пределах прямой видимости.

Наш друг, эксперт по врановым, нанес на карту сеть гнезд в деревне Йерсюр-Амби, на берегу реки Рона. На ней показаны гнезда и взаимосвязи между ними.



В этом поразительном примере параллельной эволюции компьютеры воронов работают на JavaScript. В данной главе мы напишем для них несколько простейших сетевых функций.

Обратные вызовы

Один из подходов к асинхронному программированию состоит в том, чтобы заставить функции, которые выполняют медленные действия, принимать дополнительный аргумент — *функцию обратного вызова*. Действие запускается, и, когда оно будет завершено, вызывается функция обратного вызова, которой передается результат этого действия.

Например, функция `setTimeout`, доступная и в Node.js, и в браузерах, ожидает в течение заданного количества миллисекунд (в секунде 1000 миллисекунд) и затем вызывает функцию.

```
setTimeout(() => console.log("Tick"), 500);
```

Как правило, ожидание не очень важный тип работы, но оно может быть полезно при выполнении таких действий, как обновление анимации или проверка того, занимает ли что-то больше времени, чем предусмотрено.

Одновременное выполнение нескольких асинхронных действий с помощью обратных вызовов означает, что нужно продолжать передавать новые функции для обработки результатов вычислений после выполнения действий.

У большинства компьютеров типа «воронье гнездо» есть долговременный накопитель данных, в котором фрагменты информации выгравированы на ветках, так что их впоследствии можно извлечь. Гравировка и поиск фрагмента данных занимает некоторое время, поэтому интерфейс долговременного хранилища является асинхронным и использует функции обратного вызова.

В хранилище содержатся фрагменты данных в формате JSON, каждому из которых присвоено имя. Ворон может сохранить информацию о местах, где спрятана еда, под именем `"food caches"` — там может содержаться массив имен, указывающих на другие фрагменты данных, описывающие места фактических тайников с едой. Чтобы найти такой тайник в луковицах гнезда *Большого Дуба*, ворон может выполнить следующий код:

```
import {bigOak} from "./crow-tech";

bigOak.readStorage("food caches", caches => {
  let firstCache = caches[0];
  bigOak.readStorage(firstCache, info => {
    console.log(info);
```

```
  });
});
```

Все имена привязок и строк переведены с вороньего языка на английский.

Такой стиль программирования вполне работоспособен, но уровень отступов увеличивается с каждым асинхронным действием, потому что вы попадаете в другую функцию. Выполнение чего-то более сложного, например одновременное выполнение нескольких действий, может выглядеть немного некрасиво.

Компьютеры «воронье гнездо» построены с расчетом на коммуникацию с использованием пар «запрос — ответ». Это означает, что одно гнездо отправляет сообщение другому, которое затем немедленно отправляет сообщение в ответ, с подтверждением получения и, возможно, ответом на вопрос, заданный в первом сообщении.

Каждое сообщение имеет *тип*, определяющий, как оно должно обрабатываться. Наш код позволяет выявлять обработчики для разных типов запросов; когда такой запрос поступает, вызывается соответствующий обработчик для формирования ответа.

Интерфейс, экспортируемый из модуля `"./crow-tech"`, предоставляет коммуникационные функции, основанные на обратном вызове. У гнезд есть метод `send`, отправляющий запрос. Этот метод принимает в качестве первых трех аргументов имя гнезда-адресата, тип запроса и содержимое запроса, а в качестве четвертого — функцию, которая должна быть вызвана, когда будет получен ответ.

```
bigOak.send("Коровье пастбище", "note", "Давайте громко каркать
            в 7 вечера",
            () => console.log("Записка получена."));
```

Но чтобы гнезда могли принимать такие запросы, сначала нужно определить тип запроса с именем `"note"`. Код, который обрабатывает запросы, должен выполняться не только в этом гнезде-компьютере, но и во всех остальных гнездах, которые могут получать сообщения данного типа. Хорошо, предположим, что ворон пролетел над этими гнездами и в каждое установил код нашего обработчика.

```
import {defineRequestType} from "./crow-tech";

defineRequestType("note", (nest, content, source, done) => {
```

```
  console.log(`${nest.name} получил записку: ${content}`);
  done();
});
```

Функция `defineRequestType` определяет новый тип запроса. В этом примере добавлена поддержка запросов `"note"`, которая заключается в том, что просто отправляет записку в данное гнездо. В нашей реализации вызывается `console.log`, чтобы убедиться, что запрос получен. У каждого гнезда есть свойство `name`, которое содержит его имя.

Четвертый аргумент, переданный обработчику `done`, — функция обратного вызова, которую он должен вызвать после завершения запроса. Если бы мы использовали возвращаемое значение обработчика в качестве значения отклика, это бы означало, что обработчик запроса не может сам выполнять асинхронные действия. Функция, выполняющая асинхронную работу, обычно возвращается до того, как работа будет завершена, а после завершения работы совершает обратный вызов. Поэтому нам нужен какой-то асинхронный механизм — в данном случае другая функция обратного вызова, — чтобы сигнализировать о доступности ответа.

В некотором смысле асинхронность *заразна*. Любая функция, вызывающая функцию, работающую асинхронно, сама по себе также должна быть асинхронной и использовать обратный вызов или аналогичный механизм для доставки результата. Запуск функции обратного вызова несколько более сложен и подвержен ошибкам, чем простой возврат значения, поэтому необходимость структурировать большие части программы таким образом — неблагодарное дело.

Промисы

Работать с абстрактными понятиями зачастую проще, если их можно представить в виде значений. В случае асинхронных действий мы могли бы вместо функции, вызываемой когда-нибудь в будущем, вернуть объект, который представляет собой будущее событие.

Именно для этого существует стандартный класс `Promise`. *Промис*, или *обещание* (promise), — асинхронное действие, которое может завершиться в какой-то момент и произвести значение. Оно может уведомить любой заинтересованный объект о том, что такое значение доступно.

Самый простой способ создать промис — вызвать `Promise.resolve`. Эта функция гарантирует, что переданное ей значение будет обернуто в промис. Если это значение и так уже является промисом, то оно будет просто возвращено, в противном случае вы получите новый промис, который сразу же выдает заданное значение в качестве результата.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Получено ${value}`));
// → Получено 15
```

Чтобы получить результат промиса, можно использовать его метод `then`. Этот метод регистрирует функцию обратного вызова, которая будет вызвана, когда промис разрешится и создаст значение. У одного промиса может быть несколько обратных вызовов, и они будут вызываться, даже если добавить их после того, как промис будет *разрешен* (завершен).

Но это еще не все, что делает метод `then`. Он возвращает другой промис, разрешаемый значением, которое возвращает функция-обработчик, или, если та возвращает промис, ожидает этого промиса и затем разрешает его результат.

Бывает полезно рассматривать промисы как устройства, позволяющие перенести значения в асинхронную реальность. Обычное значение просто существует. Обещанное значение — это значение, которое *может* уже существовать или может появиться в какой-то момент в будущем. Вычисления, определенные в терминах промисов, оперируют такими упакованными значениями и выполняются асинхронно, по мере того как значения становятся доступными.

Чтобы создать промис, можно использовать `Promise` в качестве конструктора. У `Promise` немного странный интерфейс — конструктор ожидает в качестве аргумента функцию, которую он немедленно вызывает, передавая ей функцию, которую можно использовать для разрешения промиса. Конструктор работает таким образом, а не, например, с методом `resolve`, поэтому разрешить промис может только код, создавший этот промис.

Ниже показано, как можно создать интерфейс на основе промисов для функции `readStorage`:

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
```

```
  });
}

storage(bigOak, "enemies")
  .then(value => console.log("Получено", value));
```

Данная асинхронная функция возвращает осмысленное значение. В этом и заключается главное преимущество промисов — они упрощают использование асинхронных функций. Вместо того чтобы передавать обратные вызовы, функции, основанные на промисах, на вид похожи на обычные функции: они принимают входные данные в качестве аргументов и возвращают свои выходные данные. Разница лишь в том, что последние могут быть еще недоступны.

Сбои

Когда обычные вычисления JavaScript завершаются с ошибкой, они могут вызвать исключение. Асинхронным вычислениям тоже часто бывает нужно нечто подобное. Сетевой запрос может завершиться неудачей, или код, являющийся частью асинхронного вычисления, может вызвать исключение.

Одна из наиболее насущных проблем асинхронного программирования, связанных со стилем обратных вызовов, заключается в том, что чрезвычайно сложно организовать правильную доставку сообщений об ошибках в обратные вызовы.

Существует широко используемое соглашение: первый аргумент функции обратного вызова применяется для указания на то, что действие завершилось неудачей, а второй содержит значение, созданное действием, если оно завершилось успешно. Такие функции обратного вызова должны всегда проверять, было ли получено исключение, и убеждаться, что любые проблемы, в том числе исключения, полученные от вызываемых ими функций, будут перехвачены и переданы нужной функции.

С промисами это делается проще. Промисы могут быть либо разрешены (действие успешно завершено), либо отклонены (неудача). Обработчики разрешений (зарегистрированные с помощью `then`) вызываются только тогда, когда действие выполнено успешно, а отклонения автоматически распространяются на новый промис, возвращаемый методом `then`. И когда обработчик генерирует исключение, это автоматически вызывает отклоне-

ние промиса, созданного его вызовом then. Таким образом, если какой-либо элемент цепочки асинхронных действий даст сбой, результат всей цепочки будет помечен как отклоненный и никакие обработчики успешного завершения не будут вызваны после точки, в которой произошел сбой.

Подобно тому как разрешение промиса создает значение, отклонение промиса также создает значение, обычно называемое *причиной* отклонения. Если исключение, возникшее в функции-обработчике, привело к отклонению промиса, в качестве причины используется значение исключения. Аналогично, если обработчик возвращает отклоненный промис, это отклонение переходит в следующий промис. Существует функция `Promise.reject`, которая создает новый, заранее отклоненный промис.

Чтобы явно обрабатывать такие отклонения, у промисов есть метод `catch`, который регистрирует обработчик, вызываемый при отклонении промиса, подобно тому как обработчики `then` обрабатывают нормальное разрешение. Это также очень похоже на то, как `then` возвращает новый промис, который разрешается в значение исходного промиса, если все нормально, и в результат обработчика `catch` в случае сбоя. Если обработчик `catch` выдает ошибку, то новый промис также отклоняется.

Для краткости записи `then` также принимает обработчик отклонения в качестве второго аргумента, поэтому можно установить оба типа обработчиков в одном вызове метода.

Функция, передаваемая конструктору `Promise`, получает, кроме функции разрешения, второй аргумент, который может использоваться для отклонения нового промиса.

Цепочки значений промисов, создаваемые вызовами `then` и `catch`, можно рассматривать как конвейер, по которому движутся асинхронные значения или результаты сбоев. Поскольку такие цепочки создаются путем регистрации обработчиков, у каждой ссылки есть связанный с ней обработчик успешного завершения или обработчик отклонения (или оба). Обработчики, которые не соответствуют типу результата (успешное завершение или сбой), игнорируются. Но те, что соответствуют, вызываются, и их результат определяет, какое значение будет передано дальше: результат успешного завершения, когда возвращается значение, не являющееся промисом, или результат отклонения, когда вызывается исключение, или же результат промиса, когда возвращается либо то, либо другое.

```
new Promise((_, reject) => reject(new Error("Сбой")))
  .then(value => console.log("Обработчик 1"))
  .catch(reason => {
    console.log("Обнаружен сбой " + reason);
    return "ничего";
  })
  .then(value => console.log("Обработчик 2", value));
// → Обнаружен сбой Error: Сбой
// → Обработчик 2 ничего
```

Подобно тому как среда JavaScript обрабатывает неперехваченные исключения, эта же среда может фиксировать необработанные отклонения промисов и сообщать об этом как об ошибке.

Сетевые трудности

Иногда для вороньих зеркальных систем не хватает света, чтобы передать сигнал, или на пути сигнала оказывается какая-нибудь помеха. В результате сигнал может быть отправлен, но не получен.

Это приведет к тому, что функция обратного вызова, переданная в метод `send`, никогда не будет вызвана, отчего программа, скорее всего, будет остановлена, и система даже не заметит, что возникла проблема. Было бы неплохо, чтобы после определенного периода отсутствия ответа на запрос время ожидания ответа *истекло* и было бы выдано сообщение об ошибке.

Часто сбои в передаче данных вызваны случайными авариями, такими как фара автомобиля, свет которой смешался со световыми сигналами, и простое повторение запроса может привести к его успешному выполнению. Поэтому в рамках нашего эксперимента сделаем так, чтобы наша функция запроса автоматически повторяла отправку запроса несколько раз, прежде чем признать поражение.

И, поскольку мы уже оценили по достоинству промисы, обеспечим, чтобы наша функция запроса возвращала промис. С точки зрения того, что они могут сделать, обратные вызовы и промисы эквивалентны. Функции, основанные на принципе обратного вызова, могут быть обернуты, чтобы поддерживать интерфейс промисов, и наоборот.

Даже если запрос и его ответ успешно доставлены, ответ может указывать на сбой — например, если при запросе была попытка использовать неиз-

вестный тип запроса, или если обработчик выдал ошибку. Для поддержки этого методы `send` и `defineRequestType` следуют вышеупомянутому соглашению, когда первый аргумент, передаваемый в функцию обратного вызова, является причиной сбоя (если таковой случился), а второй — фактическим результатом.

Это можно преобразовать в систему промисов и разрешений посредством обертки.

```
class Timeout extends Error {}

function request(nest, target, type, content) {
  return new Promise((resolve, reject) => {
    let done = false;
    function attempt(n) {
      nest.send(target, type, content, (failed, value) => {
        done = true;
        if (failed) reject(failed);
        else resolve(value);
      });
      setTimeout(() => {
        if (done) return;
        else if (n < 3) attempt(n + 1);
        else reject(new Timeout("Timed out"));
        }, 250);
      }
      attempt(1);
    });
}
```

Поскольку промисы могут быть разрешены (или отклонены) только один раз, это будет работать. Результат промиса определяется при первом вызове `resolve` или `reject`, и все остальные вызовы, такие как истечение времени, наступившее после завершения запроса, или запрос, возвращенный после завершения другого запроса, игнорируются.

Чтобы построить асинхронный цикл для повторных попыток, нам нужно использовать рекурсивную функцию — обычный цикл не позволит останавливаться и ожидать завершения асинхронных действий. Функция `attempt` делает одну попытку отправить запрос. Она также устанавливает ограничение по времени: если по истечении 250 миллисекунд не будет получен ответ, то либо будет предпринята следующая попытка, либо, если это была

четвертая попытка, промис будет отклонен и причиной отклонения станет экземпляр `Timeout`.

Определенно, стратегия повторения попыток каждые четверть секунды и прекращения попыток, если через секунду ответа не последовало, выбрана несколько произвольно. Возможна даже ситуация, когда запрос поступил, но его обработка просто заняла немного больше времени и в результате запросы были доставлены несколько раз. Мы напишем наши обработчики с учетом этой проблемы — повторяющиеся сообщения не должны причинять вред.

Вообще говоря, сейчас мы не будем строить надежную сеть мирового уровня. Но это нормально — вороны пока не предъявляют особо больших требований к вычислениям.

Чтобы полностью изолироваться от обратных вызовов, мы пойдем немного дальше и определим также обертку для `defineRequestType`, которая позволяет функции-обработчику возвращать промис или простое значение и передавать это в обратный вызов.

```
function requestType(name, handler) {
  defineRequestType(name, (nest, content, source,
                           callback) => {
    try {
      Promise.resolve(handler(nest, content, source))
        .then(response => callback(null, response),
              failure => callback(failure));
    } catch (exception) {
      callback(exception);
    }
  });
}
```

`Promise.resolve` используется для преобразования значения, возвращаемого `handler`, в промис, если это значение само еще не является промисом.

Обратите внимание, что вызов `handler` должен был быть обернут в блок `try`, чтобы убедиться, что любое исключение, которое может возникнуть, будет напрямую передано функции обратного вызова. Это хорошо иллюстрирует сложность правильной обработки ошибок при использовании простых обратных вызовов — можно легко забыть правильно перенаправить подобные исключения, и если этого не сделать, то информация о сбоях не будет по-

падать в соответствующие обратные вызовы. При применении промисов это делается почти полностью автоматически и, следовательно, вызывает меньше ошибок.

Коллекции промисов

В каждом компьютере-гнезде есть свойство `neighbors`, где хранится массив информации о других гнездах, находящихся в пределах досягаемости передачи. Чтобы проверить, какие из них доступны в данный момент, можно написать функцию, которая пытается отправить запрос `"ping"` (запрос, просто требующий ответа) каждому из них и смотрит, какие из них ответят.

При работе с коллекциями промисов нам также пригодится функция `Promise.all`. Она возвращает промис, который ожидает разрешения всех промисов в массиве, а затем сама разрешается в массив значений, созданных этими промисами (и расположенных в том же порядке, что и исходный массив). Если какой-либо промис был отклонен, результат `Promise.all` также будет отклонен.

```
requestType("ping", () => "pong");

function availableNeighbors(nest) {
  let requests = nest.neighbors.map(neighbor => {
    return request(nest, neighbor, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(requests).then(result => {
    return nest.neighbors.filter((_, i) => result[i]);
  });
}
```

Если соседний компьютер недоступен, мы не хотим, чтобы весь объединенный промис завершился сбоем, потому что так мы все равно ничего не узнаем. Поэтому функция, которая сопоставляется с множеством соседних компьютеров, чтобы преобразовать это множество в промисы запросов, присоединяет обработчики, у которых успешные запросы генерируют `true`, а отклоненные — `false`.

В обработчике для комбинированного промиса применяется метод `filter` для удаления тех элементов из массива `neighbors`, которым соответствует значение `false`. Здесь использован тот факт, что в качестве второго

аргумента `filter` передает своей функции фильтрации индекс массива текущего элемента (подобно `map`, `some` и аналогичным методам массивов высшего порядка).

Лавина в сети

Тот факт, что гнезда могут общаться только со своими соседями, значительно снижает полезность сети.

Одним из решений для широковещательной рассылки информации по всей такой сети является создание типа запроса, который бы автоматически пересылался соседям. Эти соседи затем, в свою очередь, пересылали бы его своим соседям и так далее, пока вся сеть не получит сообщение.

```
import {everywhere} from "./crow-tech";

everywhere(nest => {
  nest.state.gossip = [];
});

function sendGossip(nest, message, exceptFor = null) {
  nest.state.gossip.push(message);
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "gossip", message);
  }
}

requestType("gossip", (nest, message, source) => {
  if (nest.state.gossip.includes(message)) return;
  console.log(`${nest.name} received gossip '${
               message}' from ${source}`);
  sendGossip(nest, message, source);
});
```

Чтобы избежать ситуации, когда одно и то же сообщение передается по сети вечно, каждое гнездо хранит `gossip` — строковый массив сплетен, где хранятся уже полученные сообщения. Чтобы определить такой массив, мы используем функцию `everywhere`, которая выполняет код для каждого гнезда, чтобы добавить свойство в объект гнезда `state`, где мы будем хранить локальное состояние гнезда.

Когда гнездо получает дубликат сплетни, что обычно случается, когда все слепо передают их друг другу, оно проигнорирует это сообщение. Но если сообщение новое, то гнездо взволнованно передаст его всем соседям, за исключением того, кто отправил ему данное сообщение.

Это приведет к тому, что свежие сплетни будут распространяться по сети, как круги на воде. Даже если какие-то соединения в настоящий момент не работают, если только существует альтернативный маршрут к данному гнезду, сплетня достигнет его по этому маршруту.

Такой стиль сетевого взаимодействия называется *лавинной маршрутизацией* — он заполняет сеть информацией до тех пор, пока эта информация не достигнет всех узлов.

Маршрутизация сообщений

Если один узел хочет пообщаться с другим узлом, то лавинная маршрутизация — не самый эффективный подход. Особенно если сеть обширная — тогда это может привести к большому количеству бесполезных операций по передаче данных.

Альтернативный подход состоит в том, чтобы создать способ передачи сообщения от узла к узлу, пока оно не достигнет пункта назначения. Сложность состоит в том, что для этого требуется знание структуры сети. Чтобы отправить запрос в отдаленное гнездо, необходимо знать, какое соседнее гнездо приближает запрос к пункту назначения. Отправка запроса в неправильном направлении не принесет пользы.

Поскольку каждое гнездо знает только своих непосредственных соседей, у него нет информации, необходимой для вычисления маршрута. Мы должны каким-то образом распространить информацию об этих соединениях по всем гнездам, предпочтительно таким образом, чтобы она могла изменяться во времени, по мере того как одни гнезда будут оставлены их обитателями и будут строиться другие гнезда.

Здесь мы можем снова использовать лавинную маршрутизацию, но вместо того, чтобы проверять, было ли уже доставлено данное сообщение, теперь мы будем проверять, соответствует ли новый набор соседей для этого гнезда текущему набору, который для него уже существует.

```
requestType("connections", (nest, {name, neighbors},
                            source) => {
  let connections = nest.state.connections;
  if (JSON.stringify(connections.get(name)) ==
      JSON.stringify(neighbors)) return;
  connections.set(name, neighbors);
  broadcastConnections(nest, name, source);
});

function broadcastConnections(nest, name, exceptFor = null) {
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "connections", {
      name,
      neighbors: nest.state.connections.get(name)
    });
  }
}
everywhere(nest => {
  nest.state.connections = new Map;
  nest.state.connections.set(nest.name, nest.neighbors);
  broadcastConnections(nest, nest.name);
});
```

Для сравнения используется метод `JSON.stringify`, поскольку `==` для объектов и массивов возвращает `true` только в том случае, если оба они имеют одно и то же значение, а это не то, что нам сейчас нужно. Сравнение строк JSON — грубый, но эффективный способ сопоставления их содержимого.

Узлы немедленно начинают широковещательную рассылку информации о своих соединениях, которые должны, если только какие-то гнезда не окажутся совершенно недоступными, позволить каждому гнезду быстро составить карту текущего сетевого графа.

Как мы видели в главе 7, графы хороши именно в построении маршрутов. Если только существует маршрут к пункту назначения сообщения, то мы знаем, в каком направлении его отправлять.

Ниже показана функция `findRoute`, которая очень похожа на `findRoute` из главы 7. Она ищет способ достичь определенного узла сети. Но вместо того, чтобы возвращать весь маршрут, эта функция возвращает только следующий шаг. Следующее гнездо само, используя свою текущую информацию о сети, решит, куда *дальше* отправлять сообщение.

```
function findRoute(from, to, connections) {
  let work = [{at: from, via: null}];
  for (let i = 0; i < work.length; i++) {
    let {at, via} = work[i];
    for (let next of connections.get(at) || []) {
      if (next == to) return via;
      if (!work.some(w => w.at == next)) {
        work.push({at: next, via: via || next});
      }
    }
  }
  return null;
}
```

Теперь мы можем построить функцию, которая будет в состоянии отправлять сообщения на дальние расстояния. Если сообщение адресовано непосредственному соседу, то оно будет доставлено как обычно. Если же нет, то оно будет упаковано в объект и отправлено соседу, находящемуся ближе других к пункту назначения, с использованием типа запроса `"route"`, который заставит этого соседа повторить то же самое поведение.

```
function routeRequest(nest, target, type, content) {
  if (nest.neighbors.includes(target)) {
    return request(nest, target, type, content);
  } else {
    let via = findRoute(nest.name, target,
                        nest.state.connections);
    if (!via) throw new Error(`No route to ${target}`);
    return request(nest, via, "route",
                   {target, type, content});
  }
}

requestType("route", (nest, {target, type, content}) => {
  return routeRequest(nest, target, type, content);
});
```

Мы создали несколько уровней функциональности поверх примитивной системы связи, чтобы ее было удобно использовать. Это хорошая (хотя и упрощенная) модель того, как работают реальные компьютерные сети.

Отличительным свойством компьютерных сетей является ненадежность — построенные на их основе абстракции полезны, но мы не можем

абстрагироваться от сбоя сети. Поэтому сетевое программирование в значительной степени направлено на предотвращение и устранение сбоев.

Асинхронные функции

Для хранения важной информации вороны дублируют ее в разных гнездах. Таким образом, если ястреб разрушит гнездо, информация не пропадет.

Чтобы извлечь определенную часть информации, которой нет в собственном хранилище, компьютер-гнездо может обращаться к случайно выбранным другим гнездам сети, пока не найдет то, где эта информация есть.

```
requestType("storage", (nest, name) => storage(nest, name));

function findInStorage(nest, name) {
  return storage(nest, name).then(found => {
    if (found != null) return found;
    else return findInRemoteStorage(nest, name);
  });
}

function network(nest) {
  return Array.from(nest.state.connections.keys());
}

function findInRemoteStorage(nest, name) {
  let sources = network(nest).filter(n => n != nest.name);
  function next() {
    if (sources.length == 0) {
      return Promise.reject(new Error("Not found"));
    } else {
      let source = sources[Math.floor(Math.random() *
                                      sources.length)];
      sources = sources.filter(n => n != source);
      return routeRequest(nest, source, "storage", name)
        .then(value => value != null ? value : next(),
              next);
    }
  }
  return next();
}
```

Поскольку connections — это объект Map, то Object.keys для него не работает. У него есть *метод* keys, но он возвращает итератор, а не массив. Итератор (итеративное значение) может быть преобразован в массив с помощью функции Array.from.

Несмотря на использование промисов, это довольно неуклюжий код. Несколько асинхронных действий связаны между собой неочевидными способами. Нам снова нужна рекурсивная функция (next), чтобы моделировать циклический перебор гнезд.

В сущности, код действует полностью линейно: он всегда ожидает завершения предыдущего действия, прежде чем приступить к следующему. Это было бы проще выразить в модели синхронного программирования.

Хорошая новость заключается в том, что JavaScript позволяет писать псевдосинхронный код, описывающий асинхронные вычисления. Функция async — это функция, которая неявно возвращает промис и которая в своем теле может ожидать (await) других промисов способом, *выглядящим* синхронно.

Функцию findInStorage можно переписать следующим образом:

```
async function findInStorage(nest, name) {
  let local = await storage(nest, name);
  if (local != null) return local;

  let sources = network(nest).filter(n => n != nest.name);
  while (sources.length > 0) {
    let source = sources[Math.floor(Math.random() *
                                    sources.length)];
        sources = sources.filter(n => n != source);
        try {
          let found = await routeRequest(nest, source, "storage",
                                         name);
          if (found != null) return found;
        } catch (_) {}
    }
    throw new Error("Not found");
}
```

Асинхронная функция отмечается ключевым словом async, которое ставится перед function. Метод также можно сделать асинхронным, поставив перед его именем async. При вызове такой функции или метода возвращается

промис. Как только тело функции что-то вернет, этот промис разрешается. Если возникнет исключение, то промис будет отклонен.

Внутри асинхронной функции слово `await` можно поместить перед выражением, чтобы дождаться разрешения промиса и только потом продолжить выполнение функции.

Такая функция, в отличие от обычных функций JavaScript, не выполняется за один раз от начала до конца. Вместо этого она может быть *заморожена* в любом месте, где стоит ключевое слово `await`, и ее выполнение будет возобновлено позже.

Для нетривиального асинхронного кода такая запись обычно более удобна, чем прямое использование промисов. Даже если вам нужно сделать что-то, что не соответствует синхронной модели, — например, выполнить одновременно несколько действий, — можно легко объединить `await` с прямым использованием промисов.

Генераторы

Способность приостанавливаться и затем возобновлять работу не является исключительным свойством асинхронных функций. В JavaScript также есть технология, называемая *функцией-генератором*. Она действует подобным образом, но без промисов.

Если определить функцию с помощью ключевого слова `function*` (звездочка после слова `function`), она становится генератором. При вызове генератора возвращается итератор, который мы уже встречали в главе 6.

```
function* powers(n) {
  for (let current = n;; current *= n) {
    yield current;
  }
}

for (let power of powers(3)) {
  if (power > 50) break;
  console.log(power);
}
// → 3
// → 9
// → 27
```

Первоначально при вызове `powers` функция сразу замораживается. При каждом последующем вызове `next` в итераторе функция выполняется до тех пор, пока не достигнет выражения `yield`, которое приостанавливает ее и делает полученное значение следующим значением, создающим итератор. Когда функция возвращает результат (в примере этого никогда не происходит), итератор завершается.

Писать итераторы часто бывает намного проще, если использовать функции-генераторы. Итератор для класса `Group` (из упражнения в главе 6) можно с помощью генератора представить следующим образом:

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

Больше не приходится создавать объект для хранения состояния итерации — генераторы автоматически сохраняют свое локальное состояние при каждом завершении.

Подобные выражения `yield` можно использовать только непосредственно в функции-генераторе, а не во внутренней функции, определенной внутри генератора. Состояние, которое генератор сохраняет при завершении, — это только его *локальная* среда и положение, в котором он находится.

Асинхронная функция — это особый тип генератора. При вызове она создает промис, который разрешается при возвращении (завершении) функции и отклоняется, если возникает исключение. Каждый раз, когда функция выдает (ожидает) промис, результат этого промиса (значение или возникшее исключение) является результатом выражения `await`.

Цикл событий

Асинхронные программы выполняются по частям. Каждая часть может запустить некоторые действия и запланировать выполнение определенного кода, когда действие завершится — успешно или неудачно. В промежутках между этими частями программа простаивает, ожидая следующего действия.

Таким образом, обратные вызовы не вызываются напрямую кодом, который их запланировал. Если вызвать `setTimeout` из функции, то к моменту

вызова функции обратного вызова исходная уже завершится. А когда завершится обратный вызов, управление не возвратится к функции, которая его запланировала.

Асинхронное поведение происходит в собственном пустом стеке вызовов функций. Это одна из причин того, что без промисов сложно управлять исключениями в асинхронном коде. Поскольку каждый обратный вызов начинается с почти пустого стека, обработчики `catch` не будут находиться в стеке в момент выдачи исключения.

```
try {
  setTimeout(() => {
    throw new Error("Лови");
  }, 20);
} catch (_) {
  // Это не выполнится
  console.log("Поймал!");
}
```

Независимо от того, насколько тесно связаны события, такие как истечение времени ожидания или входящие запросы, среда JavaScript будет выполнять только одну программу за раз. Это можно представить себе как выполнение большого цикла *снаружи* программы, называемого *циклом событий*. Когда ничего не должно быть сделано, данный цикл останавливается. Но по мере поступления событий они добавляются в очередь и их код выполняется один за другим. Поскольку никакие две операции не могут выполняться одновременно, медленно работающий код может задержать обработку других событий.

В следующем примере устанавливается тайм-аут, затем программа простаивает до момента окончания тайм-аута, в результате чего тайм-аут запаздывает.

```
let start = Date.now();
setTimeout(() => {
  console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

Разрешение или отклонение промиса всегда рассматривается как новое событие. Даже если промис уже разрешен, его ожидание приведет к обратному вызову не сразу, а после завершения текущего сценария.

```
Promise.resolve("Готово").then(console.log);
console.log("Сначала я!");
// → Сначала я!
// → Готово
```

В последующих главах мы рассмотрим многие другие типы событий, которые выполняются в цикле событий.

Дефекты асинхронного программирования

Пока программа выполняется синхронно, за один проход, не происходит никаких изменений состояния, кроме тех, которые вносит сама программа. С асинхронными программами ситуация другая: у них бывают *простои* в выполнении, во время которых может выполняться другой код.

Рассмотрим следующий пример. У наших воронов есть хобби: подсчет птенцов, которые появляются в деревне за год. Гнезда хранят эту информацию в своих луковицах-хранилищах. Следующий код предназначен для подсчета количества птенцов во всех гнездах за данный год:

```
function anyStorage(nest, source, name) {
  if (source == nest.name) return storage(nest, name);
  else return routeRequest(nest, source, "storage", name);
}

async function chicks(nest, year) {
  let list = "";
  await Promise.all(network(nest).map(async name => {
    list += `${name}: ${
      await anyStorage(nest, name, `chicks in ${year}`)
    }\n`;
  }));
  return list;
}
```

Код `async name =>` показывает, что стрелочные функции также можно сделать асинхронными, поставив перед ними слово `async`.

На первый взгляд код не выглядит подозрительно... асинхронная стрелочная функция применяется для множества гнезд, создается массив промисов, а затем с помощью `Promise.all` код дожидается завершения всех этих операций, прежде чем возвратить созданный список.

Но этот код серьезно сломан. Он всегда будет возвращать только одну строку вывода, в которой указано гнездо, отреагировавшее медленнее всех.

Догадаетесь почему?

Проблема заключается в операторе +=, который принимает *текущее* значение `list` на момент начала выполнения инструкции, а затем, когда завершается ожидание, присваивает привязке `list` это значение, к которому прибавлена последняя строка.

Но между моментом начала выполнения оператора и моментом его завершения существует асинхронный разрыв. Инструкция `map` выполняется прежде, чем что-либо добавляется в список, поэтому каждый из операторов += начинается с пустой строки и заканчивается, когда заканчивается извлечение из хранилища, так что `list` каждый раз представляет собой список из одной строки — результат прибавления строки к пустой строке.

Этого можно было бы легко избежать, если возвращать строки из сопоставляемых промисов и вызывать `join` для результата `Promise.all`, вместо того чтобы строить список путем изменения привязки. Как обычно, вычисление новых значений менее подвержено ошибкам, чем изменение существующих.

```
async function chicks(nest, year) {
  let lines = network(nest).map(async name => {
    return name + ": " +
      await anyStorage(nest, name, `chicks in ${year}`);
  });
  return (await Promise.all(lines)).join("\n");
}
```

Подобные ошибки совершить легко, особенно при использовании `await`, и вам следует знать, где могут возникнуть пробелы в вашем коде. Преимущество *явной* асинхронности JavaScript (будь то с помощью обратных вызовов, промисов или `await`) заключается в том, что обнаружить эти проблемные места относительно легко.

Резюме

Асинхронное программирование позволяет описать ожидание длительных действий, не останавливая программу на время выполнения их выполнения. В среде JavaScript этот стиль программирования обычно реализуется посредством обратных вызовов — функций, которые вызываются после

завершения определенного действия. Цикл событий планирует такие обратные вызовы, которые выполняются по мере необходимости, один за другим, чтобы они не перекрывали друг друга.

Асинхронное программирование упрощается благодаря промисам — объектам, представляющим действия, которые могут быть выполнены в будущем, и асинхронным функциям, позволяющим писать асинхронную программу так, как если бы она была синхронной.

Упражнения

Где скальпель?

У деревенских ворон есть старый скальпель, который они иногда используют для специальных целей — например, чтобы прорезать двери или что-то упаковать. Чтобы можно было быстро найти скальпель каждый раз, когда его перемещают в другое гнездо, в хранилище того гнезда, откуда его взяли, и гнезда, его забравшего, добавляется запись под названием `"scalpel"`, значением которой является информация о новом местоположении скальпеля.

Таким образом, чтобы найти скальпель, нужно проследить ряд записей в хранилищах, пока не будет найдено гнездо, хранящее указание на то гнездо, где в данный момент находится скальпель.

Напишите асинхронную функцию `locateScalpel`, которая это делает, начиная с того гнезда, в котором она выполняется. Для доступа к хранилищу в произвольном гнезде можете использовать определенную ранее функцию `anyStorage`. Скальпель передается между гнездами достаточно давно; возможно, в хранилище каждого гнезда есть запись `"scalpel"`.

Затем напишите такую же функцию, не используя `async` и `await`.

Правильно ли сбои запроса отображают отклонения возвращенного промиса в обеих версиях? Как именно это реализовано?

Построение Promise.all

Для заданного массива промисов `Promise.all` возвращает промис, который ожидает завершения всех промисов в массиве. Метод успешен, если получаем массив значений результатов. Если один из промисов в массиве

не выполнится, то промис, возвращаемый `all`, также не выполнится, и причиной отказа итогового промиса будет причина отказа невыполненного промиса из массива.

Реализуйте нечто подобное самостоятельно в виде обычной функции с именем `Promise_all`.

Помните, что после того, как промис выполнен успешно или же не выполнен, он не может быть выполнен или не выполнен снова, и дальнейшие вызовы функций, которые его разрешают, игнорируются. Это может упростить способ обработки промиса в вашей функции.

12 Проект: язык программирования

> Интерпретатор, который определяет смысл выражений в языке программирования, — это всего лишь еще одна программа.
>
> *Гарольд Абельсон и Джеральд Сассман. Структура и интерпретация компьютерных программ*

Создать собственный язык программирования на удивление легко (до тех пор пока вы не ставите слишком высокие цели) и очень поучительно.

Главное, что я хочу показать в этой главе, — то, что в создании собственного языка нет никакой магии. Я часто чувствовал, что некоторые человеческие изобретения так невероятно умны и сложны, что я никогда не смогу их понять. Но стоит немного почитать литературу и поэкспериментировать, как они часто оказываются довольно обыденными.

Мы построим язык программирования под названием Egg («Яйцо»). Это будет крошечный, простой язык, но достаточно мощный, чтобы выразить любые вычисления, которые только можно себе представить. Он будет допускать простые абстракции, основанные на функциях.

Синтаксический анализ

Наиболее заметной частью языка программирования является его *синтаксис*, или нотация. *Синтаксический анализатор* — это программа,

которая читает фрагмент текста и создает структуру данных, отражающую структуру программы, описанной в этом тексте. Если текст не представляет собой допустимую программу, синтаксический анализатор должен указать на ошибку.

У нашего языка будет простой и стандартный синтаксис. В Egg все является выражением. Выражение может представлять собой имя привязки, число, строку или *приложение*. Приложения используются для вызова функций, а также для таких конструкций, как `if` или `while`.

Для простоты синтаксического анализа строки в Egg не поддерживают экранирование обратной косой чертой. Строка — это просто последовательность любых символов, кроме двойных кавычек, заключенная в двойные кавычки. Число представляет собой последовательность цифр. Имена привязок могут состоять из любых символов, кроме пробела и тех символов, которые имеют специальное значение в синтаксисе.

Приложения пишутся так же, как в JavaScript, с круглыми скобками после выражения. В этих скобках может содержаться любое количество аргументов, разделенных запятыми.

```
do(define(x, 10),
   if(>(x, 5),
      print("large"),
      print("small")))
```

Стандартность языка Egg означает: то, что мы привыкли считать операторами в JavaScript (например, `>`), в этом языке является обычными привязками, применяемыми так же, как и другие функции. А поскольку в этом синтаксисе нет понятия блока, то нам нужна конструкция `do` для описания последовательного выполнения нескольких действий.

Структура данных, которую синтаксический анализатор будет использовать для описания программы, состоит из объектов выражений, каждый из которых имеет свойство `type`, определяющее тип выражения, и другие свойства, описывающие его содержимое.

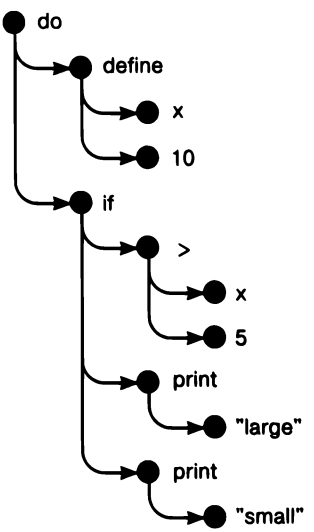
Выражения типа `"value"` представляют литеральные строки или числа. Их свойство `value` содержит строку или числовое значение, которое они представляют. Выражения типа `"word"` используются для идентификаторов (имен). Такие объекты имеют свойство `name`, которое содержит имя идентификатора в виде строки. Наконец, выражения `"apply"` представляют

приложения. У них есть свойство `operator`, которое определяет применяемое выражение, а также свойство `args`, содержащее массив выражений аргументов.

Часть `>(x, 5)` из приведенной выше программы будет представлена следующим образом:

```
{
   type: "apply",
   operator: {type: "word", name: ">"},
   args: [
     {type: "word", name: "x"},
     {type: "value", value: 5}
   ]
}
```

Такая структура данных называется *синтаксическим деревом*. Если представить объекты в виде точек, а связи между ними — в виде линий между этими точками, то получим древовидную структуру. То, что выражения содержат другие выражения, которые, в свою очередь, также могут содержать выражения, аналогично тому, как ветви дерева снова и снова разделяются на другие ветви.



Сравните это с синтаксическим анализатором, который мы написали для файла конфигурации в главе 9. Он имел простую структуру: анализатор

разбивал входные данные на строки и обрабатывал их по очереди. Каждая строка могла иметь лишь несколько простых форм.

Здесь нам нужно найти другой подход. Выражения не разделены на строки и имеют рекурсивную структуру. Выражения приложений *содержат в себе* другие выражения.

К счастью, это хорошо решается, если написать функцию синтаксического анализа, которая является рекурсивной и отражает рекурсивную природу языка.

Мы определим функцию `parseExpression`, которая принимает на входе строку и возвращает объект, содержащий структуру данных для выражения, содержащегося в начале строки, а также часть строки, оставшуюся после синтаксического анализа этого выражения. При синтаксическом анализе подвыражений (например, аргумента приложения) данную функцию можно вызвать снова и получить на выходе выражение аргумента, а также оставшийся текст. Этот текст может, в свою очередь, содержать другие аргументы или же быть закрывающей скобкой, заканчивающей список аргументов.

Вот первая часть синтаксического анализатора:

```
function parseExpression(program) {
  program = skipSpace(program);
  let 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: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

Поскольку Egg, как и JavaScript, допускает любое количество пробелов между элементами, приходится удалять эти пробелы в начале каждой строки программы. Именно для этого и предназначена функция `skipSpace`.

Пропустив все начальные пробелы, `parseExpression` использует три регулярных выражения, чтобы определить три атомарных элемента, которые поддерживает Egg: строки, числа и слова. Синтаксический анализатор строит структуру данных, тип которой зависит от того, какому из этих элементов соответствует строка. Если входные данные не соответствуют ни одной из трех форм, значит, это недопустимое выражение и анализатор выдает ошибку. В качестве конструктора исключений мы используем `SyntaxError` вместо `Error` — еще один стандартный объект для ошибок, который создается при попытке запустить недопустимую программу JavaScript.

Затем мы вырезаем из строки ту часть программы, которая соответствует шаблону, и передаем ее вместе с объектом выражения в функцию `parseApply`, проверяющую, является ли выражение приложением. Если это так, то она выполняет синтаксический анализ списка аргументов, заключенного в скобки.

```
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] != ")") {
    let 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 ',' или ')'");
    }
  }
  return parseApply(expr, program.slice(1));
}
```

Если следующий символ в программе не является открывающей скобкой, значит, это не приложение, и `parseApply` возвращает выражение, которое ему было передано.

В противном случае функция `parseApply` пропускает открывающую скобку и создает объект синтаксического дерева для этого выражения приложения. Затем она рекурсивно вызывает `parseExpression` для синтаксического анализа каждого аргумента, пока не дойдет до закрывающей скобки. Такая рекурсия является косвенной, поскольку `parseApply` и `parseExpression` вызывают друг друга.

Поскольку выражение приложения может быть применено само по себе (как `multiplier(2)(1)`), функция `parseApply` должна, после того как проанализирует приложение, снова себя вызвать, чтобы проверить, есть ли дальше еще одна пара круглых скобок.

Это все, что нам нужно для синтаксического анализа Egg. Мы оборачиваем его в удобную функцию `parse`, которая проверяет, достигнут ли конец входной строки после синтаксического анализа выражения (вся программа Egg — это одно большое выражение), и предоставляет нам структуру данных, описывающую программу.

```
function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(rest).length > 0) {
    throw new SyntaxError("Неожиданный текст после программы");
  }
  return expr;
}

console.log(parse("+(a, 10)"));
// → {type: "apply",
//    operator: {type: "word", name: "+"},
//    args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]}
```

Работает! Правда, программа не дает нам особо полезной информации в случае неудачи и не сохраняет строку и столбец, в которых начинается каждое выражение, что впоследствии пригодилось бы в сообщениях об ошибках, но для наших целей этого достаточно.

Интерпретатор

Что мы можем сделать с синтаксическим деревом программы? Конечно же, выполнить! Именно это и делает интерпретатор. Вы передаете ему синта-

ксическое дерево и объект области видимости, который связывает имена со значениями, а он выполнит выражение, описанное деревом, и возвратит полученное значение.

```
const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {
      throw new ReferenceError(
        `Неопределенная привязка: ${expr.name}`);
    }
  } else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expr.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Приложение не является функцией.");
      }
    }
  }
}
```

У интерпретатора есть код для каждого типа выражений. Выражение литерала возвращает его значение. (Например, выражение `100` выполнится и вернет просто число 100.) В случае привязки необходимо проверить, действительно ли она определена в данной области видимости, и, если это так, извлечь значение этой привязки.

С приложениями дело обстоит сложнее. Если приложение представляет собой специальную форму, такую как `if`, то мы ничего не вычисляем, а только передаем выражения аргумента вместе с областью видимости функции, которая обрабатывает данную форму. Если же это обычный вызов, то мы определяем оператор, проверяем, является ли он функцией, и вызываем его с предварительно вычисленными аргументами.

Для представления значений функций Egg мы используем обычные функциональные значения JavaScript. Мы вернемся к этому позже, когда определим специальную форму под названием `fun`.

Рекурсивная структура функции `evaluate` напоминает аналогичную структуру синтаксического анализатора, и оба они отражают структуру самого языка. Можно было бы интегрировать анализатор с интерпретатором и выполнять программу в процессе синтаксического анализа, но такое разделение делает программу более понятной.

Это действительно все, что нужно для интерпретации Egg. Все так просто. Но, не определив несколько специальных форм и не добавив в среду некоторые полезные значения, мы пока можем сделать на этом языке мало полезного.

Специальные формы

Объект `specialForms` используется в Egg для определения специального синтаксиса. Он связывает слова с функциями, которые вычисляют эти формы. В настоящее время данный объект пуст. Добавим в него `if`.

```
specialForms.if = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Неверное количество аргументов для if");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

Конструкция `if` в Egg ожидает ровно три аргумента. Функция вычисляет первый, и если результат не равен `false`, то вычисляет второй. В противном случае вычисляется третий аргумент. Такая форма `if` больше похожа на троичный оператор JavaScript `?:`, чем на `if`. Это выражение, а не инструкция, и оно возвращает значение, а именно результат вычисления второго или третьего аргумента.

Egg также отличается от JavaScript тем, как обрабатывается условное значение в `if`. Оно не приравнивает к `false` такие значения, как ноль или пустая строка, считая ложным только точное значение `false`.

Причина, по которой нам нужно представить `if` как специальную форму, а не как обычную функцию, состоит в том, что все аргументы функций вычисляются до вызова функции, тогда как в `if` должен вычисляться только второй или третий аргумент, в зависимости от значения первого аргумента.

Форма `while` аналогична `if`.

```
specialForms.while = (args, scope) => {
    if (args.length != 2) {
      throw new SyntaxError("Неверное число аргументов для while");
    }
    while (evaluate(args[0], scope) !== false) {
      evaluate(args[1], scope);
    }
    // Поскольку значения undefined в Egg не существует,
    // при отсутствии осмысленного результата возвращаем false.
    return false;
};
```

Еще одним базовым строительным блоком является `do`, который вычисляет все свои аргументы по очереди, сверху вниз. Его значение — это значение, созданное последним аргументом.

```
specialForms.do = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
  return value;
};
```

Чтобы иметь возможность создавать привязки и присваивать им новые значения, мы также создадим форму с именем `define`. В качестве первого аргумента эта форма ожидает слово, а в качестве второго — выражение, создающее значение, которое будет присвоено данному слову. Поскольку форма `define`, как и все остальное, является выражением, она должна возвращать значение. Мы сделаем так, чтобы она возвращала значение, которое было присвоено привязке (так же как оператор JavaScript `=`).

```
specialForms.define = (args, scope) => {
  if (args.length != 2 || args[0].type != "word") {
    throw new SyntaxError("Неверное использование определения");
  }
  let value = evaluate(args[1], scope);
```

```
  scope[args[0].name] = value;
  return value;
};
```

Среда выполнения

Область видимости, переданная в `evaluate`, является объектом со свойствами, чьи имена соответствуют именам привязок, а значения — значениям, с которыми связаны эти привязки. Определим объект для представления глобальной области видимости.

Чтобы использовать только что определенную конструкцию `if`, нам нужен доступ к логическим значениям. Поскольку существует только два логических значения, специальный синтаксис для них не нужен. Мы просто привяжем два имени к значениям `true` и `false` и будем их применять.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

Теперь мы можем вычислить простое выражение, которое отрицает логическое значение.

```
let prog = parse(`if(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

Чтобы описать простейшие арифметические операторы и операторы сравнения, мы добавим в область видимости еще несколько функциональных значений. Из соображений краткости кода мы не станем определять их по отдельности, а воспользуемся словом `Function`, чтобы синтезировать набор функций операторов в цикле.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

Нам также пригодится средство вывода значений, поэтому мы обернем `console.log` в функцию и назовем ее `print`.

```
topScope.print = value => {
  console.log(value);
  return value;
};
```

Теперь у нас есть достаточно элементарных инструментов для написания простых программ. Следующая функция предоставляет удобный способ синтаксического анализа программы и ее запуска в чистой области видимости:

```
function run(program) {
  return evaluate(parse(program), Object.create(topScope));
}
```

Для представления вложенных областей мы будем использовать цепочки прототипов объектов, чтобы программа могла добавлять привязки в свою локальную область видимости, не изменяя область видимости верхнего уровня.

```
run(`
do(define(total, 0),
   define(count, 1),
   while(<(count, 11),
         do(define(total, +(total, count)),
            define(count, +(count, 1)))),
   print(total))
`);
// → 55
```

Это программа, которую мы уже несколько раз видели ранее; она написана на Egg и вычисляет сумму чисел от 1 до 10. Такая программа выглядит явно менее привлекательно, чем ее эквивалент на JavaScript, но все равно неплохо для языка, в котором менее 150 строк кода.

Функции

Язык программирования без функций — это очень плохой язык программирования.

К счастью, нетрудно добавить конструкцию `fun`, которая рассматривает свой последний аргумент как тело функции и использует все предшествующие аргументы как имена параметров этой функции.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("У функции должно быть тело");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
    if (expr.type != "word") {
```

```
      throw new SyntaxError("Именами параметров должны быть слова");
    }
    return expr.name;
  });
    return function() {
      if (arguments.length != params.length) {
        throw new TypeError("Некорректное число аргументов");
      }
      let localScope = Object.create(scope);
      for (let i = 0; i < arguments.length; i++) {
        localScope[params[i]] = arguments[i];
      }
      return evaluate(body, localScope);
    };
};
```

У каждой функции в Egg своя локальная область видимости. Функция, построенная формой `fun`, создает эту локальную область и размещает в ней привязки аргументов. Затем она вычисляет тело функции в данной области видимости 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

То, что мы построили, называется интерпретатором. В процессе выполнения он воздействует непосредственно на представление программы, созданной синтаксическим анализатором.

Компиляция — это процесс добавления еще одного шага между синтаксическим анализом и выполнением программы. Компиляция превращает программу в нечто, что можно выполнить более эффективно, сделав заранее как можно больше работы. Например, в качественно разработанных языках программирования для каждой используемой привязки очевидно, что это за привязка, не нужно выполнять саму программу. К такому можно прибегать, чтобы не искать привязку по имени каждый раз, когда к ней обращаются, а просто сразу выбрать ее из некоторой предопределенной области памяти.

Традиционно компиляция подразумевает преобразование программы в машинный код — формат, который может быть выполнен компьютерным процессором непосредственно. Но, вообще говоря, любой процесс, преобразующий программу в другое представление, может рассматриваться как компиляция.

Можно было бы написать альтернативную стратегию выполнения для языка Egg, при которой программа на Egg сначала преобразуется в программу на JavaScript, используя `Function`, чтобы вызвать для нее компилятор JavaScript, а затем выполняет результат. Если все сделано правильно, то Egg получится очень быстрым, но при этом достаточно простым в реализации языком.

Если вы заинтересованы в данной теме и хотите уделить ей некоторое время, рекомендую вам в качестве упражнения попробовать реализовать такой компилятор.

Немного мошенничества

Когда мы определяли `if` и `while`, вы, вероятно, заметили, что это были более или менее тривиальные обертки вокруг собственных `if` и `while` JavaScript. Аналогично значения в Egg — это просто добрые старые значения JavaScript.

Если сравнить реализацию Egg, основанную на JavaScript, с объемом работы и сложностью, необходимой для построения языка программирования непосредственно на основе сырой функциональности, предоставляемой машиной, то разница будет огромной. Тем не менее данный пример дал вам идеальное представление о том, как работают языки программирования.

И когда дело доходит до выполнения чего-либо, лучше немного смошенничать, чем делать все самому. Хотя игрушечный язык, реализованный в этой главе, не делает ничего такого, чего нельзя было бы еще лучше сделать

в JavaScript, *существуют* ситуации, когда написание небольших языков помогает выполнить настоящую работу.

Такой язык не должен напоминать типичный язык программирования. Например, если бы JavaScript не поддерживал регулярные выражения, то можно было бы написать собственный синтаксический анализатор и интерпретатор регулярных выражений.

Или предположим, что вы строите гигантского робота-динозавра и вам нужно запрограммировать его поведение. JavaScript может оказаться не самым эффективным средством. Вместо этого можно выбрать язык, который выглядит следующим образом:

```
поведение гулять
  выполнить, когда
    пункт назначения вперед
  действия
    шаг левой ногой
    шаг правой ногой

поведение атака
  выполнять, когда
    Годзилла в поле зрения
  действия
    зажечь лазерные глаза
    запустить ракеты
```

Обычно это называют *предметно-ориентированным языком программирования* — языком, предназначенным для описания узкой предметной области знаний. Такой язык может быть более выразительным, чем язык общего назначения, поскольку он предназначен для описания именно того, что необходимо описать в его области, и ничего более.

Упражнения

Массивы

Дополните Egg поддержкой массивов, добавив в верхнюю область видимости следующие три функции: `array(...values)` для создания массива, содержащего значения аргументов, `length(array)` для получения длины массива и `element(array, n)`, чтобы получить *n*-й элемент массива.

Замыкание

То, как мы определили `fun`, позволяет функциям в Egg ссылаться на окружающую область видимости, а телу функции — использовать локальные значения, которые были видимыми во время определения функции, как это делают функции JavaScript.

Это показано в следующей программе: функция `f` возвращает функцию, добавляющую к аргументу `f` свой аргумент. Это означает, что для того, чтобы иметь возможность использовать привязку `a`, ей нужен доступ к локальной области видимости внутри `f`.

```
run(`
do(define(f, fun(a, fun(b, +(a, b)))),
   print(f(4)(5)))
`);
// → 9
```

Вернитесь к определению формы `fun` и объясните, благодаря какому механизму это работает.

Комментарии

Было бы хорошо, если бы в Egg можно было писать комментарии. Например, всякий раз, когда в коде встретится символ «решетка» (`#`), мы можем рассматривать оставшуюся часть строки как комментарий и игнорировать его, как `//` в JavaScript.

Для того чтобы это реализовать, не нужно вносить большие изменения в синтаксический анализатор. Мы можем просто изменить `skipSpace`, чтобы пропускать комментарии, как если бы они были пробелами, чтобы все точки, где вызывается `skipSpace`, теперь пропускали также и комментарии. Внесите это изменение.

Исправление области видимости

Сейчас единственным способом присвоить привязке значение является `define`. Эта конструкция действует и для определения новых привязок, и для назначения существующим привязкам нового значения.

Такая двусмысленность становится источником проблем. Если попытаться присвоить нелокальной привязке новое значение, мы вместо этого

определим локальную привязку с тем же именем. Некоторые языки так и работают, но я всегда считал, что это некрасивый способ обработки области видимости.

Добавьте специальную форму `set`, аналогичную `define`, которая назначает привязке новое значение, обновляя привязку во внешней области видимости, если она еще не существует во внутренней области. Если привязка вообще не определена, должно генерироваться исключение `ReferenceError` (еще один стандартный тип ошибки).

Технология представления областей видимости как простых объектов, которая до сих пор была для нас удобной, теперь будет немного мешать. Возможно, вы захотите использовать функцию `Object.getPrototypeOf`, возвращающую прототип объекта. Помните также, что области видимости не являются производными от `Object.prototype`, поэтому если вы хотите вызывать для них `hasOwnProperty`, то придется использовать такое неуклюжее выражение:

```
Object.prototype.hasOwnProperty.call(scope, name);
```

13 JavaScript и браузер

> Мы мечтали, что сеть станет единым информационным пространством, в котором мы будем общаться, обмениваясь информацией. Главное в сети — ее универсальность: гипертекстовая ссылка может указывать на что угодно — личное, локальное или глобальное, черновик или готовый материал.
>
> *Тим Бернерс-Ли. Всемирная паутина: очень короткая биография*

В следующих главах этой книги мы поговорим о браузерах. Без браузеров не было бы JavaScript. А даже если бы и был, то никто бы не обратил на него внимания.

С самого начала веб-технологии были децентрализованы не только технически, но и по тому, как они развивались. Различные поставщики браузеров добавляли новую функциональность, исходя из текущей ситуации, иногда плохо продуманными способами; впоследствии ее иногда перенимали другие производители — и в итоге она становилась стандартом.

Это стало и благословением, и проклятием. С одной стороны, это позволяет не создавать централизованную систему контроля, а лишь вводить улучшения со стороны различных производителей, работающих в режиме свободного сотрудничества (или иногда открытой враждебности). С другой стороны, тот случайный способ, которым была разработана сеть, означает, что получившуюся систему трудно назвать образцом внутренней согласованности. Отдельные ее части совершенно запутаны и плохо продуманы.

Интернет и другие сети

Компьютерные сети существуют с 1950-х годов. Если соединить кабелями два или более компьютера и позволить им передавать данные по этим кабелям туда и обратно, то можно будет делать множество самых удивительных вещей.

И если уже соединение двух машин, расположенных в одном здании, позволяет нам делать удивительные вещи, то каким же будет соединение машин, разбросанных по всей планете! Технология, позволившая начать реализацию этой идеи, была разработана в 1980-х годах, и получившуюся в результате сеть назвали *Интернетом*. Сеть оправдала возложенные на нее надежды.

Посредством этой сети компьютер может выстрелить пучком битов в другой компьютер. Для того чтобы из такой перестрелки битами возникла сколько-нибудь эффективная связь, компьютеры на обоих концах линии должны знать, что представляют собой данные биты. Значение любой их последовательности полностью зависит от того, что она пытается выразить, и от используемого механизма кодирования.

Стиль связи по сети описывает *сетевой протокол*. Существуют протоколы для передачи электронной почты, получения электронной почты, обмена файлами и даже для управления компьютерами, которые оказались зараженными вредоносным программным обеспечением.

Например, *протокол HTTP* (Hypertext Transfer Protocol, протокол передачи гипертекста) — протокол для получения именованных ресурсов (фрагментов информации, таких как веб-страницы или изображения). Он указывает, что сторона, выполняющая запрос, должна начинать передачу со строки, где указаны ресурс и версия протокола, который она намерена использовать:

```
GET /index.html HTTP/1.1
```

Существует еще множество правил, определяющих то, как запрашивающая сторона может включать в запрос больше информации, а также определяющих способ, которым другая сторона, возвращающая ресурс, упаковывает его содержимое. Подробнее HTTP будет рассмотрен в главе 18.

Большинство протоколов построено на основе других протоколов. HTTP рассматривает сеть как потоковое устройство, в которое можно помещать биты и получать их в заданном месте назначения в правильном порядке. Как было показано в главе 11, обеспечение этого уже само по себе является довольно сложной проблемой.

Такую проблему решает *протокол TCP* (Transmission Control Protocol, протокол управления передачей). На нем «говорят» все устройства, подключенные к Интернету, и большая часть обмена данными в Интернете построена именно на нем.

TCP-соединение работает следующим образом: один компьютер ожидает, или *прослушивает*, сеть, и тогда другие компьютеры могут начать с ним разговаривать. Чтобы иметь возможность одновременно прослушивать различные виды коммуникаций на одной машине, каждому слушателю присвоен номер (называемый *портом*). Большинство протоколов указывают, какой порт следует использовать для них по умолчанию. Например, если мы хотим отправить электронное письмо по протоколу SMTP, компьютер, с которого мы отправляем письмо, ожидает, что у адресата его будет прослушивать порт 25.

Один компьютер может установить соединение с другим компьютером, используя правильный номер порта. Если компьютер-адресат может быть доступен и будет прослушивать этот порт, то соединение будет успешно создано. Слушающий компьютер называется *сервером*, а подключающийся — *клиентом*.

Такое соединение действует как двусторонний канал, по которому могут проходить биты — машины на обоих концах могут помещать в канал данные. После того как биты будут успешно переданы, они могут быть снова прочитаны машиной на другой стороне линии. Это удобная модель. Можно сказать, что TCP обеспечивает абстракцию сети.

Web

Всемирная паутина, World Wide Web (не путать с Интернетом в целом), — это набор протоколов и форматов, которые позволяют посещать веб-страницы посредством браузера. Слово Web в названии означает, что такие страницы могут легко ссылаться друг на друга, образуя огромную сеть, по которой, как пауки по паутине, могут перемещаться пользователи.

Чтобы стать частью Web, нужно лишь подключить компьютер к Интернету и настроить его на прослушивание порта 80 по протоколу HTTP, и тогда другие компьютеры смогут запрашивать у него документы. Каждому документу в Web присвоено имя — URL (Uniform Resource Locator, универсальный локатор ресурса), который выглядит примерно так:

```
 http://eloquentjavascript.net/13_browser.html
|      |                        |               |
протокол       сервер                  путь
```

Первая часть говорит, что данный URL использует протокол HTTP (в отличие, например, от зашифрованного HTTP, в случае которого там бы стояло `https://`). Следующая часть определяет, с какого сервера запрашивается документ. Последняя строка — это путь, который идентифицирует конкретный документ (или *ресурс*), нас интересующий.

Каждая машина, подключенная к Интернету, получает *IP-адрес* — число, которое может использоваться для отправки сообщений на эту машину и выглядит примерно так: `149.210.142.219` или `2001:4860:4860::8888`. Но списки, состоящие из более или менее случайных чисел, трудно запомнить и неудобно набирать, поэтому вместо них можно зарегистрировать *доменное имя* для определенного адреса или набора адресов. Я зарегистрировал имя eloquentjavascript.net, указывающее на IP-адрес компьютера, которым я управляю, и таким образом могу использовать это доменное имя для обслуживания веб-страниц.

Если ввести этот URL в адресную строку браузера, браузер попытается получить и отобразить документ, расположенный по этому URL-адресу. Сначала браузер выяснит, на какой адрес ссылается eloquentjavascript.net. Затем, используя протокол HTTP, установит соединение с сервером по этому адресу и запросит ресурс `/13_browser.html`. Если все получится, то сервер отправит в ответ документ, который отобразится браузером на экране.

HTML

Аббревиатура HTML, которая расшифровывается как *Hypertext Markup Language*, язык разметки гипертекста, — это формат документа, используемый для веб-страниц. HTML-документ содержит текст, а также *теги*, которые определяют структуру текста, описывая такие элементы, как ссылки, абзацы и заголовки.

Короткий HTML-документ может выглядеть так:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
```

```
    <title>Моя домашняя страница</title>
  </head>
  <body>
    <h1>Моя домашняя страница</h1>
    <p>Привет, меня зовут Марейн, и это моя домашняя страница.</p>
    <p>А еще я написал книгу! Читайте ее
      <a href="http://eloquentjavascript.net">здесь</a>.</p>
  </body>
</html>
```

Вот как будет выглядеть такой документ в браузере:

Моя домашняя страница

Привет, меня зовут Марейн, и это моя домашняя страница.

А еще я написал книгу! Читайте ее здесь.

Теги, заключенные в угловые скобки (< и >, символы «меньше» и «больше»), содержат информацию о структуре документа. Остальной текст — это просто текст.

Документ начинается с тега `<!doctype html>`, который сообщает браузеру, что данную страницу следует интерпретировать как *современный* HTML, в отличие от различных диалектов, которые использовались в прошлом.

У HTML-документов есть заголовок (`<head>`) и тело (`<body>`). В заголовке заключена *информация о документе*, а тело содержит сам документ. В данном случае в заголовке сообщается, что название документа — «Моя домашняя страница» и что в документе используется кодировка UTF-8, которая является способом кодирования текста Unicode в виде двоичных данных. Тело документа содержит заголовок — `<h1>`, что означает «заголовок (heading) уровня 1»; существуют также подзаголовки от `<h2>` до `<h6>` — и два абзаца (`<p>`, от paragraph).

Теги бывают разных видов. Такой элемент, как тело, абзац или ссылка, начинается с *открывающего тега*, например `<p>`, и заканчивается *закрывающим тегом*, например `</p>`. Некоторые открывающие теги, такие как тег ссылки (`<a>`), содержат дополнительную информацию в виде пар `имя="значение"`. Это так называемые *атрибуты*. В данном случае адрес ссылки указывается как `href="http://eloquentjavascript.net"`, где `href` означает *hypertext reference* (гипертекстовая ссылка).

Некоторые виды тегов ничего в себе не заключают, и поэтому их не нужно закрывать. Примером такого тега является тег метаданных `<meta charset="utf-8">`.

Чтобы иметь возможность использовать в тексте документа угловые скобки, несмотря на то что в HTML они имеют особое значение, необходимо ввести еще одну форму специальных обозначений. Простая открывающая угловая скобка обозначается как `<` (от less than — «меньше чем»), а закрывающая — как `>` (greater than — «больше чем»). В HTML символ амперсанда (`&`), за которым следует имя или код символа и точка с запятой (`;`), называется *сущностью* и заменяется символом, который он кодирует.

Это похоже на использование обратной косой черты в строках JavaScript. Поскольку данный механизм придает символам амперсанда также особое значение, их необходимо экранировать как `&`. Внутри тегов значения атрибутов заключаются в двойные кавычки, поэтому для вставки в текст настоящих кавычек можно использовать `&quot;`.

Синтаксический анализ HTML удивительно устойчив к ошибкам. При отсутствии тегов, которые должны быть на этом месте, браузер их восстанавливает. Способ, которым это делается, был стандартизирован, и будьте уверены, что все современные браузеры делают это одинаково.

Следующий документ будет обрабатываться так же, как предыдущий:

```
<!doctype html>

<meta charset=utf-8>
<title>Моя домашняя страница</title>

<h1>Моя домашняя страница</h1>
<p>Привет, меня зовут Марейн, и это моя домашняя страница.
<p>А еще я написал книгу! Читайте ее
  <a href=http://eloquentjavascript.net>здесь</a>.
```

Теги `<html>`, `<head>` и `<body>` пропали. Браузер знает, что `<meta>` и `<title>` относятся к заголовку, а с `<h1>` начинается тело страницы. Более того, я не стал закрывать абзацы явно, так как открытие нового абзаца или завершение документа неявно закрывает предыдущий абзац. Кавычки вокруг значений атрибутов также исчезли.

В примерах, представленных в этой книге, обычно не будут использоваться теги `<html>`, `<head>` и `<body>`, чтобы они были короче и в них было бы меньше лишнего. Но я *буду* закрывать теги и заключать атрибуты в кавычки.

Я также обычно опускаю объявления `doctype` и `charset`. Это не значит, что их следует удалять из HTML-документов. Если забыть поставить такие теги, браузеры часто делают забавные вещи. Следует помнить, что тип документа и метаданные о наборе символов неявно присутствуют в примерах, даже если они не появляются в тексте.

HTML и JavaScript

В контексте этой книги самый важный для нас тег HTML — это `<script>`. Данный тег позволяет включать в документ фрагменты JavaScript.

```
<h1>Проверка предупреждения</h1>
<script>alert("Привет!");</script>
```

Этот сценарий будет запущен сразу после обнаружения тега `<script>`, в процессе чтения браузером HTML-документа. При открытии страницы появится диалоговое окно — функция `alert` напоминает `prompt` в том смысле, что текст всплывает в небольшом окне, но `alert` показывает только сообщение, без запроса ввода.

Вставлять большие программы непосредственно в HTML-документы обычно нецелесообразно. Тегу `<script>` можно присвоить атрибут `src` для подключения файла сценария (текстового файла, содержащего программу на JavaScript) из URL-адреса.

```
<h1>Проверка предупреждения</h1>
<script src="code/hello.js"></script>
```

Подключаемый здесь файл `code/hello.js` содержит ту же программу — `alert("Привет!")`. Когда HTML-страница ссылается на другие URL-адреса как часть себя — например, на файл изображения или сценарий, — браузер сразу извлекает их содержимое и включает его в страницу.

Тег сценария всегда должен быть закрыт с помощью `</script>`, даже если он ссылается на файл сценария и не содержит никакого кода. Если забыть об этом, то остальная часть страницы будет интерпретирована как часть сценария.

В браузер можно загружать ES-модули (см. главу 10), указав в теге `script` атрибут `type="module"`. Такие модули могут зависеть от других, используя в качестве имен модулей в объявлениях `import` URL-адреса этих модулей относительно себя.

Некоторые атрибуты также могут содержать JavaScript-программы. Показанный ниже тег `<button>` (отображаемый как кнопка) имеет атрибут `onclick`. Значение атрибута будет выполняться при каждом нажатии кнопки.

```
<button onclick="alert('Бабах!');">НЕ НАЖИМАЙТЕ!</button>
```

Обратите внимание, что мне пришлось применить одинарные кавычки для строки в атрибуте `onclick`, потому что двойные уже используются — в них заключен весь атрибут. Я также мог бы использовать `&quot;`.

В «песочнице»

Запускать программы, загруженные из Интернета, потенциально опасно. Вы мало что знаете о людях, стоящих за большинством посещаемых вами сайтов, и у этих людей не всегда добрые намерения. Именно запуск программ злоумышленниками приводит к заражению компьютерными вирусами, краже персональных данных и взлому учетных записей.

Однако привлекательность Web такова, что вы часто можете просматривать страницы, не особенно им доверяя. Вот почему браузеры строго ограничивают возможности программы JavaScript: она не может просматривать файлы на вашем компьютере или изменять что-либо не относящееся к вебстранице, в которую она встроена.

Такая изоляция среды программирования называется *«песочницей»*. Идея состоит в том, что играть в такой программной «песочнице» безвредно. Но вам следует представлять себе эту «песочницу» как клетку из толстых стальных прутьев, чтобы программы, которые в ней играют, не могли оттуда выйти.

Сложная часть «песочницы» заключается в том, чтобы предоставить программам достаточно места, обеспечив их полезность, и в то же время не позволить им делать что-либо опасное. Множество полезных функций, таких как коммуникация с другими серверами или чтение буфера обмена, также могут использоваться для решения проблемных задач, нарушающих конфиденциальность.

Время от времени кто-нибудь придумывает новый способ обойти ограничения браузера и сделать что-то вредное — от незначительной утечки личной информации до захвата всей машины, на которой работает браузер.

Разработчики браузеров в ответ закрывают эту дыру, и все снова становится хорошо — до тех пор пока не будет обнаружена следующая проблема и, будем надеяться, опубликована, а не тайно использована каким-нибудь правительственным агентством или мафией.

Совместимость и браузерные войны

На ранних этапах развития Web на рынке доминировал браузер Mosaic. Через несколько лет баланс сместился в сторону Netscape, а его, в свою очередь, в значительной степени потеснил Microsoft Internet Explorer. Всегда, когда доминирует какой-то один браузер, его поставщик будет иметь право в одностороннем порядке изобретать новые функции для Web. Поскольку большинство пользователей применяют самый популярный браузер, сайты просто начнут задействовать эти функции, не обращая внимания на другие браузеры.

Это были времена темного средневековья в области совместимости, часто называемые браузерными войнами. Перед веб-разработчиками оказалась не единая унифицированная сеть Web, а две или три несовместимые платформы. Хуже того, все браузеры, которые использовались в 2003 году, были полны ошибок, и, конечно же, у каждого браузера ошибки были свои. Это были тяжелые времена для тех, кто писал веб-страницы.

В конце 2000-х годов Mozilla Firefox, некоммерческая ветвь компании Netscape, бросила вызов позиции Internet Explorer. Поскольку компания Microsoft в то время была не особенно заинтересована в сохранении конкурентоспособности, браузер Firefox отнял у нее большую долю рынка. Примерно в то же время Google представила свой браузер Chrome, а браузер Apple Safari приобрел популярность. Это привело к ситуации, что вместо одного крупного игрока на рынке появилось четыре.

У новых игроков было более серьезное отношение к стандартам и рекомендуемым инженерным методам, что позволило улучшить совместимость и уменьшить количество ошибок. Microsoft, видя, что ее доля на рынке падает, поменяла убеждения и приняла эти подходы в своем браузере Edge, который пришел на смену Internet Explorer. Если вы только сейчас начинаете изучать веб-разработку, считайте себя счастливчиком. Последние версии основных браузеров ведут себя достаточно единообразно и содержат относительно мало ошибок.

14 Объектная модель документа

> Довольно скверно! Опять старая история! Окончив постройку дома, замечаешь, что при этом незаметно научился кое-чему, что непременно нужно было знать, прежде чем начинать постройку.
>
> *Фридрих Ницше.*
> *По ту сторону добра и зла*

Когда вы открываете веб-страницу в браузере, браузер получает HTML-текст страницы и выполняет его синтаксический анализ — так же, как наш анализатор из главы 12 разбирал программы. Браузер строит модель структуры документа и использует ее для воспроизведения страницы на экране.

Такое представление документа — один из аттракционов «песочницы» для JavaScript-программ. Это структура данных, которую можно читать и изменять. Она действует как *динамическая* структура данных: когда она модифицируется, страница на экране обновляется в соответствии с изменениями.

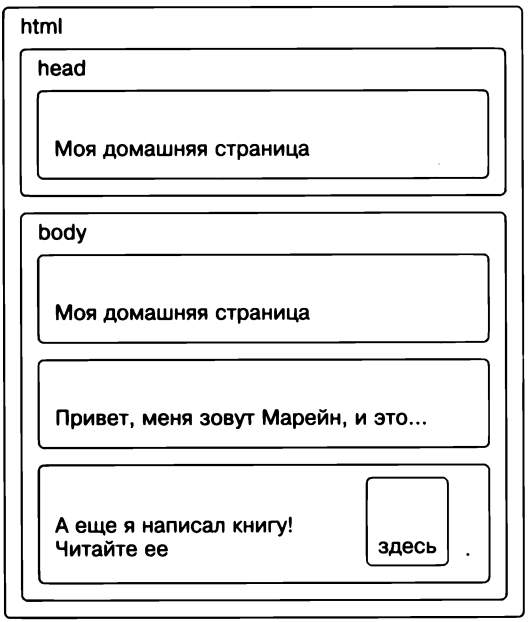
Структура документа

HTML-документ можно представить как набор вложенных блоков. Внутри таких тегов, как `<body>` и `</body>`, содержатся другие теги, которые, в свою очередь, содержат третьи теги или текст. Вот пример документа из главы 13:

```
<!doctype html>
<html>
```

```
  <head>
    <title>Моя домашняя страница</title>
  </head>
  <body>
    <h1>Моя домашняя страница</h1>
    <p>Привет, меня зовут Марейн, и это моя домашняя страница.</p>
    <p>А еще я написал книгу! Читайте ее
      <a href="http://eloquentjavascript.net">здесь</a>.</p>
  </body>
</html>
```

Эта страница имеет следующую структуру:



Структура данных, которую браузер использует для представления документа, соответствует этой форме. Для каждого блока существует объект, с которым можно взаимодействовать, чтобы выяснить, например, какой HTML-тег он представляет, какие поля и текст содержит. Такое представление называется *объектной моделью документа* — Document Object Model (DOM).

Доступ к этим объектам обеспечивает глобальная привязка `document`. Его свойство `documentElement` ссылается на объект, представляющий тег `<html>`.

Поскольку у каждого HTML-документа есть заголовок и тело, у `document` также есть свойства `head` и `body`, указывающие на эти элементы.

Деревья

Вспомним синтаксические деревья, описанные в главе 12. Их структура поразительно похожа на структуру документа в браузере. Каждый узел может ссылаться на другие узлы — *потомков*, — которые, в свою очередь, могут иметь собственных потомков. Такая форма типична для вложенных структур, элементы которых могут содержать подобные им подэлементы.

Если структура данных имеет разветвленную структуру, не содержит циклов (узел не может содержать себя, явно или неявно) и обладает единственным, четко определенным *корнем*, то такую структуру называют *деревом*. Корнем DOM является `document.documentElement`.

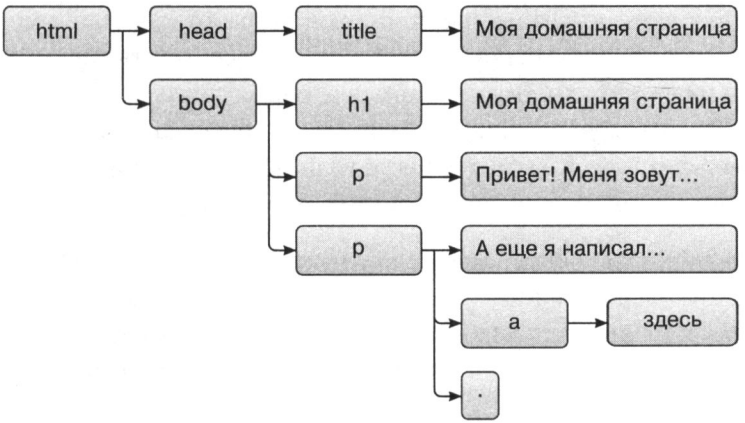
В информатике деревья используются очень широко. Помимо представления рекурсивных структур, таких как HTML-документы или программы, они часто применяются для обслуживания отсортированных множеств данных, поскольку их элементы обычно эффективнее найти или вставить в дерево, чем в плоский массив.

Как правило, дерево имеет узлы разных видов. Синтаксическое дерево для языка Egg содержит идентификаторы, значения и узлы приложения. Узлы приложения могут иметь дочерние элементы, тогда как идентификаторы и значения являются *листьями* — узлами без дочерних элементов.

То же самое касается DOM. Узлы для *элементов*, представляющих теги HTML, определяют структуру документа. У них могут быть дочерние узлы. Примером такого узла является `document.body`. Некоторые из таких потомков могут быть узлами-листьями, такими как фрагменты текста или узлы комментариев.

Каждый объект узла DOM имеет свойство `nodeType`, которое содержит код (число), идентифицирующий тип узла. Элементы имеют код 1, также определенный как константное свойство `Node.ELEMENT_NODE`. Текстовые узлы, представляющие фрагмент текста в документе, получают код 3 (`Node.TEXT_NODE`). Комментарии имеют код 8 (`Node.COMMENT_NODE`).

Другой вариант визуализации нашего дерева документов выглядит так:



Листья являются текстовыми узлами, а стрелки указывают на отношения между узлами типа «потомок — родитель».

Стандарт

Использование загадочных числовых кодов для представления типов узлов не очень свойственно JavaScript. Позже в этой главе мы увидим, что другие части интерфейса DOM тоже выглядят громоздкими и чуждыми. Причина в том, что модель DOM была разработана не только для JavaScript. Скорее, она стремится быть независимым от языка интерфейсом и его можно использовать и в других системах — не только для HTML, но также и для XML, который является обобщенным форматом данных с HTML-подобным синтаксисом.

Это неудачный подход. Стандарты часто бывают полезны. Но в данном случае преимущество (межъязыковая согласованность) не так уж велико. Наличие интерфейса, должным образом интегрированного с используемым языком, сэкономит больше времени, чем знакомый интерфейс для разных языков.

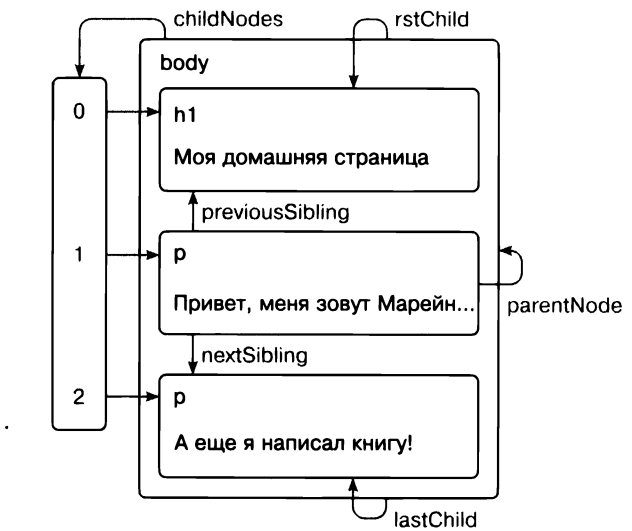
В качестве примера такой неудачной интеграции рассмотрим свойство `childNodes`, которым обладают узлы элементов в DOM. Данное свойство содержит массивоподобный объект со свойством `length` и другими свойствами, помеченными числами и обеспечивающими доступ к дочерним узлам. Но это не настоящий массив, а экземпляр типа `NodeList`, поэтому у него нет таких методов, как `slice` и `map`.

Другие проблемы вызваны просто плохой разработкой. Например, не существует способа создать новый узел и сразу же добавить к нему дочерние элементы или атрибуты. Вместо этого приходится сначала создать узел, а затем добавить к нему дочерние элементы и атрибуты по очереди, используя побочные эффекты. Код, интенсивно взаимодействующий с DOM, часто становится длинным, повторяющимся и безобразным.

Но эти недостатки не являются фатальными. Поскольку JavaScript позволяет создавать собственные абстракции, можно разработать улучшенные способы выражения выполняемых операций. Такие инструменты входят в состав многих библиотек, предназначенных для программирования браузеров.

Перемещения по дереву

Узлы DOM содержат множество ссылок на соседние узлы. Это показано на следующей схеме:



На схеме приведена только одна ссылка каждого типа, но у каждого узла есть свойство `parentNode`, указывающее на узел, частью которого он является (если таковой имеется). Аналогично у каждого узла элемента (тип узла 1) имеется свойство `childNodes`, которое указывает на массивоподобный объект, содержащий его дочерние элементы.

Теоретически можно перемещаться по дереву в любом направлении, используя только эти родительские и дочерние ссылки. Но JavaScript также дает доступ к нескольким дополнительным удобным ссылкам. Свойства `firstChild` и `lastChild` указывают на первый и последний дочерние элементы или имеют значение `null` для узлов без дочерних элементов.

Аналогично `previousSibling` и `nextSibling` указывают на соседние узлы, которые имеют одного и того же родителя и стоят непосредственно перед самим узлом или после него. Для первого потомка `previousSibling` будет равен `null`, а для последнего потомка `nextSibling` будет равен `null`.

Существует также свойство `children` — оно похоже на `childNodes`, но содержит не все дочерние узлы, а только дочерние элементы (тип 1). Это может быть полезно, если вас не интересуют текстовые узлы.

При работе с подобной вложенной структурой данных часто бывают полезны рекурсивные функции. Следующая функция сканирует документ, находит текстовые узлы, содержащие заданную строку, и возвращает `true`, если таковая найдена:

```
function talksAbout(node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

Поскольку `childNodes` не является настоящим массивом, мы не можем перебирать его в цикле с помощью `for/of` и вынуждены работать с индексным диапазоном, используя обычный цикл `for` или `Array.from`.

Свойство текстового узла `nodeValue` включает строку текста, которая содержится в этом узле.

Поиск элементов

Навигация по этим ссылкам между родителями, детьми и узлами одного уровня часто бывает полезна. Но если мы хотим найти конкретный узел документа, добираться до него, начав с `document.body` и следуя далее по фиксированному пути свойств, — плохая идея. Поступать так — значит исходить из предположения, что структура документа является фиксированной. А ведь мы, возможно, впоследствии захотим эту структуру изменить. Другим усложняющим фактором является то, что текстовые узлы создаются даже для пробелов между узлами. Тег `<body>` в нашем примере документа в действительности имеет не три дочерних элемента (`<h1>` и два элемента `<p>`), а семь: кроме этих трех, еще пробелы до, после и между ними.

Поэтому, если нам нужно получить атрибут ссылки `href` из этого документа, то едва ли мы захотим строить инструкцию по принципу: «Получить второй дочерний элемент шестого дочернего элемента тела документа». Было бы лучше, если бы мы могли сказать: «Получить первую ссылку в документе». И мы можем это сделать.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

Каждый узел-элемент имеет метод `getElementsByTagName`, собирающий все элементы с заданным именем тега, которые являются потомками (прямыми или косвенными дочерними элементами) этого узла, и возвращающий их в виде массивоподобного объекта.

Для того чтобы найти *один* определенный узел, можно присвоить ему атрибут `id` и использовать вместо `getElementsByTagName` метод `document.getElementById`.

```
<p>Мой страус Гертруда:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

Третий подобный метод — `getElementsByClassName`; он, подобно `getElementsByTagName`, просматривает содержимое узла элемента и извлекает все элементы, которые имеют заданную строку в атрибуте `class`.

Изменение документа

В структуре данных DOM может быть изменено практически все. Чтобы изменить форму дерева документа, нужно изменить отношения между родительскими и дочерними узлами. Для узлов существует метод `remove`, позволяющий удалить их из их текущего родительского узла. Чтобы добавить дочерний узел к узлу элемента, можно использовать метод `appendChild`, который помещает новый узел в конец списка дочерних элементов, или `insertBefore`, вставляющий узел, указанный в качестве первого аргумента, перед узлом, заданным в качестве второго аргумента.

```
<p>Один</p>
<p>Два</p>
<p>Три</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

Узел может существовать в документе только в одном месте. Таким образом, если вставить абзац `Три` перед абзацем `Один`, то сначала абзац `Три` будет удален с конца документа, а затем вставлен в начале, в результате чего абзацы будут идти в последовательности `Три/Один/Два`. Все операции, которые вставляют узел куда-либо, будут в качестве побочного эффекта приводить к его удалению из текущей позиции (если он там есть).

Метод `replaceChild` используется для замены дочернего узла другим узлом. Он принимает в качестве аргументов два узла: новый и заменяемый. Заменяемый узел должен быть дочерним элементом узла, для которого вызывается метод. Обратите внимание, что и `replaceChild`, и `insertBefore` в качестве первого аргумента ожидают *новый* узел.

Создание узлов

Предположим, что мы хотим написать сценарий, который заменял бы каждое изображение (тег `<img>`) в документе текстом, содержащимся в его атрибуте `alt` — альтернативном текстовом представлении изображения.

Для этого нужно не только удалить изображение, но и добавить вместо него новый текстовый узел. Текстовые узлы создаются с помощью метода `document.createTextNode`.

```
<p> <img src="img/cat.png" alt="Cat"> В
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Заменить</button></p>

<script>   function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Принимая на входе строку, `createTextNode` возвращает текстовый узел, который можно вставить в документ, чтобы он появился на экране.

Цикл, перебирающий изображения, начинает с конца списка. Это необходимо, потому что список узлов, возвращаемый методом, подобным `getElementsByTagName` (или свойством, подобным `childNodes`), является *динамическим*. Другими словами, он обновляется по мере изменения документа. Если бы мы начали с начала, то удаление первого изображения привело бы к тому, что список потерял бы первый элемент, так что при втором прохождении цикла — для `i`, равного 1, — цикл остановился бы, потому что длина коллекции теперь также была бы равна 1.

Если вам нужна *фиксированная*, а не динамическая коллекция узлов, то можно преобразовать коллекцию в настоящий массив, вызвав `Array.from`.

```
let arrayish = {0: "один", 1: "два", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ОДИН", "ДВА"]
```

Для того чтобы создать узлы-элементы, можно использовать метод `document.createElement`. Этот метод принимает имя тега и возвращает новый пустой узел заданного типа.

В следующем примере определяется утилита `elt`, которая создает узел-элемент и рассматривает остальные аргументы как дочерние узлы этого узла. Затем `elt`-функция используется для добавления ссылки на цитату.

```
<blockquote id="quote">
  Работа ни над одной книгой никогда не может считаться полностью
  завершенной. Работая над книгой, мы так много постигаем, что момент
  ее завершения нам всегда кажется преждевременным.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child != "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(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>
```

Вот как выглядит готовый документ:

> **Работа ни над одной книгой никогда не может считаться полностью завершенной. Работая над книгой, мы так много постигаем, что момент ее завершения нам всегда кажется преждевременным.**
>
> **Карл Поппер, предисловие ко второму изданию** *"Открытое общество и его враги"*, **1950**

Атрибуты

Некоторые атрибуты элементов, такие как `href` для ссылок, могут быть доступны через одноименное свойство DOM-объекта соответствующего элемента. Это касается наиболее часто используемых стандартных атрибутов.

Но HTML позволяет назначить узлу любой атрибут. Это бывает полезно, поскольку позволяет хранить в документе дополнительную информацию. Однако если создать собственные имена атрибутов, то такие атрибуты не будут представлены в качестве свойств узла элемента. Вместо этого для работы с такими атрибутами нужно использовать методы `getAttribute` и `setAttribute`.

```
<p data-classified="secret">Код загрузки - 00000000.</p>
<p data-classified="unclassified">У меня две ноги.</p>

<script>
    let paras = document.body.getElementsByTagName("p");
    for (let para of Array.from(paras)) {
      if (para.getAttribute("data-classified") == "secret") {
        para.remove();
      }
    }
</script>
```

Рекомендуется ставить перед именами таких «самодельных» атрибутов префикс `data-` во избежание конфликтов с какими-либо другими атрибутами.

Существует широко используемый атрибут `class`, который является ключевым словом в языке JavaScript. По историческим причинам — отдельные старые реализации JavaScript не могли обрабатывать имена свойств, совпадающие с ключевыми словами, — свойство, применяемое для доступа к этому атрибуту, называется `className`. Вы также можете получить к нему доступ через его настоящее имя `"class"`, используя методы `getAttribute` и `setAttribute`.

Разметка

Возможно, вы заметили, что различные типы элементов отображаются поразному. Некоторые, такие как абзацы (`<p>`) или заголовки (`<h1>`), занимают всю ширину документа и размещаются в отдельных строках. Это так называемые *блочные* элементы. Другие, такие как ссылки (`<a>`) или элемент `<strong>`, отображаются в той же строке, что и окружающий их текст. Они называются *строковыми* элементами.

Для любого документа браузер может сформировать разметку, которая определит размер и положение каждого элемента в зависимости от его типа и содержимого. Затем эта разметка используется для фактического отображения документа.

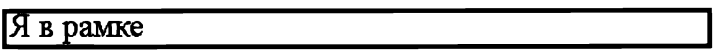
Данные о размере и положении элемента доступны из JavaScript. Свойства `offsetWidth` и `offsetHeight` содержат информацию о пространстве, занимаемом элементом, в *пикселах*. Пиксел — это основная единица измерения в браузере. Традиционно он соответствует наименьшей точке, которую можно нарисовать на экране. Но на современных дисплеях, способных рисовать *очень* маленькие точки, это уже не всегда так и пиксел браузера может охватывать несколько экранных точек.

Аналогично свойства `clientWidth` и `clientHeight` содержат данные о размере пространства *внутри* элемента, без учета ширины рамки.

```
<p style="border: 3px solid red">
  Я в рамке
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

Если задать для абзаца параметры рамки, то вокруг абзаца появится прямоугольник.



Наиболее эффективным способом определить точное положение элемента на экране является метод `getBoundingClientRect`. Он возвращает объект со свойствами `top`, `bottom`, `left` и `right`, показывающими расстояния в пикселах сторон элемента относительно верхнего левого угла экрана. Если вы хотите получить эти данные относительно всего документа, то нужно прибавить к ним текущую позицию прокрутки, которую вы найдете в привязках `pageXOffset` и `pageYOffset`.

Создание разметки документа иногда требует довольно большой работы. Из соображений скорости работы движки браузеров не пересчитывают разметку и не обновляют вид документа на экране каждый раз, когда вы его изменяете, а ждут до последней возможности. Когда программа JavaScript, изменившая документ, завершает работу, браузер должен вычислить новую разметку, чтобы вывести измененный документ на экран. Когда программа *запрашивает* положение или размер чего-либо, читая такие свойства, как

`offsetHeight`, или вызывая `getBoundingClientRect`, предоставление корректной информации также требует вычисления разметки.

Программа, которая постоянно переключается между чтением информации о разметке DOM и изменением DOM, приводит к большому количеству вычислений разметки и, следовательно, будет работать очень медленно. Примером является следующий код. В нем содержатся две разные программы, составляющие строку из символов *X* шириной 2000 пикселов и измеряющие время, которое на это тратится.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Текущее время в миллисекундах
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

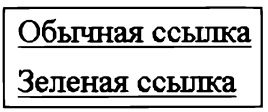
Стили

Как мы уже видели, разные HTML-элементы выводятся на экран поразному. Одни отображаются в виде блоков, другие встроены в строку. Одни добавляют стиль: `<strong>` выделяет текст жирным шрифтом, а `<a>` — синим цветом и подчеркиванием.

То, как тег `<img>` показывает изображение или тег `<a>` привязывает к элементу ссылку, которая открывается, если щелкнуть на этом элементе, сильно зависит от типа элемента. Но мы можем изменить связанный с элементом стиль, такой как цвет текста или подчеркивание. Вот пример, в котором используется свойство `style`:

```
<p><a href=".">Обычная ссылка</a></p>
<p><a href="." style="color: green">Зеленая ссылка</a></p>
```

Вторая ссылка будет зеленого цвета вместо цвета, предлагаемого по умолчанию.

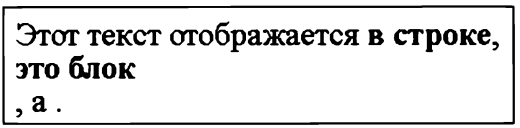


Атрибут стиля может содержать одно или несколько *объявлений*, представляющих собой свойство (например, `color`), после которого стоит двоеточие и значение (например, `green`). Если объявлений несколько, то они должны быть разделены точками с запятой, как здесь: `"color: red; border: none"`.

На стили документа влияют различные аспекты. Например, свойство `display` определяет, отображается элемент как блок или как строчный элемент.

```
Этот текст отображается <strong>в строке</strong>,
<strong style = "display: block"> это блок </strong>, а
<strong style = "display: none">этот вообще не виден</strong>.
```

Тег с параметром `block` приведет к тому, что элемент будет занимать отдельную строку, поскольку блочные элементы не отображаются в одном абзаце с окружающим их текстом. Последний элемент вообще не будет виден — `display: none` отменяет отображение элемента на экране. Это способ скрывать элементы, который часто лучше, чем полностью удалить их из документа, потому что тогда будет легче впоследствии обнаружить такие элементы.



Код JavaScript позволяет напрямую манипулировать стилем элемента через свойство `style`. Оно содержит объект, свойства которого соответствуют

всем возможным свойствам стиля. Их значения — это строки; если вписать в них значения, то можно изменить конкретный аспект стиля элемента.

```
<p id="para" style="color: purple">
  Симпатичный текст
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Некоторые имена свойств стиля содержат дефисы, такие как `font-family`. Поскольку в JavaScript с такими именами неудобно работать (нужно было бы написать `style["font-family"]`), то из имен таких свойств в объекте `style` дефисы удалены, а буквы после дефисов заменены заглавными (`style.fontFamily`).

Каскадные стили

Система стилей для HTML называется CSS, что расшифровывается как *Cascading Style Sheets* (каскадные таблицы стилей). *Таблица стилей* — это набор правил о стилях элементов в документе. Такая таблица может размещаться внутри тега `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Был <strong>текст жирным шрифтом</strong>, а стал курсивом серого
цвета.</p>
```

Слово *«каскадные»* в названии означает возможность объединения нескольких таких правил, чтобы сформировать окончательный стиль элемента. В данном примере стандартный стиль тегов `<strong>`, согласно которому им присваивается свойство `font-weight: bold`, перекрывается правилом в теге `<style>`, добавляющим свойства `font-style` и `color`.

Когда несколько правил определяют значение для одного и того же свойства, последнее прочитанное правило получает более высокий приоритет и является определяющим. Поэтому если правило в теге `<style>` включает в себя `font-weight: normal`, что противоречит стандартному правилу `font-weight`, то текст будет обычным, *а не полужирным*. Стили в атрибуте `style`, применяемые непосредственно к узлу, имеют наивысший приоритет и всегда являются определяющими.

В правилах CSS можно ориентироваться не только на имена тегов. Правило для `.abc` будет применяться ко всем элементам, у которых в атрибуте `class` есть значение `"abc"`. Правило для `#xyz` применяется к элементу, у которого атрибут `id` имеет значение `"xyz"` (в документе должен быть только один такой элемент).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* Элементы p, у которых id равен main и которым присвоены классы a и b */
p#main.a.b {
  margin-bottom: 20px;
}
```

Правило приоритета, согласно которому выполняется правило, определенное последним, применяется только в том случае, если правила имеют одинаковую *специфичность*. Специфичность правила — это мера того, насколько точно оно описывает соответствующие элементы. Специфичность определяется количеством и видом (тег, класс или идентификатор) указанных в правиле аспектов элемента. Например, правило, определенное для `p.a`, является более специфичным, чем правила, определенные только для `p` или только для `.a`, и, таким образом, правило для `p.a` будет иметь над ними приоритет.

Обозначение `p > a ... {}` означает, что данные стили относятся ко всем тегам `<a>`, которые являются прямыми потомками тегов `<p>`. Аналогично правило `p a ... {}` будет применяться ко всем тегам `<a>`, расположенным внутри тегов `<p>`, независимо от того, выступают они прямыми или косвенными дочерними элементами.

Селекторы запросов

В этой книге мы не будем использовать таблицы стилей слишком часто. Их понимание полезно при программировании в браузере, но их сложности хватит на отдельную книгу.

Основная причина, по которой я описал здесь *селекторный* синтаксис — способ записи, используемый в таблицах стилей для определения множества элементов, к каким применяется данный набор стилей, — заключается в том, что мы можем использовать этот же мини-язык как эффективный способ поиска элементов DOM.

Метод `querySelectorAll`, который определен как для объекта `document`, так и для узлов-элементов, принимает строку селектора и возвращает список типа `NodeList`, содержащий все соответствующие селектору элементы.

```
<p>Знай, что, если ты отправишься в погоню за
  <span class="animal">белым кроликом,</span></p>
<p>То рано или поздно упадешь.</p>
<p>Tell 'em a <span class="character">Объясни им, что
  <span class="animal">Гусеница</span></span></p>
<p>Позвала тебя</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 внутри <p>
  // → 2
  console.log(count("p > .animal"));  // Прямой потомок <p>
  // → 1
</script>
```

В отличие от таких методов, как `getElementsByTagName`, объект, возвращаемый `querySelectorAll`, *не является* динамическим. Он не изменится при изменении документа. Но это все еще не настоящий массив, так что, если вы захотите обращаться с ним как с массивом, вам все равно придется вызвать `Array.from`.

Метод `querySelector` (без `All`) работает аналогичным образом. Он полезен, если вам нужен конкретный, единственный элемент. Метод вернет только первый элемент, соответствующий селектору, или `null`, если ни один элемент не подходит.

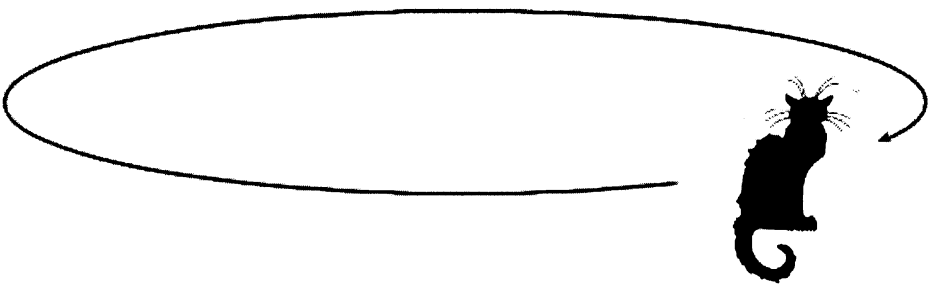
Позиционирование и анимация

Свойство стиля `position` очень сильно влияет на разметку. По умолчанию его значение равно `static` — это означает, что элемент находится на своем обычном месте в документе. Если присвоить этому свойству значение `relative`, то элемент все еще будет занимать место в документе, но теперь можно будет использовать его свойства стиля `top` и `left` для перемещения элемента относительно этого обычного места. Если `position` равно `absolute`, то элемент будет удален из обычной последовательности элементов в документе — другими словами, он больше не будет занимать место и может перекрываться другими элементами. Кроме того, его свойства `top` и `left` можно использовать для абсолютного позиционирования элемента относительно верхнего левого угла ближайшего элемента верхнего уровня, у которого свойство `position` не равно `static`, или относительно всего документа, если такого элемента верхнего уровня не существует.

Мы можем использовать это обстоятельство для создания анимации. В следующем документе выводится картинка с кошкой, которая движется по эллипсу:

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

Серая стрелка показывает путь, по которому движется изображение.



Наша картинка расположена по центру страницы, и значение ее свойства `position` равно `relative`. Для перемещения картинки мы будем многократно обновлять ее стили `top` и `left`.

Сценарий использует функцию `requestAnimationFrame`, чтобы запланировать запуск функции `animate` всякий раз, когда браузер будет готов перерисовать изображение на экране. Сама функция `animate` тоже вызывает `requestAnimationFrame`, чтобы запланировать следующее обновление. Когда окно (или вкладка) браузера активно, обновления будут происходить со скоростью около 60 в секунду, что приведет к красивой анимации.

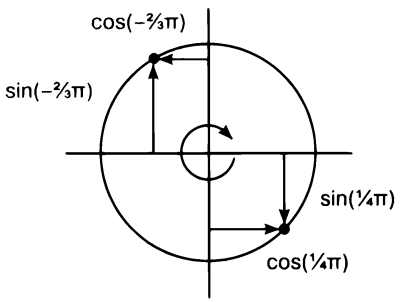
Если бы мы просто обновляли DOM в цикле, то страница зависла бы и на экране ничего не появилось. Браузеры не обновляют экран во время работы JavaScript-программы и не допускают взаимодействия со страницей в это время. Именно поэтому нам нужна функция `requestAnimationFrame` — она сообщает браузеру, что мы пока что закончили и браузер может заняться своими делами, такими как обновление экрана и реагирование на действия пользователя.

Функции анимации передается текущее время в качестве аргумента. Чтобы обеспечить равномерное движение кошки в течение каждой миллисекунды, функция вычисляет скорость, с которой изменяется угол, как разность между текущим временем и временем последнего запуска функции. Если бы функция просто изменяла угол на фиксированную величину за каждый шаг, то движение было бы прерывистым в то время, когда, например, другая ресурсоемкая задача, выполняемая на том же компьютере, мешала бы запустить функцию в течение доли секунды.

Перемещение по кругу осуществляется с помощью тригонометрических функций `Math.cos` и `Math.sin`. Для тех, кто с ними не знаком, я дам их краткое описание, так как мы иногда будем использовать их в этой книге.

`Math.cos` и `Math.sin` полезны для поиска точек, которые лежат на окружности с центром в точке (0, 0) и радиусом, равным единице. Обе функции интерпретируют свой аргумент как позицию на этой окружности, причем ноль означает крайнюю правую точку. Далее значения увеличиваются, если двигаться по часовой стрелке, пока не достигнут 2π (около 6,28) и круг не замкнется. `Math.cos` возвращает x-координату точки, которая соответствует данной позиции, а `Math.sin` — y-координату. Положения (углы), превышающие 2π или меньшие чем 0, являются действительными — вращение повторяется, так что $a + 2\pi$ соответствует тому же углу, что и a.

Такая единица измерения углов называется радианом. Полный круг равен 2π радиан — то же самое, что 360 градусов при измерении в градусах. Константа π доступна в JavaScript как `Math.PI`.



В коде для анимации с кошкой ведется счетчик `angle` для текущего угла анимации. Счетчик увеличивается каждый раз, когда вызывается функция `animate`. Затем этот угол можно использовать для вычисления текущей позиции элемента изображения. Стиль `top` вычисляется с помощью `Math.sin` и умножается на 20 — это вертикальный радиус нашего эллипса. Стиль `left` основан на `Math.cos` и умножается на 200, так что эллипс сильно вытянут в ширину.

Обратите внимание, что обычно стили требуют *единиц измерения*. В данном случае нам нужно добавить к числу слово `"px"`, чтобы сообщить браузеру, что это значение в пикселах (а не в сантиметрах `"ems"` и не в единицах). Это

легко забыть. Использование чисел без единиц измерения приведет к игнорированию стиля — если только число не равно нулю, что всегда означает одно и то же, независимо от его единиц измерения.

Резюме

Программы на JavaScript могут читать и изменять документ, отображаемый браузером, посредством структуры данных, называемой DOM. Эта структура данных представляет модель документа в браузере. Программа на JavaScript может изменять эту модель и, таким образом, изменять видимый документ.

Модель DOM организована в виде дерева, элементы которого расположены иерархически в соответствии со структурой документа. Объекты, представляющие элементы, имеют свойства, такие как `parentNode` и `childNodes`, которые можно использовать для навигации по этому дереву.

Способ отображения документа может зависеть от *стиля* путем как непосредственного назначения стилей узлам, так и определения правил, которые применяются к соответствующим узлам. Есть много различных свойств стилей, таких как цвет или отображение. Код JavaScript может манипулировать стилем элемента напрямую через его свойство `style`.

Упражнения

Построение таблицы

Таблица на HTML строится с помощью следующей структуры тегов:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
```

```
  </tr>
</table>
```

Внутри тега таблицы `<table>` для каждой *строки* существует тег `<tr>`. Внутри тегов `<tr>` можно поместить элементы ячеек: либо ячейки заголовка (`<th>`), либо обычные ячейки (`<td>`).

Для заданного множества данных о горных вершинах — массива объектов со свойствами `name`, `height` и `place` — создайте структуру DOM для таблицы, в которой перечисляются эти объекты. В таблице должно быть по одному столбцу для каждого ключа и по одной строке для каждого объекта, а вверху еще одна строка заголовка с элементами `<th>`, в которой перечислены имена столбцов.

Запишите это так, чтобы столбцы автоматически создавались из объектов, а первый столбец — из имен свойств.

Поместите полученную таблицу внутри элемента с атрибутом `id`, значение которого равно `"mountains"`, чтобы таблица стала видимой в документе.

Когда у вас это получится, выровняйте ячейки, содержащие числовые значения, по правому краю, задав для них свойство `style.textAlign` со значением `"right"`.

Элементы по имени тега

Метод `document.getElementsByTagName` возвращает все дочерние элементы с заданным именем тега. Реализуйте собственную версию этого метода как функцию, которая принимает в качестве аргументов узел и строку (имя тега) и возвращает массив, содержащий все узлы элемента-потомка с заданным именем тега.

Чтобы найти имя тега элемента, используйте его свойство `nodeName`. Но обратите внимание, что это свойство вернет имя тега в верхнем регистре. Чтобы это исправить, задействуйте методы для работы со строками — `toLowerCase` или `toUpperCase`.

Кошка и ее шляпа

Расширьте описанную ранее функцию анимации с кошкой, чтобы по разные стороны эллипса вращались кошка и ее шляпа (`<img src = "img/hat.png">`).

Или пускай шляпа вращается вокруг кошки. Или измените анимацию другим интересным способом.

Чтобы упростить позиционирование нескольких объектов, возможно, стоит переключиться на абсолютное позиционирование. Тогда значения `top` и `left` будут отсчитываться относительно верхнего левого края документа. Чтобы избежать использования отрицательных координат, из-за которых изображение выходит за пределы видимой страницы, можно прибавить к значениям позиции фиксированное количество пикселов.

15 Обработка событий

> Вы имеете власть над своим
> разумом — но не над событиями.
> Уразумев это, вы обретете силу.
>
> *Марк Аврелий. Наедине с собой*

Есть программы, которые работают непосредственно с пользовательским вводом, таким как действия мыши и клавиатуры. Такого рода входные данные нельзя представить в виде хорошо организованной структуры — они поступают по частям в режиме реального времени, и ожидается, что программа ответит на них так же быстро.

Обработчики событий

Представьте себе интерфейс, в котором единственный способ узнать, была ли нажата клавиша, — это прочитать текущее состояние этой клавиши. Тогда, чтобы иметь возможность реагировать на нажатия клавиш, нам бы потребовалось постоянно считывать состояние клавиши, чтобы его можно было перехватить прежде, чем клавишу отпустят. Выполнять в это время другие ресурсоемкие вычисления было бы опасно, поскольку из-за них можно было бы пропустить нажатие клавиши.

Отдельные примитивные машины действительно обрабатывают входные данные подобным образом. Шагом вперед по сравнению с этим были бы аппаратное обеспечение или операционная система, которые бы фиксировали нажатие клавиши и помещали его в очередь. Тогда программа могла бы периодически проверять очередь на наличие новых событий и реагировать на то, что она там обнаружила.

Конечно, нужно не забывать заглядывать в очередь и делать это часто, поскольку все время между нажатием клавиши и моментом, когда программа

это заметила, ситуация выглядит так, будто программа не реагирует. Такой подход называется *опросом*, и большинство программистов предпочитают его избегать.

Лучше, когда система активно уведомляет наш код о том, что произошло событие. Браузеры для этого позволяют нам регистрировать функции в качестве *обработчиков* для определенных событий.

```
<p>Щелкните на этом документе, чтобы активировать обработчик.</p>
<script>
  window.addEventListener("click", () => {
    console.log("Кто там?");
  });
</script>
```

Привязка `window` указывает на встроенный объект, предоставляемый браузером. Он соответствует окну браузера, в котором содержится документ. При вызове его метода `addEventListener` второй аргумент метода регистрируется как функция, которая будет вызываться всякий раз, когда будет происходить событие, описанное первым аргументом.

События и DOM-узлы

Каждый обработчик событий браузера регистрируется в каком-то контексте. В предыдущем примере мы вызывали `addEventListener` для объекта `window`, чтобы зарегистрировать обработчик для всего окна. Такой же метод существует для элементов DOM и некоторых других типов объектов. Слушатели событий вызываются только тогда, когда событие происходит в контексте объекта, для которого они зарегистрированы.

```
<button>Щелкни здесь</button>
<p>Здесь нет обработчика.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Кнопка нажата.");
  });
</script>
```

В этом примере обработчик присоединен к узлу кнопки. Если щелкнуть на кнопке, то запускается данный обработчик, а если щелкнуть в любой другой точке документа — то нет. Присвоение узлу атрибута `onclick` имеет

аналогичный эффект. Это работает для большинства типов событий: можно прикрепить обработчик через атрибут, именем которого является имя события с приставкой on.

Но у узла может быть только один атрибут onclick, так что таким образом можно присвоить узлу только один обработчик. Метод addEventListener позволяет добавлять любое количество обработчиков, так что можно безопасно добавлять обработчики, даже если у элемента уже один есть.

Метод removeEventListener, вызываемый с аргументами, аналогичными addEventListener, удаляет обработчик.

```
<button>Одноразовая кнопка</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Готово.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

В removeEventListener нужно передать имя той же функции, которая была ранее передана в addEventListener. Таким образом, чтобы отменить регистрацию обработчика, нужно присвоить функции имя (в нашем примере once), и тогда можно будет передавать обоим методам одно и то же функциональное значение.

Объекты событий

До сих пор мы это игнорировали, но функциям — обработчикам событий передается аргумент *«объект события»*. Он содержит дополнительную информацию о событии. Например, если мы хотим узнать, *какая именно* кнопка мыши была нажата, то можем посмотреть на свойство button объекта события.

```
<button>Щелкните любой кнопкой мыши</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Левая кнопка");
```

```
    } else if (event.button == 1) {
      console.log("Средняя кнопка");
    } else if (event.button == 2) {
      console.log("Правая кнопка");
    }
  });
</script>
```

Информация, хранящаяся в объекте события, зависит от типа события. Мы обсудим различные типы далее в этой главе. Свойство `type` объекта события всегда содержит строку, идентифицирующую событие (например, `"click"` или `"mousedown"`).

Распространение событий

Для большинства типов событий обработчики, зарегистрированные на узлах с дочерними элементами, также будут получать события, которые происходят с дочерними элементами. Если щелкнуть на кнопке, находящейся внутри абзаца, то обработчики событий для этого абзаца также увидят это событие.

Но если обработчик есть и у абзаца, и у кнопки, то более специфичный обработчик — тот, что назначен кнопке, — будет выполняться первым. Говорят, что событие *распространяется* наружу, от узла, где оно произошло, к родительскому узлу этого узла и так далее до корня документа. Наконец, после того, как выполнились все обработчики, зарегистрированные для определенного узла, очередь доходит до обработчиков, зарегистрированных для всего окна, и они получают возможность отреагировать на событие.

Чтобы предотвратить получение события следующими обработчиками, обработчик события может в любой момент вызвать для объекта события метод `stopPropagation`. Это бывает полезно, когда, например, есть кнопка внутри другого активного элемента и вы не хотите, чтобы при щелчке на кнопке активировалось поведение внешнего элемента, соответствующее щелчку на этом элементе.

В следующем примере регистрируются обработчики `"mousedown"` как для кнопки, так и для абзаца, внутри которого она находится. При щелчке правой кнопкой мыши обработчик для этой кнопки вызывает `stopPropagation`, что препятствует запуску обработчика для абзаца. Если щелкнуть другой кнопкой мыши, то будут выполнены оба обработчика.

```
<p>Абзац с <button>кнопкой</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Обработчик для абзаца.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Обработчик для кнопки.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

У большинства объектов событий есть свойство `target`, ссылающееся на узел, где возникло событие. Это свойство можно использовать, чтобы убедиться, что вы случайно не обрабатываете событие, которое распространяется от другого узла и которое не нужно обрабатывать.

Можно также применять свойство `target` при создании широкой сети для определенного типа события. Например, если есть узел, содержащий множество кнопок, может оказаться удобнее зарегистрировать обработчик щелчка на внешнем узле и использовать в нем свойство `target`, чтобы определить, какая именно кнопка была нажата, вместо того чтобы регистрировать отдельный обработчик для каждой кнопки.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
        console.log("Щелчок на кнопке", event.target.textContent);
      }
    });
</script>
```

Действия по умолчанию

У многих событий есть действие, определенное для них по умолчанию. Если щелкнуть на ссылке, то вы перейдете к объекту, на который указывает эта ссылка. Если нажать клавишу ↓, то браузер прокрутит страницу вниз. Если щелкнуть правой кнопкой мыши, то откроется контекстное меню. И так далее.

Для большинства типов событий обработчики JavaScript вызываются *прежде*, чем сработает поведение по умолчанию. Если обработчику нежелательно, чтобы выполнялось такое стандартное поведение — обычно потому, что он уже позаботился об обработке события, — то этот обработчик может вызвать метод `preventDefault` для данного объекта события.

Это можно использовать для реализации нестандартных сочетаний клавиш или контекстных меню. Данный метод также позволяет грубо нарушать поведение, которого ожидают пользователи. Например, вот ссылка, по которой нельзя перейти:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Старайтесь не делать подобных вещей, если только для них нет действительно веских причин. Людям, использующим вашу страницу, будет неприятно, если ожидаемое поведение будет нарушено.

В зависимости от браузера некоторые события вообще не могут быть перехвачены. Например, в Chrome сочетание клавиш для закрытия текущей вкладки (Control+W или Command+W) не может быть обработано JavaScript.

События клавиш

При нажатии клавиши браузер активирует событие `"keydown"`. Когда пользователь отпускает клавишу, активируется событие `"keyup"`.

```
<p>Если нажать клавишу V, то эта страница станет фиолетовой.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
```

```
      document.body.style.background = "";
    }
  });
</script>
```

Несмотря на название, событие "keydown" срабатывает не только тогда, когда клавиша физически нажата. Если нажать и удерживать клавишу, то событие будет запускаться при каждом *повторе* клавиши. Иногда с этим следует быть осторожными. Например, если при нажатии клавиши вы вставляете кнопку в DOM и удаляете ее при отпускании клавиши, то пользователь может случайно добавить несколько сотен кнопок, если чуть дольше будет удерживать клавишу нажатой.

В рассмотренном примере было использовано свойство key объекта события, позволяющее увидеть, о какой клавише идет речь. Это свойство содержит строку, которая для большинства клавиш соответствует тому, что будет введено при ее нажатии. Для специальных клавиш, таких как Enter, в этой строке содержится имя (в данном случае "Enter"). Если при нажатии клавиши одновременно нажать и удерживать клавишу Shift, это также может повлиять на имя клавиши: "v" превратится в "V", а "1" — в "!", как при нажатии Shift+1.

Клавиши-модификаторы, такие как Shift, Control, Alt и Meta (на Mac — Command), генерируют события клавиатуры так же, как и обычные клавиши. Но при изучении сочетаний клавиш вы обнаружите, что о том, были ли нажаты эти клавиши, можно узнать из свойств shiftKey, ctrlKey, altKey и metaKey событий клавиатуры и мыши.

```
<p>Для продолжения нажмите Control+Пробел.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Продолжаем!");
    }
  });
</script>
```

Узел DOM, к которому относится событие клавиши, зависит от элемента, на котором находился фокус в момент нажатия клавиши. Большинство узлов не может иметь фокус, если не присвоить им атрибут tabindex, но такие элементы, как ссылки, кнопки и поля формы, — могут. Подробнее поля формы будут описаны в главе 18. Если нет определенного элемента с фокусом, целевым узлом для событий клавиатуры является document.body.

Когда пользователь печатает текст, использовать события клавиатуры для того, чтобы выяснить, что именно вводится, проблематично. Некоторые платформы, особенно виртуальная клавиатура на телефонах Android, не генерируют события клавиш. Но даже если у вас добрая старая клавиатура, определенные варианты ввода текста не соответствуют прямому нажатию клавиш. Например, люди, чьи сценарии работы выходят за рамки стандартной клавиатуры, используют программные *редакторы методов ввода* (Input Method Editor, IME), где для создания символов объединяются несколько нажатий клавиш.

Чтобы обнаружить, что что-то было напечатано, элементы, в которых можно вводить текст, такие как теги `<input>` и `<textarea>`, генерируют события `"input"` всякий раз, когда пользователь изменяет их содержимое. Чтобы получить то, что действительно было введено, лучше всего непосредственно прочитать содержимое поля. О том, как это сделать, вы узнаете в главе 18.

События мыши

В настоящее время существует два распространенных способа указывать на объекты на экране: с помощью мыши (включая устройства, которые действуют аналогично мыши, такие как сенсорные панели и трекболлы) и сенсорных экранов. Они производят различные виды событий.

Щелчки кнопкой мыши

Нажатие кнопки мыши вызывает ряд событий. События `"mousedown"` и `"mouseup"` аналогичны `"keydown"` и `"keyup"` и активируются при нажатии и отпускании кнопки мыши. Это происходит в том узле DOM, который находится непосредственно под указателем мыши, когда происходит событие.

После события `"mouseup"` происходит событие `"click"` на самом внутреннем узле, который содержал как нажатие, так и отпускание кнопки. Например, если нажать кнопку мыши на одном абзаце, а затем переместить указатель на другой абзац и отпустить кнопку, то событие `"click"` произойдет с элементом, который содержит оба этих абзаца.

Если интервал между двумя щелчками достаточно мал, то после второго события `"click"` происходит также событие `"dblclick"` (двойной щелчок).

Чтобы получить точную информацию о месте, в котором произошло событие мыши, можно посмотреть его свойства `clientX` и `clientY`, где содержатся координ-

наты события (в пикселах) относительно верхнего левого угла окна, или pageX и pageY, где хранятся координаты относительно верхнего левого угла всего документа (и которые могут отличаться от clientX и clientY при прокрутке окна).

Далее реализована простейшая программа для рисования. Каждый раз, когда вы щелкаете на документе, программа добавляет точку под указателем мыши. Менее примитивную программу рисования вы найдете в главе 19.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* скругленные углы */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let 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>
```

Движения мыши

При каждом перемещении указателя мыши происходит событие "mousemove". Это событие можно использовать для отслеживания положения мыши. Распространенной ситуацией, в которой это бывает полезно, является реализация какой-либо функции перетаскивания мышью.

Например, следующая программа отображает на экране полоску и настраивает обработчики событий так, чтобы при перетаскивании указателя влево или вправо по панели она становилась уже или шире:

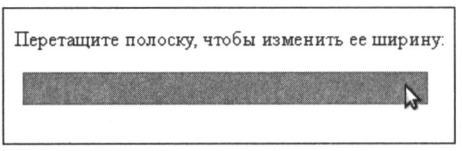
```
<p>Перетащите полоску, чтобы изменить ее ширину:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
```

```
<script>
  let lastX; // Отслеживает x-координату мыши
  let bar = document.querySelector("div")

  bar.addEventListener("mousedown", event => {
    if (event.button == 0) {
      lastX = event.clientX;
       window.addEventListener("mousemove", moved);
       event.preventDefault(); // Заблокировать возможность выделения
    }
  });

    function moved(event) {
      if (event.buttons == 0) {
        window.removeEventListener("mousemove", moved);
      } else {
        let dist = event.clientX - lastX;
        let newWidth = Math.max(10, bar.offsetWidth + dist);
        bar.style.width = newWidth + "px";
        lastX = event.clientX;
      }
  }
</script>
```

Получившаяся страница выглядит так:



Обратите внимание, что обработчик события `"mousemove"` зарегистрирован для всего окна. Даже если при изменении размера, пока кнопка удерживается нажатой, указатель мыши выйдет за пределы полоски, мы все равно хотим, чтобы размер полоски изменялся.

Но когда пользователь отпустит кнопку мыши, изменение размера полоски должно прекратиться. Для этого мы можем использовать свойство `buttons` (обратите внимание на множественное число), которое содержит данные о кнопках мыши, нажатых в данный момент. Когда это свойство равно нулю, ни одна кнопка не нажата. Если нажата одна из кнопок, то значением свойства является сумма кодов для этих кнопок — левая кнопка имеет код 1, правая кнопка — код 2, средняя — код 4. Таким образом, чтобы проверить,

какая кнопка нажата, нужно взять остаток от деления значения `buttons` на код кнопки.

Обратите внимание, что последовательность этих кодов отличается от той, что используется в `button`, где средняя кнопка находится перед правой. Как уже упоминалось, согласованность не относится к сильным сторонам интерфейса программирования браузера.

Сенсорные события

Современный стиль графических браузеров был разработан с расчетом на интерфейс мыши, сенсорные экраны в те времена были редкостью. Чтобы заставить Web «работать» на ранних моделях телефонов с сенсорными экранами, браузеры для этих устройств в определенной степени делали вид, что сенсорные события были событиями мыши. Касания экрана расценивались как события `"mousedown"`, `"mouseup"` и `"click"`.

Но это не очень надежная иллюзия. Сенсорный экран работает иначе, чем мышь: у него нет нескольких кнопок, мы не можем отслеживать положение пальца, когда он не касается экрана (чтобы имитировать `"mousemove"`), зато он позволяет касаться экрана несколькими пальцами одновременно.

События мыши описывают взаимодействие с сенсорным экраном только в простых случаях — если добавить обработчик события `"click"`, то пользователь сенсорного экрана сможет его использовать. Но такие вещи, как изменение размера полоски, описанное в предыдущем примере, для сенсорного экрана не будут работать.

Существуют специальные типы событий, инициируемые сенсорным взаимодействием. Когда палец прикасается к экрану, возникает событие `"touchstart"`. Когда палец перемещается по экрану, возникает событие `"touchmove"`. Наконец, когда палец перестанет касаться экрана, происходит событие `"touchend"`.

Поскольку многие сенсорные экраны позволяют обнаруживать несколько касаний одновременно, у данных событий нет единого связанного с ними набора координат. Вместо этого их объекты событий имеют свойство `touch`, содержащее массивоподобное множество точек, каждая из которых имеет собственные свойства `clientX`, `clientY`, `pageX` и `pageY`.

Например, можно написать такую программу, которая будет показывать красные круги вокруг каждого пальца, касающегося экрана:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Коснитесь этой страницы</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
      }
    }
    window.addEventListener("touchstart", update);
    window.addEventListener("touchmove", update);
    window.addEventListener("touchend", update);
  </script>
```

В сенсорных обработчиках событий часто приходится вызывать `protectDefault`, чтобы переопределить стандартное поведение браузера (которое может включать в себя прокрутку страницы при протягивании пальца по экрану) и предотвращать запуск событий мыши, для которых у вас также может быть *отдельный* обработчик.

События прокрутки

Всякий раз при прокрутке элемента для него происходит событие `"scroll"`. Это можно использовать по-разному — например, чтобы узнать, что в данный момент видит пользователь (для отключения закадровой анимации или для отправки шпионских отчетов в вашу злобную штаб-квартиру), или для отображения какого-нибудь индикатора выполнения (выделяя строку в оглавлении или отображая номер страницы). В следующем примере показан индикатор выполнения прокрутки в документе, который заполняется по мере прокрутки вниз:

```
<style>
  #progress {
```

```
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Создание содержимого
  document.body.appendChild(document.createTextNode(
    "суперархиэкстраультрамегаграндиозно".repeat(1000)));

  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Если задать элементу свойство `position` со значением `fixed`, то результат будет во многом похож на ситуацию, когда свойство `position` равно `absolute`, но только элемент не прокручивается вместе с остальной частью документа. В результате наш индикатор выполнения остается вверху окна. Его ширина изменяется в соответствии с текущим состоянием прокрутки. Мы использовали `%` вместо `px` в качестве единицы измерения ширины, чтобы размер элемента соответствовал ширине страницы.

Глобальная привязка `innerHeight` дает нам высоту окна, которую нужно вычесть из общей высоты прокрутки, — мы не сможем продолжать прокрутку, когда дойдем до низа документа. Существует также привязка `innerWidth` для ширины окна. Разделив `pageYOffset`, текущую позицию прокрутки, на ее максимальную позицию и умножив результат на 100, получим процент для индикатора выполнения.

Вызов `preventDefault` для события прокрутки не отменяет ее, поскольку обработчик события вызывается только *после* прокрутки.

События фокуса

Когда элемент получает фокус, браузер запускает для этого элемента событие `"focus"`. Когда элемент теряет фокус, для данного элемента возникает событие `"blur"`. В отличие от событий, описанных ранее, эти два события

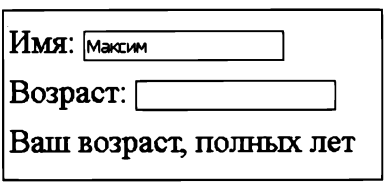
не распространяются. Обработчик родительского элемента не получает уведомление о том, что его дочерний элемент получил или потерял фокус.

В следующем примере отображается текст справки для текстового поля, которое в данный момент получило фокус:

```
<p>Имя: <input type="text" data-help="Ваше полное имя"></p>
<p>Возраст: <input type="text" data-help="Ваш возраст, полных лет"></p>
<p id="help"></p>

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
</script>
```

Вот скриншот, на котором показан справочный текст для поля возраста.



Объект `window` получает события `"focus"` и `"blur"`, когда пользователь переходит с одной вкладки браузера на другую или из одного в другое окно, в котором отображается документ.

Событие загрузки

Когда завершается загрузка страницы, в окне и объектах тела документа возникает событие `"load"`. Это событие часто используется, чтобы запланировать действия по инициализации, к моменту выполнения которых весь документ должен быть сформирован. Помните, что содержимое те-

гов `<script>` выполняется сразу же после обнаружения тега. Иногда это слишком рано — например, если сценарий должен что-то сделать с частями документа, которые расположены после тега `<script>`.

Такие элементы, как изображения и теги сценариев, загружающие внешний файл, также имеют событие `"load"`. В данном случае это событие показывает, что файл, на который оно ссылается, уже загружен. Подобно событиям, связанным с фокусом, события загрузки не распространяются на DOM.

Когда пользователь закрывает страницу или переходит на другую страницу (например, по ссылке), возникает событие `"beforeunload"`. Его главное назначение — предотвратить случайную потерю данных пользователем при закрытии документа. Как и следует ожидать, метод `protectDefault` не препятствует закрытию страницы. Это делается иначе — путем возврата обработчиком ненулевого значения. Если так сделать, браузер покажет пользователю диалоговое окно с вопросом: уверен ли он, что хочет покинуть страницу. Такой механизм гарантирует, что пользователь всегда сможет уйти со страницы, даже если она вредоносная, которая предпочла бы задержать его там навсегда, заставляя смотреть на коварную рекламу средств для потери веса.

События и цикл событий

В контексте цикла событий, описанного в главе 11, обработчики событий браузера ведут себя подобно другим асинхронным уведомлениям. Их выполнение планируется, когда происходит соответствующее событие, но им приходится дождаться завершения других запущенных сценариев, прежде чем они получат возможность выполниться.

Тот факт, что события могут обрабатываться только тогда, когда не выполняется ничего другого, означает, что, если цикл событий связан с другой работой, любое взаимодействие со страницей (которое происходит посредством событий) будет отложено до тех пор, пока не появится время для его обработки. Поэтому, если вы запланируете слишком много работы, либо с длительными обработчиками событий, либо с множеством коротких, страница станет медленной и неудобной в использовании.

На тот случай, если вы *действительно* захотите сделать что-то трудоемкое в фоновом режиме, не останавливая страницу, браузеры предоставляют так

называемую технологию *веб-обработчиков (Web Workers)*. Веб-обработчик — это процесс JavaScript, выполняющийся одновременно с основным сценарием, но по собственной временной шкале.

Предположим, что возведение числа в квадрат — это тяжелое, длительное вычисление, которое мы хотим выполнить в отдельном потоке. Мы могли бы написать файл с именем `code/squareworker.js`, и он отвечал бы на сообщения, вычисляя квадрат числа и возвращая его в сообщении.

```
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```

Чтобы избежать проблем, связанных с тем, что несколько потоков имеют доступ к одним и тем же данным, веб-обработчики не делают свою глобальную область действия или какие-либо другие данные доступными из среды основного сценария. Вместо этого к ним необходимо обращаться, отправляя сообщения туда и обратно.

Следующий код порождает веб-обработчик, который выполняет этот сценарий, отправляет ему несколько сообщений и выводит сообщения, полученные в ответ.

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("Ответ от веб-обработчика:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

Функция `postMessage` отправляет сообщение, которое вызывает у получателя событие `"message"`. Сценарий, создавший веб-обработчик, отправляет и получает сообщения через объект `Worker`, тогда как сам веб-обработчик общается с создавшим его сценарием, отправляя сообщения и ожидая их получения непосредственно в своей глобальной области видимости. В виде сообщений могут быть переданы только значения, представленные в формате JSON, — другая сторона получит не само значение, а его *копию*.

Таймеры

Мы уже встречали функцию `setTimeout` в главе 11. Она планирует выполнение другой функции, которая будет запущена позже, через заданное количество миллисекунд.

Иногда бывает необходимо отменить уже запланированную функцию. Для этого нужно сохранить значение, возвращенное `setTimeout`, и вызвать `clearTimeout` для этого значения.

```
let bombTimer = setTimeout(() => {
  console.log("БАБАХ!");
}, 500);

if (Math.random() < 0.5) { // вероятность 50 %
  console.log("Осечка.");
  clearTimeout(bombTimer);
}
```

Функция `cancelAnimationFrame` работает так же, как и `clearTimeout`, — вызов ее для значения, возвращаемого `requestAnimationFrame`, отменяет этот кадр анимации (при условии, что он еще не был вызван).

Аналогичная пара функций, `setInterval` и `clearInterval`, используется для установки таймеров, выполнение которых должно *повторяться* каждые *X* миллисекунд.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Устранение повторных срабатываний

Некоторые типы событий могут возникать быстро, много раз подряд (например, события `"mousemove"` и `"scroll"`). При обработке таких событий необходимо следить, чтобы не сделать что-то требующее слишком много времени, иначе обработчик будет работать слишком долго и взаимодействие пользователя с документом станет заметно медленнее.

Если вам нужно сделать что-то нетривиальное в таком обработчике, можно воспользоваться `setTimeout`, чтобы гарантировать, что подобное не будет делаться слишком часто. Обычно это называется *подавлением ложных срабатываний события*. В этой области существует несколько слегка различающихся подходов.

В первом примере мы хотим, чтобы обработчик реагировал, когда пользователь что-то ввел с клавиатуры, но не хотим делать это сразу же, для каждого события ввода. Если пользователь печатает быстро, мы хотим лишь дождаться паузы. Вместо немедленного выполнения действия в обработчике события мы устанавливаем задержку. Кроме того, очищаем предыдущую задержку (если таковая имеется), чтобы, если события происходят через малый интервал (меньше, чем задержка), задержка для предыдущего события была отменена.

```
<textarea>Введите здесь какой-нибудь текст...</textarea>
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Текст введен!"), 500);
  });
</script>
```

Если передать `clearTimeout` неопределенное значение или вызвать данную функцию по истечении времени ожидания, это не будет иметь никакого эффекта. Таким образом, можно не заботиться о том, когда вызывать функцию, мы просто делаем это для каждого события.

Если мы хотим, чтобы между откликами был как минимум заданный промежуток времени, но при этом чтобы они происходили *во время* серии событий, а не только после них, можно применить немного другой шаблон. Например, ниже представлено, как реагировать на события `"mousemove"`, показывая текущие координаты мыши, но не каждый раз, а только каждые 250 миллисекунд.

```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
</script>
```

Резюме

Обработчики событий позволяют обнаруживать события, происходящие на веб-странице, и реагировать на них. Для регистрации таких обработчиков используется метод `addEventListener`. Каждое событие имеет свой тип, который его идентифицирует (`"keydown"`, `"focus"` и т. п.). Большинство событий вызывается для определенного элемента DOM и затем *распространяется* на его родительские элементы, что позволяет обработчикам, связанным с этими элементами, обрабатывать указанные события.

При вызове обработчик события получает объект события с дополнительной информацией о событии. У этого объекта также есть методы, которые позволяют останавливать дальнейшее распространение события (`stopPropagation`) и предотвращать его обработку по умолчанию в браузере (`protectDefault`).

Нажатие клавиши вызывает события `"keydown"` и `"keyup"`. Нажатие кнопки мыши вызывает события `"mousedown"`, `"mouseup"` и `"click"`. Перемещение мыши вызывает событие `"mousemove"`. Взаимодействие с сенсорным экраном приводит к событиям `"touchstart"`, `"touchmove"` и `"touchend"`.

Прокрутка может быть обнаружена с помощью события `"scroll"`, а изменение фокуса — с помощью событий `"focus"` и `"blur"`. После окончания загрузки документа в окне возникает событие `"load"`.

Упражнения

Воздушный шарик

Напишите страницу, которая отображает воздушный шарик (с помощью смайлика 🎈). При нажатии стрелки вверх шарик должен надуваться (увеличиваться) на 10 %, а при нажатии стрелки вниз — сдуваться (уменьшаться) на 10 %.

Для того чтобы управлять размером текста (смайлики — это текст), вы можете использовать CSS-свойство `font-size` (`style.fontSize`) для его родительского элемента. Не забудьте указать в значении единицу измерения — например, пикселы (`10px`).

Имена клавиш со стрелками — `"ArrowUp"` и `"ArrowDown"`. Убедитесь, что нажатия клавиш меняют только размер шарика и не приводят к прокрутке страницы.

Когда это сработает, добавьте еще одно свойство: если надуть шарик больше определенного размера, он лопнет. В данном случае взрыв означает замену шарика на смайлик 💥 с одновременным удалением обработчика события (так что вы не сможете больше надувать или сдувать лопнувший шарик).

След мыши

В первое время существования JavaScript, когда вошли в моду яркие домашние страницы с множеством анимированных изображений, было придумано несколько по-настоящему вдохновляющих способов использования языка. Одним из них был *след мыши* — серия элементов, которые тянулись за указателем при его перемещении по странице.

В этом упражнении вы попробуете реализовать след мыши. Используйте элементы `<div>` с фиксированным размером, заданным цветом фона и абсолютным позиционированием. Создайте группу таких элементов и при перемещении мыши отображайте их в виде указателя мыши.

Здесь возможны различные подходы. Можно сделать решение настолько простым или сложным, насколько вы захотите. Начните с простого решения — сохраните фиксированное количество элементов, образующих след и, перебирая их в цикле, перемещайте каждый следующий элемент в текущее положение мыши всякий раз, когда происходит событие `mousemove`.

Вкладки

Панели с вкладками очень распространены в пользовательских интерфейсах. Они позволяют выбрать интерфейсную панель, перейдя на одну из нескольких вкладок, «торчащих» над элементом.

В этом упражнении вам предлагается реализовать простой интерфейс с вкладками. Напишите функцию `asTabs`, которая принимает узел DOM и создает интерфейс с вкладками, показывающий дочерние элементы этого узла. Она должна вставлять список элементов `<button>` в верхней части узла, по одному для каждого дочернего элемента, содержащего текст, полученный из атрибута `data-tabname` дочернего элемента. Все исходные дочерние элементы, кроме одного, должны быть скрыты (свойство `display` имеет значение `none`). Чтобы выбрать узел, который должен быть видимым, нужно нажать его кнопку.

Когда это заработает, дополните интерфейс, чтобы кнопка выбранной вкладки имела другой стиль и чтобы было понятно, какая вкладка выбрана.

16 Проект: игровая платформа

Любая реальность — это игра.

Иэн Бэнкс. Игрок

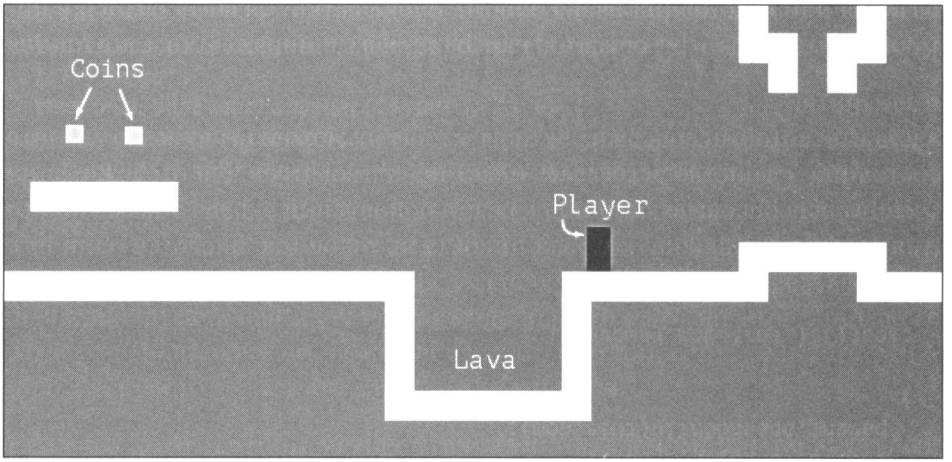
В детстве мое увлечение компьютерами, как и у многих ботаников, сводилось по большей части к компьютерным играм. Меня поглотили миниатюрные модели миров, которыми я мог управлять и в которых разворачивались своего рода истории — полагаю, скорее благодаря тому, что я в них вкладывал усилия воображения, чем вследствие возможностей, предлагаемых ими на самом деле.

Я никому не пожелаю карьеры в программировании игр. Подобно музыкальной индустрии, несоответствие между количеством молодых людей, желающих работать в данной области, и реальным спросом на таких людей создает довольно нездоровую среду. Однако писать игры для развлечения бывает забавно.

В этой главе будет рассказано о реализации небольшой игровой платформы. В платформенных играх (или играх типа «прыгай и беги») игрок должен перемещать фигуру по миру, который обычно является двумерным; когда игрок перепрыгивает препятствия или запрыгивает на них, он виден сбоку.

Игра

Наша игра будет приблизительно напоминать Dark Blue (www.lessmilk.com/games/10) Томаса Пейлфа. Я выбрал эту игру потому, что она одновременно и развлекательная, и минималистичная, и еще потому, что для ее построения не требуется слишком много кода. Сама игра выглядит так:



Темный прямоугольник — это игрок, чья задача — собрать желтые квадратики (монеты), избегая при этом красных квадратов (лава). Уровень считается пройденным, когда собраны все монеты.

Для передвижения игрока используются стрелки влево и вправо, а для прыжков — стрелка вверх. Прыжки — это особенность данного игрового персонажа. Он может подпрыгивать в несколько раз выше своего роста и менять направление в воздухе. Это не очень реалистично, но зато позволяет игроку почувствовать прямой контроль над своим экранным воплощением.

Игра состоит из статичного фона, представленного в виде сетки, и движущихся элементов, наложенных на этот фон. Каждое поле в сетке может быть пустым, сплошным или заполненным лавой. Движущимися элементами являются игрок, монеты и отдельные кусочки лавы. Позиции данных элементов не ограничены сеткой — их координаты могут быть дробными, что обеспечивает плавное перемещение.

Технология

Для отображения игры мы будем применять браузерную DOM-модель и считывать данные, вводимые пользователем, обрабатывая ключевые события.

Код, связанный с экраном и клавиатурой, — это лишь малая часть работы, которую нужно проделать, чтобы создать такую игру. Поскольку все здесь выглядит как цветные прямоугольники, нарисовать игру несложно: нужно

лишь создать элементы DOM и использовать стили, чтобы придать им цвет фона, размер и положение.

Поскольку фон — это неизменная сетка квадратов, мы можем представить его в виде таблицы. Свободно движущиеся элементы могут перекрывать друг друга посредством абсолютного позиционирования элементов.

В играх и других программах, где применяется анимированная графика и нужно без заметной задержки реагировать на данные, вводимые пользователем, особенно важна эффективность. Несмотря на то что модель DOM изначально не предназначена для высокопроизводительной графики, она тем не менее лучше, чем можно было ожидать. Некоторые элементы анимации нам уже встречались в главе 14. На современном компьютере простая игра вроде этой работает достаточно хорошо, даже если мы не особенно беспокоимся об оптимизации.

В следующей главе мы рассмотрим другую браузерную технологию — тег `<canvas>`, который предоставляет более традиционный способ рисования графики, позволяя работать с формами и пикселами, а не с DOM-элементами.

Уровни

Нам понадобится такой способ представления игровых уровней, чтобы их было удобно читать и редактировать. Поскольку все можно начать с сетки, мы можем использовать длинные строки, в которых каждый символ соответствует элементу — это либо часть фоновой сетки, либо движущийся элемент.

Схема небольшого уровня могла бы выглядеть так:

```
let simpleLevelPlan = `
/......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#++++++++++++#..
......###############..
......................`;
```

Здесь точки представляют пустое пространство, решетки (#) — стены, а знаки «плюс» — лаву. Начальная позиция игрока — символ «собака» (@). Каждый символ o представляет собой монету, а знак равенства (=) вверху — блок лавы, который перемещается назад и вперед по горизонтали.

Мы будем поддерживать еще два дополнительных типа движущейся лавы: символ «конвейер» (|) создает вертикально движущиеся капли, а символ v указывает на *капающую* лаву — вертикально движущиеся куски лавы, которые не прыгают вверх-вниз, а только падают вниз, возвращаясь в исходное положение, когда достигнут нижнего края.

Вся игра состоит из нескольких уровней, которые игрок должен пройти. Уровень считается завершенным, когда собраны все монеты. Если игрок коснулся лавы, то текущий уровень возвращается на начальную позицию и игрок может попытаться снова его пройти.

Чтение уровня

В следующем классе хранится объект уровня. Его аргумент должен быть строкой, которая определяет уровень.

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l]);
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];

    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
                let type = levelChars[ch];
                if (typeof type == "string") return type;
                this.startActors.push(
                    type.create(new Vec(x, y), ch));
                return "empty";
              });
        });
    }
}
```

Метод `trim` используется для удаления пробелов в начале и в конце строки схемы. В нашем примере это позволяет начать схему с новой строки, чтобы

все строки располагались непосредственно одна под другой. Полученная строка разделяется на части символами новой строки, а каждая из строк преобразуется в массив, так что в итоге мы получаем массивы символов.

Таким образом, `rows` содержит массив массивов символов — строки схемы. Отсюда мы можем получить ширину и высоту уровня. Но нам все равно нужно отделить движущиеся элементы от фоновой сетки. Мы будем называть движущиеся элементы *акторами*. Акторы будут храниться в массиве объектов. Фон будет представлять собой массив массивов строк, содержащих поля следующих типов: `"empty"` (пусто), `"wall"` (стена) или `"lava"` (лава).

Для того чтобы сформировать эти массивы, мы выполним преобразование строк, а затем — их содержимого. Помните, что функция `map` передает индекс массива вторым аргументом в функцию отображения, которая сообщает координаты *x* и *y* для заданного символа. Позиции в игре будут храниться в виде пар координат, причем координаты верхнего левого угла равны (0, 0), а размер каждого квадрата фона равен 1 в высоту и 1 в ширину.

Для того чтобы интерпретировать эти символы на схеме, конструктор `Level` использует объект `levelChars`, который преобразует фоновые элементы в строки, а символы акторов — в классы. Если `type` является классом актора, то его статический метод `create` применяет для создания объекта, который добавляется в `startActors`, а функция преобразования возвращает значение `"empty"` для данного фонового квадрата.

Положение актора сохраняется в виде объекта `Vec`. Это двумерный вектор, объект со свойствами `x` и `y`, как известно из упражнений к главе 6.

По ходу игры акторы окажутся в разных местах или даже полностью исчезнут (как собранные монеты). Мы будем использовать класс `State` для отслеживания состояния работающей игры.

```
class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }

  static start(level) {
    return new State(level, level.startActors, "playing");
  }
```

```
  get player() {
    return this.actors.find(a => a.type == "player");
  }
}
```

Когда игра закончится, свойство `status` примет значение `"lost"` (проиграл) или `"won"` (выиграл).

Это тоже постоянная структура данных — при обновлении игрового состояния создается новое состояние, а старое остается нетронутым.

Акторы

Объекты-акторы содержат текущую позицию и состояние данного движущегося элемента в нашей игре. Все объекты-акторы имеют одинаковый интерфейс. В их свойстве `pos` хранятся координаты верхнего левого угла элемента, а в свойстве `size` — размер элемента.

У них также есть метод `update`, который используется для вычисления нового состояния и положения после заданного временного шага. Указанный метод имитирует то, что делает актор: если это игрок, то движение в ответ на нажатие клавиш со стрелками, а если лава, то колебание назад и вперед — и возвращает новый, измененный объект актора.

Свойство `type` содержит строку, которая идентифицирует тип актора — `"player"` (игрок), `"coin"` (монета) или `"lava"` (лава). Это полезно при отрисовке игры: вид прямоугольника, нарисованного для актора, определяется его типом.

У классов акторов есть статический метод `create`, который используется конструктором `Level` для создания актора из символа в схеме уровня. Этому методу передаются координаты символа и сам символ, что необходимо, поскольку класс `Lava` обрабатывает несколько разных символов.

Ниже показан класс `Vec`, который мы будем задействовать для хранения двумерных значений, таких как положение и размер акторов.

```
class Vec {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
```

```
  plus(other) {
    return new Vec(this.x + other.x, this.y + other.y);
    }
    times(factor) {
      return new Vec(this.x * factor, this.y * factor);
    }
}
```

Метод `times` масштабирует вектор в заданное число раз. Это бывает полезно, если нужно умножить вектор скорости на временной интервал, чтобы получить расстояние, пройденное за данное время.

Каждый тип акторов имеет собственный класс, так как их поведение сильно различается. Определим эти классы. Их методы `update` мы рассмотрим позже.

У класса игрока есть свойство `speed`, в котором хранится текущая скорость для имитации инерции и силы тяжести.

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }

  static create(pos) {
    return new Player(pos.plus(new Vec(0, -0.5)),
                      new Vec(0, 0));
  }
}

Player.prototype.size = new Vec(0.8, 1.5);
```

Поскольку игрок имеет высоту в полтора квадрата, его начальная позиция устанавливается на полквадрата выше позиции, где должен появиться символ `@`. Тогда его низ будет совпадать с нижним ребром квадрата, внутри которого он появился.

Свойство `size` одинаково для всех экземпляров `Player`, поэтому мы храним его в прототипе, а не в самих экземплярах. Мы могли бы использовать геттер вроде `type`, но тогда при каждом чтении свойства создавался бы и возвращался новый объект `Vec`, что было бы расточительно. (Строки,

будучи неизменяемыми, не нужно заново создавать каждый раз при их вычислении.)

При создании актора `Lava` нам нужно инициализировать объект разными способами в зависимости от символа, на котором он основан. Динамическая лава движется с текущей скоростью, пока не столкнется с препятствием. В этот момент, если у нее есть свойство `reset`, она вернется в исходное положение (капнет). Если же этого не произойдет, то лава изменит скорость и продолжит движение в другом направлении (подпрыгнет).

Метод `create` создает актор лавы в зависимости от символа, переданного конструктором `Level`.

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vec(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vec(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vec(1, 1);
```

Акторы `Coin` относительно просты. По большей части они просто остаются на своих местах. Но, чтобы немного оживить игру, им заданы «колебания» — небольшие вертикальные движения назад-вперед. Чтобы отследить это, в объекте `Coin` хранится исходная позиция, а также свойство `wobble`, которое отслеживает фазу колебательного движения. Вместе они определяют реальное положение монеты (хранящееся в свойстве `pos`).

```
class Coin {
  constructor(pos, basePos, wobble) {
```

```
    this.pos = pos;
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vec(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

Coin.prototype.size = new Vec(0.6, 0.6);
```

В главе 14 было показано, что `Math.sin` позволяет найти y-координату точки, расположенной на окружности. При движении по кругу эта координата движется вперед и назад по плавной волнообразной кривой, что делает функцию синуса полезной для моделирования волнообразного движения.

Чтобы избежать ситуации, когда все монеты движутся вверх и вниз синхронно, начальная фаза каждой монеты выбирается случайным образом. *Фаза* волны `Math.sin` — ширина волны, которую она производит, — составляет 2π. Чтобы присвоить монете случайную начальную позицию на волне, мы умножаем значение, возвращаемое `Math.random`, на это число.

Теперь мы можем определить объект `levelChars`, который преобразует символы схемы в фоновые типы сетки либо в классы акторов.

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

Теперь у нас есть все детали, необходимые для создания экземпляра `Level`.

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9
```

Следующая задача — отображать такие уровни на экране, моделировать время и движение внутри них.

Инкапсуляция как бремя

В коде, представленном в данной главе, по большей части не уделяется внимания инкапсуляции, и это делается по двум причинам. Во-первых, инкапсуляция требует дополнительных усилий. Из-за нее размер программы становится больше, требуются дополнительные концепции и интерфейсы. Поскольку объем кода, который способен вынести читатель, прежде чем его глаза остекленеют, ограничен, я постарался сделать программу как можно короче.

Во-вторых, различные элементы этой игры так тесно связаны друг с другом, что, если бы поведение одного из них изменилось, маловероятно, что какой-либо из остальных элементов можно было бы оставить прежним. В конце концов, многие правила, по которым работает игра, закодированы в ее интерфейсах. Из-за этого они гораздо менее эффективны: стоит изменить одну часть системы, как приходится беспокоиться о том, как это повлияет на другие части, потому что их интерфейсы не будут учитывать новую ситуацию.

Иногда в системе есть *точки разделения*, которые приводят к хорошему разделению системы посредством строгих интерфейсов, но так происходит не всегда. Попытка инкапсулировать нечто, у чего нет четких границ, — верный способ потратить много сил понапрасну. Если вы совершите подобную ошибку, то, скорее всего, вскоре обнаружите, что ваши интерфейсы стали слишком большими и неуклюжими, чересчур детализированными и их приходится часто изменять по мере развития программы.

Но есть вещь, которую мы *обязательно* будем инкапсулировать, — это подсистема рисования. Дело в том, что в следующей главе я покажу, как можно реализовать эту же игру другим способом. Вынеся рисование за рамки интерфейса, мы сможем загрузить ту же игровую программу, подключив другой модуль отображения.

Рисование

Для того чтобы инкапсулировать код, отвечающий за рисование, нужно определить *экранный* объект, отображающий заданный уровень и состояние. Тип отображения, который мы определим в этой главе, называется `DOMDisplay`, потому что для отображения уровня в нем используются DOM-элементы.

Для того чтобы задать цвета и другие фиксированные свойства элементов, из которых состоит игра, мы будем задействовать таблицу стилей. Можно было бы напрямую назначить свойство `style` для элементов при их создании, но это привело бы к созданию более многословных программ.

Следующая вспомогательная функция представляет собой лаконичный способ создания элемента, присвоения ему некоторых атрибутов и добавления дочерних узлов:

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

Отображение создается путем назначения родительского элемента, к которому элемент отображения должен добавить себя и объект уровня.

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

Фоновая сетка уровня, которая никогда не меняется, рисуется один раз. Акторы перерисовываются при каждом обновлении экрана для заданного состояния. Свойство `actorLayer` будет использоваться для отслеживания элемента, который содержит акторы, чтобы их можно было легко удалять и заменять.

Координаты и размеры объектов измеряются в единицах сетки, где единица размера или расстояния равна одному блоку сетки. Задавая размер в пикселах, мы будем вынуждены масштабировать эти координаты, иначе, если размер квадрата будет равен одному пикселу, все в игре станет смехотворно

маленьким. Для этого используется константа `scale`, равная количеству пикселов, которое занимает сторона квадрата на экране.

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
        ...row.map(type => elt("td", {class: type}))));
  ));
}
```

Как уже отмечалось, фон отображается в виде элемента `<table>`. Это хорошо соответствует структуре свойства `rows` для уровня: каждая строка сетки представляется в виде строки таблицы (элемент `<tr>`). Строки сетки используются в качестве имен классов для элементов ячеек таблицы (`<td>`). Оператор распространения (тройная точка) применяется для передачи массивов дочерних узлов в `elt` в качестве отдельных аргументов.

Следующий код CSS делает таблицу похожей на тот фон, который мы хотим получить:

```
.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 этих свойств (`table-layout`, `border-spacing` и `padding`) используются для подавления нежелательного поведения по умолчанию. Мы не хотим, чтобы верстка таблицы зависела от содержимого ее ячеек, и нам не нужны отступы между ячейками таблицы и отступы внутри ячеек.

Правило `background` определяет цвет фона. CSS позволяет указывать цвета в виде слов (`white`) или в таком формате, как `rgb(R, G, B)`, где красная, зеленая и синяя цветовые составляющие представлены в виде трех чисел со значениями от 0 до 255. Таким образом, в `rgb(52, 166, 251)` красная составляющая равна 52, зеленая — 166, а синяя — 251. Поскольку значение синей составляющей является самым большим, то результирующий цвет

будет голубоватым. Как видим, в правиле `.lava` самым большим будет первое число (красная составляющая).

Чтобы нарисовать акторы, мы создадим для каждого из них DOM-элемент и зададим положение и размер для этого элемента в соответствии со свойствами актора. Чтобы перейти от игровых единиц к пикселам, эти значения должны быть умножены на `scale`.

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${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 rect;
  }));
}
```

Чтобы назначить элементу несколько классов, мы разделяем имена классов пробелами. В следующем CSS-коде класс `actor` задает для акторов абсолютное позиционирование. Их имя типа применяется как дополнительный класс, чтобы назначить цвет. Нам не нужно заново определять класс `lava`, мы лишь повторно используем его для квадратов сетки с лавой, которые мы определили ранее.

```
.actor    { position: absolute; }
.coin     { background: rgb(241, 229, 89); }
.player   { background: rgb(64, 64, 64); }
```

Метод `syncState` применяется для отображения состояния на экране. Сначала он удаляет старую графику актора, если таковая имеется, а затем перерисовывает акторы с учетом их новых позиций. Может быть заманчивым попытаться повторно использовать для акторов старые DOM-элементы, но для этого нам бы потребовалось много дополнительной бухгалтерии, чтобы связать акторов с DOM-элементами и обеспечить удаление элементов, когда их акторы исчезают. Поскольку в игре, как правило, участвует очень мало акторов, то их перерисовка не потребует много ресурсов.

```
DOMDisplay.prototype.syncState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
```

```
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

Добавляя текущее состояние уровня в качестве имени класса оболочки, мы можем немного менять стиль актора-игрока в тех случаях, когда игра выиграна или проиграна, добавляя CSS-правило, которое вступает в силу только тогда, когда у игрока есть элемент-предок с заданным классом.

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

Если игрок коснется лавы, то его цвет становится темно-красным, что говорит об ожоге. После того как будет подобрана последняя монета, мы добавим к игроку две размытые белые тени — одну вверху слева и одну вверху справа, — чтобы получился эффект белого ореола.

Мы не можем рассчитывать, что уровень всегда будет помещаться в *окне просмотра* — элементе, в котором мы рисуем игру. Именно поэтому необходим вызов функции `scrollPlayerIntoView`. Он гарантирует, что если уровень выйдет за пределы окна просмотра, то оно прокрутится так, чтобы игрок оказался примерно в центре. Следующий CSS-код обеспечивает максимальный размер наружного DOM-элемента игры и гарантирует, что все выходящее за пределы квадрата элемента не будет видно. Мы также задаем окну просмотра относительное позиционирование, чтобы положение находящихся внутри акторов определялось относительно верхнего левого угла уровня.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

В методе `scrollPlayerIntoView` определяется позиция игрока и изменяется позиция прокрутки наружного элемента. Положение прокрутки также изменяется с помощью свойств `scrollLeft` и `scrollTop` этого элемента, когда игрок находится слишком близко от края.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // Окно просмотра
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
                         .times(scale);

  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
};
```

Способ, которым определяется центр игрока, показывает, как методы нашего типа `Vec` позволяют выполнять вычисления с объектами относительно легко читаемым способом. Для того чтобы найти центр актора, мы складываем его позицию (левый верхний угол) с половиной его размера. Это значение центра в координатах уровня, но нам нужно значение в пиксельных координатах, поэтому затем мы умножаем полученный вектор на наш масштаб отображения.

После этого серия проверок позволяет удостовериться, что позиция игрока не выходит за пределы допустимого диапазона. Обратите внимание, что иногда в результате получаются бессмысленные координаты прокрутки — меньше нуля или выходящие за пределы области прокрутки элемента. Это нормально — DOM ограничит их до приемлемых значений. Если присвоить `scrollLeft` значение `-10`, то в результате его значение станет равным `0`.

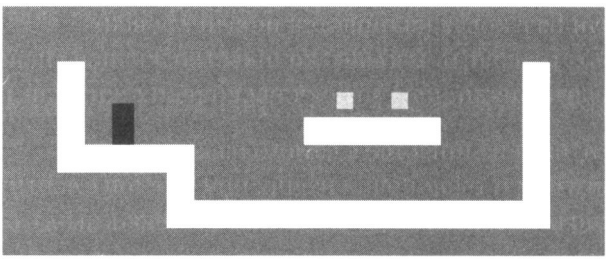
Было бы немного проще всегда пытаться прокручивать окно просмотра так, чтобы игрок оказывался в центре. Но это создает довольно резкий эффект. При прыжках игрока вид будет постоянно смещаться вверх и вниз.

Приятнее, когда в середине экрана есть «нейтральная» область, в которой можно перемещаться, не вызывая прокрутки.

Теперь мы можем отобразить наш мини-уровень.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```



Когда тег `<link>` используется с `rel="stylesheet"`, это является способом загрузки CSS-файла на странице. Файл `game.css` содержит стили, необходимые для нашей игры.

Движение и столкновения

Мы достигли точки, когда можем начать добавлять движение — самый интересный аспект игры. Основной подход, используемый в большинстве подобных игр, состоит в том, чтобы разделить время на маленькие шаги и на каждом шаге перемещать акторы на расстояние, равное их скорости, умноженной на размер временного шага. Мы будем измерять время в секундах, поэтому скорости будут выражаться в единицах в секунду.

Перемещать элементы легко. Трудность заключается в организации взаимодействия между элементами. Когда игрок сталкивается со стеной или полом, он не должен просто проходить сквозь них. Игра должна заметить, что данное движение вызывает попадание объекта в другой объект, и отреагировать соответствующим образом. В случае стен движение должно быть

остановлено. При столкновении с монетой она должна быть подобрана. При прикосновении к лаве игра должна быть проиграна.

В общем случае это большая и сложная задача. Существуют целые библиотеки, обычно называемые *физическими движками*, которые моделируют взаимодействие между физическими объектами в двух или трех измерениях. В этой главе мы воспользуемся более скромным подходом, обрабатывая только столкновения между прямоугольными объектами и весьма упрощенным способом.

Прежде чем перемещать игрока или блок лавы, мы проверим, будет ли движение проходить внутри стены. Если это так, то мы просто полностью отменим движение. Реакция на такое столкновение зависит от типа актора — игрок остановится, а блок лавы отскочит назад.

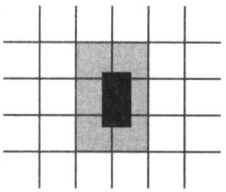
Такой подход требует, чтобы наши временные шаги были достаточно маленькими, чтобы остановить движение до того, как объекты соприкоснутся. Если временные шаги (и, следовательно, шаги движения) будут слишком большими, то игрок в итоге замрет на заметном расстоянии над землей. Другой подход, возможно, лучше, но сложнее: найти точное место столкновения и двигаться до него. Мы выберем простой подход и скроем его проблемы, выбирая маленький шаг анимации.

Следующий метод сообщает нам, касается ли прямоугольник (заданный позицией и размером) элемента сетки данного типа.

```
Level.prototype.touches = function(pos, size, type) {
  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);

  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
                      y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

Данный метод вычисляет множество квадратов сетки, с которыми тело соприкасается, используя для этого его координаты и методы `Math.floor` и `Math.ceil`. Помните, что квадраты сетки имеют размер 1 на 1. Округляя стороны прямоугольника вверх и вниз, мы получим диапазон фоновых квадратов, которых касается заданный элемент.



Перебирая блок квадратов сетки, найденных путем округления координат, мы возвращаем `true`, когда найден подходящий квадрат. Квадраты, расположенные за пределами уровня, всегда рассматриваются как стена (`"wall"`), чтобы гарантировать, что игрок не может покинуть мир игры и что мы случайно не попытаемся прочитать данные за пределами нашего массива строк.

Метод обновления состояния `update` использует метод `touches`, чтобы выяснить, касается ли игрок лавы.

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);

  if (newState.status != "playing") return newState;

  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }

  for (let actor of actors) {
    if (actor != player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
  return newState;
};
```

Методу передается временной шаг и структура данных, содержащая данные о нажатых клавишах. Вначале метод вызывает `update` для всех акторов, создавая массив обновленных акторов. Акторы также получают временной шаг, данные о нажатых клавишах и состояние, чтобы получить обновления на их основе. Читать данные о клавишах будет только игрок, так как это единственный актор, который управляется клавиатурой.

Если игра уже закончена, то дальнейшая обработка не требуется (игра не может быть выиграна после проигрыша и не может быть проиграна после выигрыша). В противном случае метод проверяет, касается ли игрок фоновой лавы. Если касается, то игра окончена — мы проиграли. Наконец, если игра все еще продолжается, то метод проверяет, перекрывают ли игрока другие акторы.

Перекрытие акторов друг другом обнаруживается с помощью функции `overlap`. Она принимает два объекта-актора и возвращает `true`, если они соприкасаются, что имеет место, когда они перекрываются как по оси *X*, так и по оси *Y*.

```
function overlap(actor1, actor2) {
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&
         actor1.pos.x < actor2.pos.x + actor2.size.x &&
         actor1.pos.y + actor1.size.y > actor2.pos.y &&
         actor1.pos.y < actor2.pos.y + actor2.size.y;
}
```

Если какой-либо из акторов перекрывается другим актором, то его метод `collide` получает шанс обновить состояние. Прикосновение к актору-лаве меняет состояние игры на `"lost"` (проигрыш). При касании монеты она исчезает, а если это была последняя монета уровня, то состояние игры меняется на `"won"` (выигрыш).

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

Изменение акторов

Методы `update` объектов-акторов принимают в качестве аргументов временной шаг, объект состояния и объект `keys`. Для актора типа `Lava` этот метод игнорирует объект `keys`.

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

Этот метод `update` вычисляет новую позицию, прибавляя к старой позиции произведение временного шага на текущую скорость. Если эта новая позиция не заблокирована каким-либо препятствием, то объект перемещается туда. Если есть препятствие, то поведение объекта зависит от типа блока лавы: у капающей лавы есть позиция `reset`, в которую она отскакивает, когда наталкивается на что-то. У прыгающей лавы инвертируется скорость путем умножения ее значения на –1, чтобы начать двигаться в противоположном направлении.

У монет есть собственный метод `update`, позволяющий им колебаться. Монеты игнорируют столкновения с сеткой, так как они только колеблются внутри своего квадрата.

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vec(0, wobblePos)),
                  this.basePos, wobble);
};
```

Свойство `wobble` увеличивается, чтобы отслеживать время, а затем используется в качестве аргумента для `Math.sin`, чтобы получить новое положение на волне. Затем текущее положение монеты вычисляется исходя из ее исходного положения и смещения на основе этой волны.

Остается сам игрок. Движение игрока обрабатывается отдельно по каждой оси, потому что соприкосновение с полом не должно препятствовать движению по горизонтали, а удар в стену не должен останавливать падение или прыжок.

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vec(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vec(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall")) {
    pos = movedY;
  } else if (keys.ArrowUp && ySpeed > 0) {
    ySpeed = -jumpSpeed;
  } else {
    ySpeed = 0;
  }
  return new Player(pos, new Vec(xSpeed, ySpeed));
};
```

Движение по горизонтали вычисляется на основании состояния клавиш со стрелками влево и вправо. Если нет стены, блокирующей новую позицию, созданную этим движением, используется данная позиция, иначе — сохраняется старая.

Движение по вертикали выполняется аналогичным образом, но должно также имитировать прыжки и гравитацию. Вертикальная скорость игрока (`ySpeed`) сначала увеличивается для учета силы тяжести.

И снова мы проверяем стены. Если мы не касаемся ни одной из них, то используем новую позицию. Если же нам *встретилась* стена, то есть два варианта. Когда нажата стрелка вверх *и* мы движемся вниз (то есть предмет,

с которым мы столкнулись, находится ниже нас), скорость меняется на относительно большое отрицательное значение, что заставляет игрока прыгнуть. Если же это не так, то игрок просто столкнулся с чем-то и скорости присваивается нулевое значение.

Сила гравитации, скорость прыжка и почти все остальные константы в этой игре устанавливались методом проб и ошибок. Я проверял разные значения, пока не нашел понравившуюся комбинацию.

Отслеживание нажатий клавиш

Для таких игр, как эта, желательно, чтобы программа реагировала не на каждое нажатие клавиши, а чтобы эффект (перемещение фигуры игрока) действовал все время, пока клавиша удерживается.

Нам нужно создать обработчик событий клавиш, который бы хранил текущее состояние клавиш со стрелками влево, вправо и вверх. Нам также потребуется вызывать для этих клавиш `warnDefault`, чтобы избежать прокручивания страницы.

Следующая функция получает массив имен клавиш и возвращает объект, который отслеживает текущее состояние этих клавиш. Функция регистрирует обработчики для событий `"keydown"` и `"keyup"` и изменяет объект, когда в событии в наборе отслеживаемых кодов присутствует код заданной клавиши.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

Одна и та же функция обработчика используется для обоих типов событий. Она просматривает свойство `type` объекта события и определяет, следует ли изменить состояние клавиши на `true` (`"keydown"`) или `false` (`"keyup"`).

Игра в действии

Функция `requestAnimationFrame`, которую мы уже видели в главе 14, — хороший способ добавить анимацию в игру. Но интерфейс этой функции весьма примитивен: чтобы его использовать, нужно отслеживать время, когда наша функция была вызвана в последний раз, и вызвать `requestAnimationFrame` после каждого кадра.

Определим вспомогательную функцию, которая будет оборачивать эти скучные части работы в удобный интерфейс и позволит нам просто вызывать `runAnimation` — функцию, принимающую в качестве аргумента разницу во времени и рисующую один кадр. Когда функция `frame` возвращает значение `false`, анимация останавливается.

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

Я установил максимальный интервал между кадрами 100 миллисекунд (одна десятая секунды). Когда вкладка или окно браузера с нашей страницей закрыты, вызовы `requestAnimationFrame` приостанавливаются до тех пор, пока вкладка или окно не отобразятся снова. В этом случае разница между `lastTime` и `time` будет равна времени, в течение которого страница была закрыта. Вносить изменения в игру за столь значительный интервал в рамках одного шага было бы глупо и могло бы вызвать странные побочные эффекты, такие как падение игрока сквозь пол.

Функция также преобразует временные шаги в секунды, которые проще себе представить, чем миллисекунды.

Функция `runLevel` принимает объект `Level` и конструктор отображения и возвращает промис. Она отображает уровень (в `document.body`) и позволяет пользователю играть на нем. Когда уровень закончен (проигран или выигран), `runLevel` ждет еще одну секунду (чтобы пользователь успел увидеть, что произошло) и очищает экран, останавливает анимацию и разрешает промис до конечного состояния игры.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

Игра представляет собой последовательность уровней. Всякий раз, когда игрок умирает, текущий уровень перезапускается. Когда уровень пройден, игрок переходит на следующий уровень. Это можно выразить в виде следующей функции, которая принимает массив схем уровней (в виде строк) и конструктор отображения:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
```

```
  }
  console.log("Ты победил!");
}
```

Поскольку мы заставили `runLevel` возвращать промис, функцию `runGame` можно написать с использованием асинхронной функции, как показано в главе 11. Она возвращает другой промис, который разрешается, когда игрок заканчивает игру.

В привязке `GAME_LEVELS` в «песочнице» этой главы (https://eloquentjavascript.net/code#16) есть готовый набор планов уровней. Данная страница передает их в `runGame`, запуская игру.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Упражнения

Игра окончена

В платформенных играх игрок по традиции начинает игру с ограниченным количеством *жизней* и теряет одну из них при каждой смерти. Когда жизни заканчиваются, игра начинается с самого начала.

Добавьте в `runGame` наличие у игрока нескольких жизней. Пускай игрок начинает с трех. Выводите текущее количество жизней (используя `console.log`) каждый раз, когда начинается новый уровень.

Приостановка игры

Сделайте возможным приостановить (поставить на паузу) или отменить игру, нажав клавишу Esc.

Для этого можно изменить функцию `runLevel`, использовав другой обработчик событий клавиатуры и прерывая либо возобновляя анимацию при каждом нажатии клавиши Esc.

На первый взгляд может показаться, что интерфейс `runAnimation` для этого не предназначен, но вы решите проблему, изменив способ, которым `runLevel` вызывает данный интерфейс.

Когда это заработает, можно попробовать кое-что еще. Наш способ регистрации обработчиков событий клавиатуры несколько проблематичен. Сейчас объект `arrowKeys` является глобальной привязкой, и его обработчики событий сохраняются даже тогда, когда игра не запущена. Расширьте `trackKeys` так, чтобы появился способ отменить регистрацию обработчиков, а затем измените `runLevel`, чтобы регистрировать обработчики при запуске и отменять их регистрацию после завершения программы.

Монстр

В платформенных играх обычно есть враги — чтобы их победить, на них нужно запрыгнуть. В этом упражнении я предлагаю вам добавить в игру актор такого типа.

Мы назовем его монстром. Монстры передвигаются только по горизонтали. Вы можете заставить их двигаться в направлении игрока, прыгать назад и вперед, как горизонтальная лава, или иметь любую другую схему движения. Класс не должен обрабатывать падения, но должен следить за тем, чтобы монстр не проходил сквозь стены.

Когда монстр касается игрока, результат зависит от того, запрыгнул игрок сверху на монстра или нет. Это можно описать, проверяя, находится ли основание игрока около вершины монстра. Если так, то монстр исчезает, если нет — игра проиграна.

17 Рисование на холсте

> Рисование — это обман.
>
> *М. К. Эшер. Магическое зеркало*

Браузеры предоставляют нам несколько способов отображения графики. Самый простой способ — использовать стили для позиционирования и назначения цвета обычных элементов DOM. Это позволяет сделать довольно много, как показала игра, описанная в предыдущей главе. Добавляя к узлам частично прозрачные фоновые изображения, можно заставить их выглядеть именно так, как мы хотим. Можно даже вращать или наклонять узлы с помощью стиля `transform`.

Но мы будем использовать DOM-модель не только для того, для чего она была предназначена изначально. Некоторые задачи, такие как рисование линии между произвольными точками, крайне неудобно выполнять, задействуя обычные элементы HTML.

Этому есть две альтернативы. Первая основана на DOM, но вместо HTML использует *масштабируемую векторную графику* (Scalable Vector Graphics, SVG). SVG можно представить себе как диалект языка разметки документов, в котором основное внимание уделяется не тексту, а формам. SVG-документ можно встроить в HTML-документ непосредственно или вставить в тег `<img>`.

Второй вариант называется *холстом*. Холст — это отдельный элемент DOM, в котором инкапсулируется изображение. Холст предоставляет программный интерфейс для рисования фигур в области, занимаемой узлом. Основное различие между холстом и SVG-рисунком заключается в том, что в SVG сохраняется первоначальное описание фигур, так что их можно перемещать или изменять в любое время. Холст, напротив, преобразует фигуры в пикселы (цветные точки растра) сразу после того, как они будут нарисованы,

и не запоминает, что представляют собой эти пикселы. Единственный способ переместить фигуру на холсте — очистить холст (или часть холста, на которой размещена фигура) и перерисовать его на новой позиции.

SVG

В этой книге я не буду углубляться в детали SVG, а только кратко объясню, как это работает. В конце главы я вернусь к компромиссам, которые приходится учитывать при принятии решения о том, какой механизм рисования лучше подходит для данного приложения.

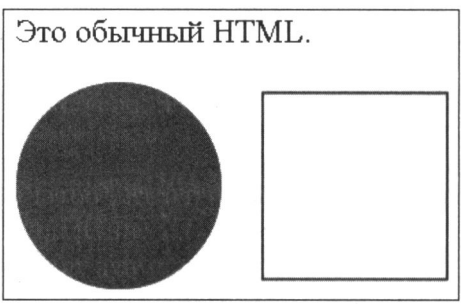
Вот HTML-документ с простой SVG-картинкой:

```
<p>Это обычный HTML.</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 элемент (и его потомков) на другое *пространство имен XML*. Это пространство имен, идентифицируемое по URL, определяет диалект языка, о каком мы сейчас говорим. Теги `<circle>` и `<rect>`, которых не существует в HTML, есть в SVG, где они позволяют рисовать фигуры, используя стиль и положение, определенные их атрибутами.

Документ отображается так.



Подобно тегам HTML, эти теги создают DOM-элементы, с которыми могут взаимодействовать сценарии JavaScript. Например, следующий сценарий окрашивает элемент `<circle>` в голубой цвет:

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

Элемент canvas

Графика холста отрисовывается в элементе `<canvas>`. Ему можно присвоить такие атрибуты, как `width` и `height`, определяющие его ширину и высоту в пикселах.

Только что созданный холст пуст — это означает, что он полностью прозрачен и, следовательно, отображается в документе как пустое место.

Тег `<canvas>` предназначен для реализации различных стилей рисования. Чтобы получить доступ к реальному интерфейсу рисования, сначала нужно создать *контекст* — объект, методы которого реализуют интерфейс рисования. В настоящее время существует два широко поддерживаемых стиля рисования: `"2d"` для двумерной графики и `"webgl"` для трехмерной графики через интерфейс OpenGL.

В этой книге мы не станем обсуждать WebGL, так как будем придерживаться двух измерений. Но если вас интересует трехмерная графика, советую обратить внимание на WebGL. Он обеспечивает прямой интерфейс с графическим оборудованием и позволяет эффективно воспроизводить даже сложные сцены с использованием JavaScript.

Для создания контекста в DOM-элементе `<canvas>` используется метод `getContext`.

```
<p>Абзац перед холстом.</p>
<canvas width="120" height="60"></canvas>
<p>Абзац после холста.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

После того как создан объект контекста, в примере рисуется красный прямоугольник шириной 100 пикселов и высотой 50 пикселов с верхним левым углом в точке (10, 10).

Подобно HTML (и SVG), начальная точка (0, 0) системы координат, которая используется для холста, находится в верхнем левом углу, а ось *Y* направлена из этой точки вниз. Таким образом, точка (10, 10) расположена на 10 пикселов ниже и правее верхнего левого угла.

Линии и поверхности

В интерфейсе холста форма может быть *заполнена* — это означает, что ее области присвоен определенный цвет или узор, — или снабжена *контуром*, то есть вдоль ее края рисуется линия. Такая же терминология используется и в SVG.

Метод `fillRect` заполняет прямоугольник. Сначала он принимает *x*- и *y*-координаты верхнего левого угла прямоугольника, затем его ширину и высоту. Аналогичный метод `strokeRect` рисует контур прямоугольника.

Ни один из методов не принимает никаких дополнительных параметров. Цвет заливки, толщина контура и т. д. определяются не аргументом метода (как было бы разумно ожидать), а свойствами объекта контекста.

Свойство `fillStyle` определяет способ заполнения фигур. Ему может быть присвоена строка, которая определяет цвет посредством цветовой нотации CSS.

Свойство `strokeStyle` работает аналогично, но определяет цвет контура. Ширина контура определяется свойством `lineWidth` и может содержать любое положительное число.

```
<canvas></canvas>
<script>
  let 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>
```

Этот код рисует два синих квадрата, используя для второго более толстую линию контура.

Если, как в данном примере, не указан атрибут `width` или `height`, то для элемента холста по умолчанию устанавливается ширина 300 пикселов и высота 150 пикселов.

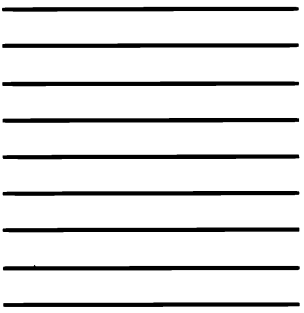
Пути

Путь — это последовательность линий. В двумерном интерфейсе холста используется особый подход к описанию такого пути. Это делается исключительно за счет побочных эффектов. Пути не являются значениями, которые могут быть сохранены и переданы. Вместо этого, чтобы что-то сделать с путем, нужно последовательно вызвать ряд методов, описывающих его форму.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

В этом примере создается путь, состоящий из нескольких горизонтальных отрезков, а затем им присваивается контур с помощью метода `stroke`. Каждый сегмент, созданный с помощью `lineTo`, начинается с текущей *позиции* пути. Эта позиция обычно является конечной точкой последнего сегмента, если только не был вызван метод `moveTo`. В данном случае следующий сегмент будет начинаться с позиции, переданной в `moveTo`.

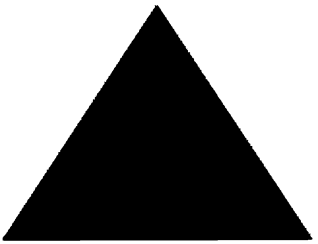
Путь, описанный в предыдущей программе, выглядит так:



При заполнении пути (с помощью метода `fill`) каждая фигура заполняется отдельно. Путь может содержать несколько фигур — каждое передвижение `moveTo` начинает новую фигуру. Но для того, чтобы путь можно было заполнить, он должен быть *замкнутым* (то есть его начало и конец находятся в одной точке). Если путь еще не замкнут, то к нему добавляется линия, соединяющая его конечную и начальную точки, после чего замкнутая форма, полученная в результате завершенного пути, заполняется.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

В этом примере рисуется заполненный треугольник. Обратите внимание, что только две его стороны нарисованы явно. Третья сторона, от правого нижнего угла до вершины, лишь подразумевается, и при отрисовке контура пути ее нет.



Можно также использовать метод `closePath` для явного замыкания пути. Этот метод добавляет реальный отрезок линии между конечной и начальной точками пути. Указанный сегмент рисуется при создании контура пути.

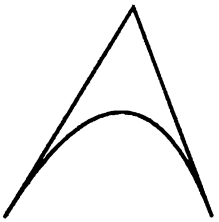
Кривые

Путь может также содержать кривые линии. К сожалению, их рисовать немного сложнее.

Метод `quadraticCurveTo` рисует кривую, которая заканчивается в заданной точке. Чтобы определить кривизну линии, этому методу передается контрольная точка, а также конечная точка. Данную контрольную точку можно представить как точку, которая *притягивает* к себе линию, определяя ее кривизну. Линия не проходит через контрольную точку, но ее направление в начальной и конечной точках будет таким, что прямая линия, проведенная в данном направлении, проходила бы через контрольную точку. Это показано в следующем примере:

```
<canvas></canvas>
<script>
  let 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>
```

Здесь создается путь, который выглядит следующим образом.

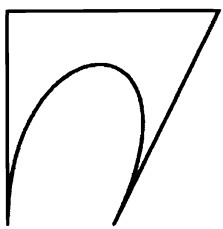


Мы рисуем квадратичную кривую слева направо, с контрольной точкой с координатами (60, 10), а затем два отрезка, проходящих через эту контрольную точку и возвращающихся в начало линии. Результат несколько напоминает эмблему «Звездного пути». Здесь видно, на что влияет контрольная точка: линии, выходящие из нижних вершин, начинаются в направлении нее, а затем отклоняются к конечной точке.

Метод `bezierCurveTo` рисует аналогичную кривую, но вместо одной контрольной точки здесь их две: по одной для каждой конечной точки линии. Вот аналогичный пример, иллюстрирующий поведение такой кривой:

```
<canvas></canvas>
<script>
  let 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 определяют направление на обоих концах кривой. Чем дальше они находятся от соответствующей точки, тем сильнее кривая будет «выпирать» в этом направлении.



С такими кривыми иногда сложно работать — не всегда понятно, как разместить контрольные точки, чтобы получить желаемую форму. Иногда их можно вычислить, а иногда приходится искать подходящее значение методом проб и ошибок.

Метод `arc` — это способ нарисовать линию, которая является частью окружности. Данный метод принимает пару координат, определяющих центр дуги, радиус, а также начальный и конечный углы.

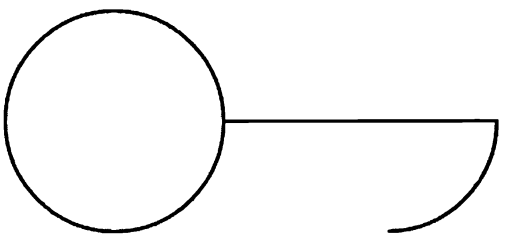
Эти два последних параметра позволяют нарисовать только часть окружности. Углы измеряются не в градусах, а в радианах. Это означает, что для полного круга угол равен 2π или `2*Math.PI`, что составляет около 6,28. Отсчет углов начинается из крайней правой точки от центра окружности и идет по часовой стрелке.

Чтобы нарисовать полный круг, можно задать начальную точку 0 и конечную больше 2π (например, 7).

```
<canvas></canvas>
<script>
  let 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 π 1/2
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

На картинке мы получим линию справа от полного круга (первый вызов `arc`) и еще четверть круга (второй вызов). Как и другие методы рисования пути, линия, нарисованная с помощью метода `arc`, соединяется с предыдущим сегментом пути.

Чтобы этого не происходило, можно вызвать `moveTo` или начать новый путь.



Рисование круговой диаграммы

Представьте себе, что вы только что устроились на работу в какую-нибудь EconomiCorp Inc. и ваше первое задание — нарисовать круговую диаграмму результатов опроса удовлетворенности клиентов.

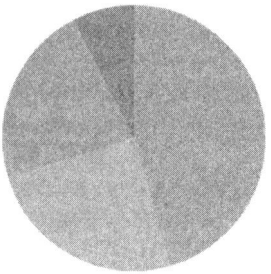
Привязка `results` содержит массив объектов, представляющих ответы на опрос.

```
const results = [
  {name: "Удовлетворен", count: 1043, color: "lightblue"},
  {name: "Нормально", count: 563, color: "lightgreen"},
  {name: "Не удовлетворен", count: 510, color: "pink"},
  {name: "Без комментариев", count: 175, color: "silver"}
];
```

Для того чтобы создать круговую диаграмму, мы нарисуем несколько сегментов круга, каждый из которых будет состоять из дуги и пары линий до центра этой дуги. Чтобы вычислить угол, занятый каждой дугой, нам нужно разделить полный круг (2π) на общее количество ответов, а затем умножить это число (размер угла для одного ответа) на количество людей, которые выбрали данный ответ.

```
<canvas width="200" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  // Начинаем сверху
  let currentAngle = -0.5 * Math.PI;
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
   // center=100,100, radius=100
   // с текущего угла, по часовой стрелке, по размеру каждого сегмента
   cx.arc(100, 100, 100,
          currentAngle, currentAngle + sliceAngle);
   currentAngle += sliceAngle;
   cx.lineTo(100, 100);
   cx.fillStyle = result.color;
   cx.fill();
  }
</script>
```

В результате получится такая диаграмма.



Но диаграмма, по которой непонятно, что означают эти сегменты, не особенно полезна. Нам нужен способ вывода текста на холсте.

Текст

Контекст рисования на 2D-холсте предоставляет методы `fillText` и `strokeText`. Последний может быть полезен для обводки букв контуром, но обычно нам нужен `fillText`. Этот метод заполняет контур текста текущим `fillStyle`.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>
```

Размер, стиль и шрифт текста можно указать с помощью свойства `font`. В этом примере заданы только размер и гарнитура шрифта. Кроме того, в начале строки можно указать стиль — курсив (`italic`) или полужирный (`bold`).

Последние два аргумента `fillText` и `strokeText` определяют позицию, в которой рисуется текст. По умолчанию это начало алфавитной базовой линии текста, то есть линии, на которой «стоят» буквы, не считая «хвостиков» в таких буквах, как *б* или *р*. Чтобы изменить позицию по горизонтали, нужно присвоить свойству `textAlign` значение `"end"` или `"center"`, а чтобы изменить позицию по вертикали — присвоить `textBaseline` значение `"top"`, `"middle"` или `"bottom"`.

Мы еще вернемся к нашей круговой диаграмме и проблеме обозначения сегментов в упражнениях в конце этой главы.

Изображения

В компьютерной графике часто проводится различие между *векторной* и *растровой* графикой. Векторная графика — это то, что мы до сих пор делали в данной главе, определяя картинку в виде логического описания фигур. Растровая графика, напротив, не описывает реальные фигуры, а работает с пиксельными данными (растрами цветных точек).

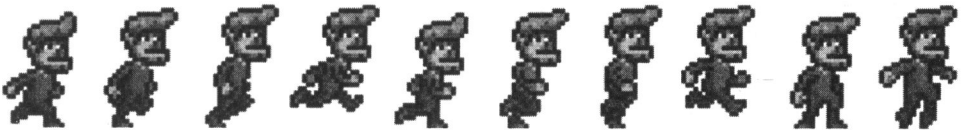
Метод `drawImage` позволяет рисовать пиксельные данные на холсте. Они могут быть получены из элемента `<img>` или из другого холста. В следующем примере создается не связанный с деревом DOM элемент `<img>`, в который загружается файл изображения. Но мы не можем сразу начать рисовать данные с этой картинки, потому что браузер, возможно, ее еще не загрузил. Чтобы справиться с этой проблемой, нужно зарегистрировать обработчик события `load` и выполнить рисование после загрузки изображения.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>
```

По умолчанию `drawImage` нарисует изображение оригинального размера. Но можно задать другие ширину и высоту, передав методу два дополнительных аргумента.

Если передать `drawImage` *девять* аргументов, то этот метод можно использовать для рисования отдельного фрагмента изображения. Аргументы со второго по пятый определяют прямоугольник (координаты x и y, ширина и высота) исходного изображения, содержимое которого должно быть скопировано, а аргументы с шестого по девятый определяют прямоугольную область (на холсте), в которую это содержимое должно быть скопировано.

Это можно использовать для упаковки нескольких *спрайтов* (элементов изображения) в один файл изображения и затем отрисовки только той части, которая нам нужна. Предположим, что у нас есть эта картинка, содержащая разные положения игрового персонажа.



Чередуя позиции, которые выводятся на экран, можно изобразить анимацию, и будет казаться, что персонаж идет.

Для анимации изображения на холсте полезен метод `clearRect`. Он похож на `fillRect`, но вместо того, чтобы окрашивать прямоугольник цветом, он делает его прозрачным, удаляя ранее нарисованные пикселы.

Как нам известно, каждый *спрайт* (каждый подкадр) имеет ширину 24 пиксела и высоту 30 пикселов. Приведенный далее код загружает изображение, а затем устанавливает интервал (повторяющийся таймер) для рисования следующего кадра:

```
<canvas></canvas>
/<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // исходный прямоугольник
                   cycle * spriteW, 0, spriteW, spriteH,
                   // прямоугольная область,
                   // куда нужно вписать изображение
                   0,               0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

Привязка `cycle` отслеживает текущую позицию в анимации. Для каждого кадра эта позиция увеличивается до 7, а затем снова обнуляется с помощью оператора остатка. Эта привязка используется для вычисления x-координаты спрайта текущей позиции персонажа на рисунке.

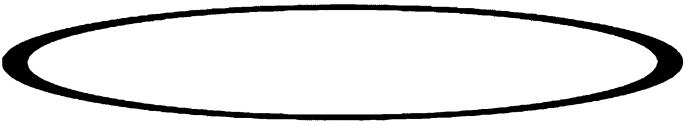
Преобразования

А что, если мы захотим, чтобы наш персонаж шел не направо, а налево? Конечно, мы могли бы нарисовать другой набор спрайтов. Но мы также можем поручить холсту нарисовать картинку наоборот.

Метод `scale` сделает так, что все нарисованное после вызова этого метода будет масштабировано. Данный метод принимает два параметра: один для установки масштаба по горизонтали и второй — по вертикали.

```
<canvas></canvas>
<script>
  let 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>
```

Из-за вызова `scale` круг получается в три раза шире и в полтора раза выше, чем исходное изображение.



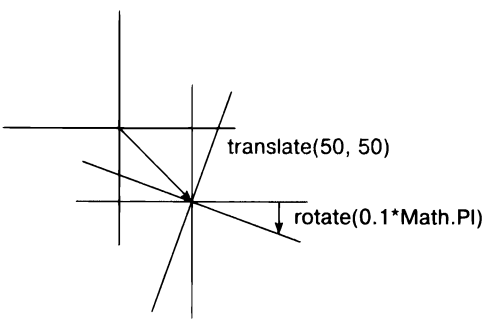
Масштабирование приводит к тому, что все изображение, включая ширину линии, будет растягиваться или сжиматься в соответствии с указанными параметрами. При масштабировании на отрицательную величину картинка перевернется. Переворот происходит вокруг точки (0, 0), а следовательно, также будет изменено и направление системы координат. При масштабировании по горизонтали с коэффициентом −1 фигура, нарисованная в позиции 100 по координате x, окажется в позиции, которая раньше соответствовала координате −100.

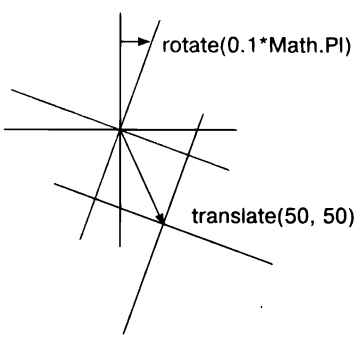
Таким образом, чтобы перевернуть картинку, мы не можем просто добавить `cx.scale(-1, 1)` перед вызовом `drawImage`, потому что тогда картинка переместится за пределы холста, где она не будет видна. Чтобы это компенсировать, мы могли бы изменить координаты, заданные для `drawImage`, рисуя изображение в позиции с x-координатой, равной −50 вместо 0. Но есть и другое

решение, не требующее, чтобы код, выполняющий рисование, учитывал изменение масштаба, — это правильно выбрать ось, относительно которой происходит масштабирование.

Существует еще несколько методов, помимо `scale`, влияющих на систему координат холста. С помощью метода `rotate` можно вращать нарисованные после вызова этого метода фигуры, а с помощью метода `translate` — перемещать их. Интересный — и вносящий путаницу — момент заключается в том, что эти преобразования *накладываются* друг на друга, так что каждое из них происходит относительно предыдущих преобразований.

Таким образом, если дважды сдвинуть изображение на 10 пикселов по горизонтали, то в итоге оно будет сдвинуто вправо на 20 пикселов. Если сначала переместить центр системы координат в точку (50, 50), а затем повернуть изображение на 20 градусов (примерно 0,1π радиан), то вращение произойдет вокруг точки (50, 50).





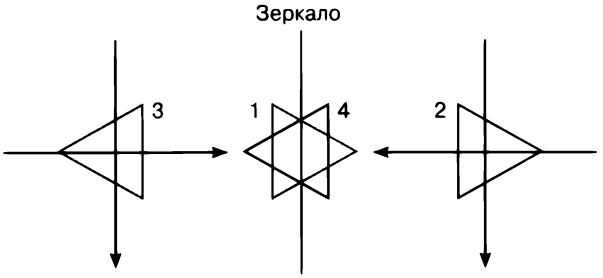
Но если *сначала* повернуть изображение на 20 градусов, а *затем* переместить в точку (50, 50), то перемещение произойдет в повернутой системе координат и, таким образом, получится другое положение картинки. Последовательность, в которой применяются преобразования, имеет значение.

Чтобы перевернуть изображение вокруг вертикальной оси в заданной позиции x, можно сделать следующее:

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

Мы перемещаем ось *Y* туда, где хотим разместить наше «зеркало», применяем зеркальное отражение и, наконец, возвращаем ось *Y* на исходное место в новой «зеркально отображенной» вселенной.

На следующем рисунке показано, как это работает:



Здесь показаны системы координат до и после зеркального отражения относительно центральной оси. Треугольники пронумерованы, чтобы проиллюстрировать каждый шаг. Если нарисовать треугольник с положительной *x*-координатой, то по умолчанию он будет размещаться в том месте, где находится треугольник 1. При вызове `flipHorizontally` сначала выполняется перемещение вправо, так что получается треугольник 2. Затем выполняется масштабирование и треугольник переворачивается в положение 3. Но это не то положение, где он должен был бы оказаться в случае отражения относительно заданной линии. Повторный вызов `translate` исправляет это — первоначальное перемещение «отменяется», так что треугольник 4 появляется именно там, где и должен.

Теперь мы можем нарисовать зеркальный персонаж в позиции (100, 0), переворачивая мир относительно его вертикальной оси.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

Сохранение и отмена преобразований

Преобразования сохраняются. Все остальное, что было нарисовано после зеркально повернутого персонажа, также будет зеркально повернуто. Иногда это бывает неудобно.

Можно сохранить текущее преобразование, нарисовать что-то, выполнить другие преобразования, а затем восстановить старое преобразование. Обычно это правильное решение для функции, которая должна временно преобразовать систему координат. Сначала мы сохраняем любое преобразование, использованное в коде, вызвавшем функцию. Затем функция делает свое дело, добавляя другие преобразования поверх текущего. И в конце мы возвращаемся к тому преобразованию, с которого начали.

Такое управление преобразованиями в контексте 2D-холста выполняется с помощью методов `save` и `restore`. Концептуально эти методы сохраняют стек состояний преобразования. При вызове `save` текущее состояние помещается в стек, а при вызове `restore` извлекается состояние из верхней части стека и используется в качестве текущего преобразования контекста. Для полного сброса преобразования можно также вызвать метод `resetTransform`.

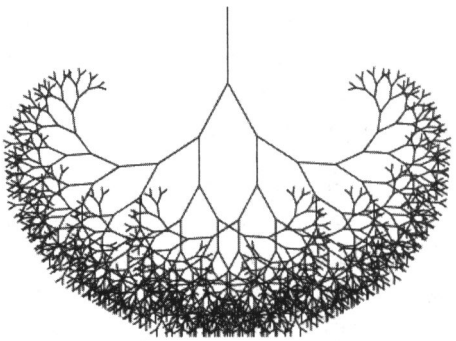
Показанная в следующем примере функция `branch` демонстрирует, что можно сделать с помощью функции, которая изменяет преобразование и затем вызывает функцию (в данном случае сама себя), продолжающую рисование с заданным преобразованием.

Эта функция выводит древовидную фигуру, рисуя линию, затем перемещая центр системы координат в конечную точку данной линии и вызывая себя два раза: сначала с поворотом влево, а затем с поворотом вправо. При каждом вызове длина нарисованной ветви уменьшается. Рекурсия останавливается, когда длина ветви становится меньше 8.

```
<canvas width="600" height="300"></canvas>
<script>
  let 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>
```

В результате получается простой фрактал.



Без вызовов `save` и `restore` второй рекурсивный вызов ветви закончился бы с положением и поворотом, созданными первым вызовом. Он был бы связан не с текущей ветвью, а с самой внутренней, самой правой ветвью, нарисованной первым вызовом. Получилась бы тоже интересная фигура, но это определенно не было бы дерево.

Возвращаясь к игре

Теперь мы знаем о рисовании на холсте достаточно, чтобы начать работу над системой отображения на основе холста для игры, описанной в предыдущей главе. Новая картинка больше не будет отображать одни лишь цветные прямоугольники. Вместо этого мы с помощью `drawImage` будем рисовать картинки, представляющие элементы игры.

Мы определим еще один тип экранного объекта под названием `CanvasDisplay`, поддерживающий тот же интерфейс, что и `DOMDisplay` из главы 16, а именно методы `syncState` и `clear`.

Этот объект содержит немного больше информации, чем `DOMDisplay`. Вместо того чтобы использовать позицию прокрутки для своего DOM-элемента,

он отслеживает собственное окно просмотра, которое сообщает нам, какую часть уровня мы в данный момент видим. Наконец, у этого объекта есть свойство `flipPlayer`, так что, даже когда игрок стоит на месте, он продолжает смотреть в том направлении, в котором он двигался в прошлый раз.

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

Метод `syncState` сначала вычисляет новое окно просмотра, а затем рисует игровую сцену в соответствующей позиции.

```
CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};
```

В отличие от `DOMDisplay` этот стиль отображения *действительно* перерисовывает фон при каждом обновлении. Поскольку фигуры на холсте — это просто пикселы, то после того, как мы их нарисовали, не существует удобного способа их переместить (или удалить). Единственный способ обновить изображение на холсте — это очистить его и перерисовать сцену заново. Нам также нужно иметь возможность прокрутки, что требует изменения положения фона.

Метод `updateViewport` аналогичен методу `scrollPlayerIntoView` в `DOMDisplay`. Он проверяет, находится ли игрок слишком близко к краю экрана, и, если это так, перемещает окно просмотра.

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let 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,
                         state.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,
                        state.level.height - view.height);
  }
};
```

Вызовы `Math.max` и `Math.min` гарантируют, что окно просмотра никогда не будет отображать пространство вне уровня. `Math.max(x, 0)` гарантирует, что полученное число не будет меньше нуля, а `Math.min` — что значение всегда будет меньше заданной границы.

При очистке дисплея мы будем использовать немного другой цвет, в зависимости от того, выиграна игра (ярче) или проиграна (темнее).

```
CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
      this.cx.fillStyle = "rgb(68, 191, 255)";
    } else if (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);
};
```

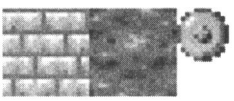
Чтобы нарисовать фон, мы перебираем все плитки, которые видны в текущем окне просмотра, используя тот же прием, что и в методе `touches` из предыдущей главы.

```
let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? Scale : 0;
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
                        screenX, screenY, scale, scale);
    }
  }
};
```

Непустые плитки отрисовываются с помощью `drawImage`. Изображение `otherSprites` содержит картинки, используемые для всех элементов, кроме игрока. В этом изображении содержатся, слева направо: плитка для стены, плитка для лавы и спрайт для монеты.



Фоновые плитки имеют размер 20 × 20 пикселов, поскольку мы будем использовать тот же масштаб, что и в `DOMDisplay`. Таким образом, смещение для плиток лавы равно 20 (значение привязки `scale`), а смещение для стен равно 0.

Мы не будем ждать, пока закончится загрузка изображения со спрайтами. Вызов `drawImage` для изображения, которое еще не было загружено, просто ни к чему не приведет. Таким образом, мы можем не нарисовать игру должным образом для первых нескольких кадров, пока загружается изображение, но это не является серьезной проблемой. Поскольку мы продолжим обновлять экран, правильная сцена появится сразу после завершения загрузки.

Для представления игрока будет использоваться показанный ранее ходячий персонаж. Код, который его рисует, должен выбирать правильный спрайт и направление в зависимости от текущего движения игрока. Первые восемь спрайтов содержат анимацию ходьбы. Когда игрок движется по полу, мы перебираем эти спрайты в цикле, ориентируясь по текущему времени. Мы будем менять кадры каждые 60 миллисекунд, поэтому вначале время делится на 60. Когда игрок стоит на месте, мы рисуем девятый спрайт. Во время прыжков, которые распознаются по тому, что вертикальная скорость не равна нулю, мы используем десятый, самый правый, спрайт.

Поскольку спрайты немного шире, чем объект игрока, — 24 вместо 16 пикселов, чтобы оставить место для ног и рук, — метод должен изменить x-координату и ширину на заданную величину (`playerXOverlap`).

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0) {
    this.flipPlayer = player.speed.x < 0;
  }

  let tile = 8;
  if (player.speed.y != 0) {
    tile = 9;
  } else if (player.speed.x != 0) {
    tile = Math.floor(Date.now() / 60) % 8;
  }

  this.cx.save();
  if (this.flipPlayer) {
    flipHorizontally(this.cx, x + width / 2);
  }

  let tileX = tile * width;
  this.cx.drawImage(playerSprites, tileX, 0, width, height,
                                   x,     y, width, height);
  this.cx.restore();
};
```

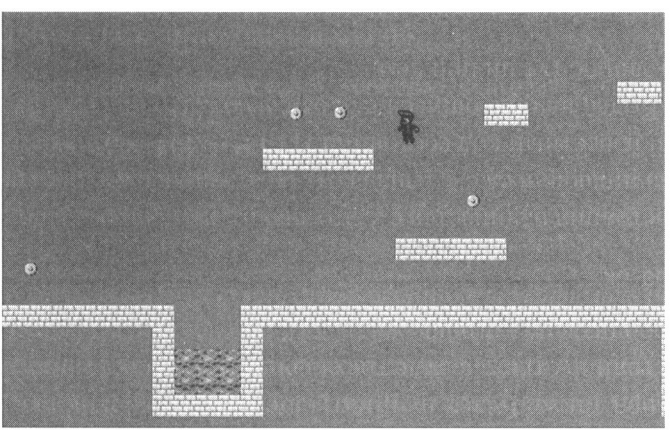
Метод `drawPlayer` вызывается методом `drawActors`, который отвечает за рисование всех акторов в игре.

```
CanvasDisplay.prototype.drawActors = function(actors) {
  for (let actor of actors) {
    let width = actor.size.x * scale;
    let height = actor.size.y * scale;
    let x = (actor.pos.x - this.viewport.left) * scale;
    let y = (actor.pos.y - this.viewport.top) * scale;
    if (actor.type == "player") {
      this.drawPlayer(actor, x, y, width, height);
    } else {
      let tileX = (actor.type == "coin" ? 2 : 1) * scale;
      this.cx.drawImage(otherSprites,
                        tileX, 0, width, height,
                        x,     y, width, height);
    }
  }
};
```

Рисуя что-то, что не является игроком, мы по его типу определяем правильное смещение для спрайта. Плитка лавы имеет смещение 20, а спрайт монеты — смещение 40 (в два раза больше).

При вычислении позиции актора мы должны вычесть позицию окна просмотра, поскольку точка (0, 0) на нашем холсте соответствует верхнему левому углу окна просмотра, а не верхнему левому краю уровня. Мы также могли бы использовать для этого метод `translate` — он будет работать в любом случае.

Вот и все, что касалось новой системы отображения. Получившаяся игра выглядит примерно так.



Выбор графического интерфейса

Итак, если вам нужно сгенерировать графику в браузере, вы можете выбирать между простым HTML, SVG и холстом. *Наилучшего* подхода на все случаи жизни не существует. У каждого варианта есть свои сильные и слабые стороны.

Преимущество простого HTML — в его простоте. Он также хорошо интегрируется с текстом. SVG и холст тоже позволяют рисовать текст, но они не помогут вам позиционировать этот текст или разместить его в нескольких строках. В изображение на основе HTML гораздо проще включать блоки текста.

Формат SVG удобен для создания четкой графики, которая хорошо выглядит при любом уровне масштабирования. В отличие от HTML он предназначен специально для рисования и поэтому лучше подходит для данной цели.

И SVG, и HTML строят структуру данных (DOM), которая описывает изображение. Это позволяет изменять элементы после их отрисовки. Если вам нужно многократно изменять небольшую часть большой картинки в ответ на действия пользователя или как часть анимации, выполнение этого на холсте может потребовать слишком много ресурсов. DOM также позволяет регистрировать обработчики событий мыши для каждого элемента изображения (даже для фигур, нарисованных с помощью SVG). На холсте это сделать невозможно.

Но при использовании холста пиксельно-ориентированный подход может быть преимуществом при рисовании очень большого количества очень маленьких элементов. Тот факт, что при этом не создается структура данных, а лишь многократно создается изображение на одной и той же растровой поверхности, обеспечивает для холста более низкую стоимость каждой фигуры.

Существуют также такие эффекты, как визуализация сцены по одному пикселу за раз (например, с использованием трассировщика лучей) или постобработка изображения с помощью JavaScript (размытие или искажение), которые физически могут быть реализованы только при растровом подходе.

Иногда применяют объединение нескольких из этих методов. Например, мы могли бы нарисовать граф с применением SVG или холста, но показать текстовую информацию, разместив HTML-элемент поверх изображения.

Для нетребовательных приложений не имеет большого значения, какой интерфейс вы выберете. Экран, который мы создали для игры в этой главе, можно было бы реализовать с использованием любой из этих трех графических технологий, поскольку здесь не нужно рисовать текст, обрабатывать взаимодействие с мышью или работать с необычайно большим количеством элементов.

Резюме

В этой главе мы обсудили технологии рисования графики в браузере, уделив особое внимание элементу `<canvas>` (холст).

Узел холста представляет собой область в документе, в которой программа может рисовать. Этот рисунок выполняется через объект контекста рисования, созданный с помощью метода `getContext`.

Интерфейс 2D-рисования позволяет заполнять и обводить контуром различные фигуры. Свойство контекста `fillStyle` определяет способ заполнения фигур. Свойства `strokeStyle` и `lineWidth` выявляют способ рисования линий.

Прямоугольники и фрагменты текста можно нарисовать с помощью вызова одного метода. Методы `fillRect` и `strokeRect` создают прямоугольники, а методы `fillText` и `strokeText` выводят текст. Чтобы создать нестандартные фигуры, нужно сначала создать путь.

Вызов метода `beginPath` создает новый путь. Еще несколько методов позволяют добавлять линии и кривые к текущему пути. Например, чтобы добавить прямую линию, нужно вызвать `lineTo`. Если путь замкнут, его можно заполнить цветом с помощью метода `fill` или добавить контур методом `stroke`.

Перемещение пикселов из изображения или с другого холста на наш холст выполняется с помощью метода `drawImage`. По умолчанию этот метод рисует все исходное изображение, но с помощью дополнительных параметров можно скопировать определенную его область. Мы использовали это в нашей игре, копируя отдельные позы игрового персонажа из изображения, содержащего множество таких поз.

Преобразования позволяют рисовать фигуры в нескольких ориентациях. В контексте 2D-рисования сохраняется текущее преобразование, которое

можно изменить с помощью методов `translate`, `scale` и `rotate`. Это повлияет на все последующие операции рисования. Состояние преобразования можно сохранить с помощью метода `save` и восстановить методом `restore`.

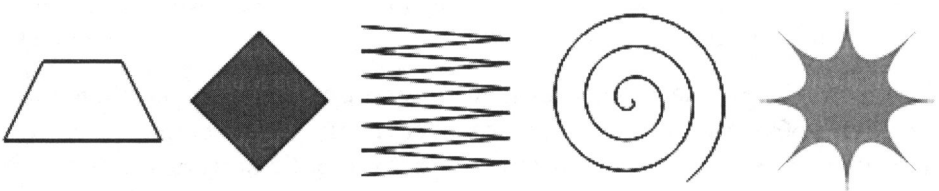
При отображении анимации на холсте можно использовать метод `clearRect` для очистки части холста перед его перерисовкой.

Упражнения

Фигуры

Напишите программу, рисующую на холсте следующие фигуры:

- трапецию (четырехугольник, который с одной стороны шире, а с другой — у́же);
- красный ромб (прямоугольник, повернутый на 45 градусов или 1/4 π радиан);
- зигзагообразную линию;
- спираль, состоящую из 100 прямых отрезков;
- желтую звезду.



Когда будете рисовать две последние фигуры, можете обратиться к описанию `Math.cos` и `Math.sin` в главе 14, где показано, как с помощью этих функций можно получить координаты точки, расположенной на окружности.

Советую создать для каждой фигуры отдельную функцию. Передавайте ей в качестве параметров положение и при необходимости другие свойства, такие как размер или количество точек. Если вместо этого жестко кодировать числа во всем коде, код станет слишком трудно читать и изменять.

Круговая диаграмма

Ранее в этой главе мы видели пример программы, создающей круговую диаграмму. Измените данную программу так, чтобы имя каждой категории отображалось рядом с сегментом, который ее представляет. Попробуйте подобрать приятный на вид способ автоматического позиционирования этого текста, который бы работал для других наборов данных.

Прыгающий шарик

Используя метод `requestAnimationFrame`, описанный в главах 14 и 16, нарисуйте прямоугольник с прыгающим шариком внутри. Шарик движется с постоянной скоростью и отскакивает от стенок прямоугольника, когда касается их.

Заранее рассчитанное зеркальное отражение

Один из неприятных моментов, касающихся преобразований, заключается в том, что они замедляют рисование растровых изображений. Необходимо преобразовать положение и размер каждого пиксела. Возможно, в будущем браузеры станут более умными в отношении преобразования, но пока что они заметно замедляются при рисовании растровых изображений.

В такой игре, как наша, где мы рисуем только один трансформированный спрайт, это не проблема. Но представьте, что нам нужно нарисовать сотни символов или тысячи вращающихся частиц, полученных в результате взрыва.

Придумайте способ, позволяющий рисовать перевернутый символ без загрузки дополнительных файлов изображений и без необходимости преобразовывать `drawImage` при вызове каждого кадра.

18 HTTP и формы

> Коммуникация по своей природе должна быть процессом без сохранения состояния [...], при котором каждый запрос от клиента к серверу содержал бы всю информацию, необходимую для понимания запроса, и не мог бы использовать какой-либо сохраненный контекст на сервере.
>
> *Рой Филдинг. Архитектурные стили и проектирование сетевых программных архитектур*

Протокол передачи гипертекста (Hypertext Transfer Protocol), уже упоминавшийся в главе 13, представляет собой механизм запросов и передачи данных в World Wide Web. В этой главе указанный протокол описывается более подробно; здесь будет показано, каким образом можно получить к нему доступ из браузера посредством JavaScript.

Протокол

Если ввести в адресной строке браузера адрес eloquentjavascript.net/18_http.html, то браузер сначала найдет адрес сервера, связанного с eloquentjavascript.net, а затем попытается открыть для него TCP-соединение через порт 80, который используется по умолчанию для HTTP-трафика. Если такой сервер существует и принимает соединение, то браузер может передать что-нибудь наподобие этого:

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: имя браузера
```

Сервер ответит через это же соединение:

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

<!doctype html>
... остальная часть документа
```

После этого браузер выбирает часть ответа, которая находится после пустой строки, — *тело* (не путать с тегом HTML `<body>`) — и отображает его в виде HTML-документа.

Информация, отправляемая клиентом, называется *запросом*. Запрос начинается со следующей строки:

```
GET /18_http.html HTTP/1.1
```

Первое слово — это *метод* запроса. `GET` означает, что мы хотим *получить* указанный ресурс. Другими распространенными методами являются `DELETE` для удаления ресурса, `PUT` для его создания или замены и `POST` для отправки информации. Обратите внимание, что сервер не обязан выполнять каждый запрос, который он получает. Если вы зайдете на случайный сайт и дадите ему запрос на удаление главной страницы, то, скорее всего, получите отказ.

То, что стоит после имени метода, — это путь к *ресурсу*, к которому применяется запрос. В простейшем случае ресурс — просто файл на сервере, но протокол не требует его наличия. Ресурс может быть чем угодно, что может быть передано в виде файла. Многие серверы динамически генерируют ответы. Например, если открыть страницу https://github.com/marijnh, то сервер будет искать в своей базе данных пользователя с именем marijnh, и если он его найдет, то сгенерирует страницу профиля для этого пользователя.

После пути к ресурсу в первой строке запроса указывается `HTTP/1.1` — используемая версия HTTP-протокола.

На практике многие сайты применяют HTTP версии 2, которая поддерживает те же концепции, что и версия 1.1, но намного сложнее, так что она может работать быстрее. При обращении к серверу браузеры автоматически переключаются на соответствующую версию протокола, так что результат запроса остается одинаковым независимо от используемой версии.

Поскольку версия 1.1 более проста и с ней легче иметь дело, мы сосредоточимся именно на ней.

Ответ сервера также будет начинаться с версии протокола, после чего указано состояние ответа в виде трехзначного кода состояния, за которым следует удобочитаемая строка.

```
HTTP/1.1 200 OK
```

Если код состояния начинается с 2, это значит, что запрос выполнен успешно. Коды, начинающиеся с 4, означают, что с запросом что-то не так. Вероятно, самый известный код состояния HTTP — 404 — означает, что ресурс не найден. Коды, начинающиеся с 5, означают, что на сервере произошла ошибка, но запрос тут ни при чем.

После первой строки запроса или ответа может идти любое количество заголовков. Это строки в формате «имя: значение», в которых содержится дополнительная информация о запросе или ответе. Следующие заголовки были частью примера ответа:

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Thu, 04 Jan 2018 14:05:30 GMT
```

Здесь определяется размер и тип документа, полученного в ответ. В данном случае это HTML-документ размером 65 585 байт. Здесь также сообщается, когда этот документ был изменен последний раз.

В большинстве случаев клиент и сервер сами решают, включать заголовки в запрос или ответ. Но иногда заголовки являются обязательными. Например, заголовок `Host`, который задает имя хоста, должен быть включен в запрос, поскольку сервер может обслуживать несколько имен хостов на одном IP-адресе и без этого заголовка сервер не будет знать, к какому имени хоста пытается обратиться клиент.

После заголовков запросы и ответы могут содержать пустую строку, за которой следует тело, где содержатся отправляемые данные. Запросы `GET` и `DELETE` не отправляют данных, а запросы `PUT` и `POST` — отправляют. Аналогичным образом некоторые типы ответов, такие как сообщения об ошибках, не требуют тела.

Браузеры и HTTP

Как мы видели в примере, после того как мы ввели URL-адрес в адресную строку, браузер отправляет запрос. Если полученная HTML-страница ссылается на другие файлы, такие как изображения и сценарии JavaScript, эти файлы также будут переданы.

Сайт средней сложности вполне может включать в себя от 10 до 200 ресурсов. Чтобы быстро их получать, браузеры выполняют одновременно несколько GET-запросов, вместо того чтобы ожидать ответы по одному.

HTML-страницы могут включать в себя *формы*, которые позволяют пользователю вводить информацию и отправлять ее на сервер. Вот пример такой формы:

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

Этот код описывает форму с двумя полями: маленькое поле содержит запрос имени, а в большом нужно написать сообщение. Когда вы нажимаете кнопку Отправить, форма *отправляется* — это означает, что содержимое ее полей упаковывается в HTTP-запрос и браузер переходит к результату этого запроса.

Если атрибут `method` элемента `<form>` имеет значение `GET` (или отсутствует), информация в форме добавляется в конец URL-адреса, указанного в качестве значения для атрибута `action` в виде *строки запроса*. Браузер может сделать запрос на этот URL:

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

Вопросительный знак указывает на конец пути URL-адреса и начало запроса. После него идут пары имен и значений, соответствующие атрибуту `name` в элементах полей формы и содержимому этих элементов. Для разделения пар используется символ амперсанда (`&`).

Фактическое сообщение, закодированное в URL, — это `Yes?`, но вопросительный знак заменяется странным кодом. Отдельные символы в строках запроса должны быть экранированы. Вопросительный знак, представленный как `%3F`, является одним из них. Похоже, существует неписаное

правило, согласно которому у каждого формата должен быть собственный способ экранирования символов. В этом варианте, называемом *URL-кодировкой*, используется знак процента, после него идут две шестнадцатеричные (по основанию 16) цифры, кодирующие код символа. В данном случае 3F, что соответствует 63 в десятичной системе счисления, является кодом вопросительного знака. В JavaScript есть функции `encodeURIComponent` и `decodeURIComponent` для кодирования и декодирования этого формата.

```
console.log(encodeURIComponent("Yes?"));
// → Yes%3F
console.log(decodeURIComponent("Yes%3F"));
// → Yes?
```

Если изменить значение атрибута `method` в приведенном ранее примере HTML-формы на `POST`, то HTTP-запрос, создаваемый для отправки формы, будет использовать метод `POST` и поместит строку запроса в тело запроса, а не добавит ее в URL.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

GET-запросы следует использовать для таких запросов, у которых нет побочных эффектов и которые просто запрашивают информацию. Запросы, которые что-то изменяют на сервере — например, создают новую учетную запись или публикуют сообщение, — должны быть выражены другими методами, такими как POST. Клиентское программное обеспечение, такое как браузер, знает, что оно не должно слепо создавать POST-запросы, поэтому часто неявно создает GET-запросы — например, для предварительной выборки ресурса, который, по мнению браузера, должен скоро понадобиться пользователю.

Позже в этой главе мы еще вернемся к формам и рассмотрим, как можно взаимодействовать с ними из JavaScript.

Fetch

Интерфейс, через который браузер JavaScript делает HTTP-запросы, называется `fetch`. Поскольку этот интерфейс относительно новый, в нем эффективно используются промисы (что редко встречается в интерфейсах браузеров).

```
fetch("example/data.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

Вызов `fetch` возвращает промис, который разрешается для объекта `Response`, содержащего информацию об ответе сервера, такую как код состояния и заголовки. Заголовки обернуты в объект, подобный `Map`, обрабатывающий свои ключи (имена заголовков) как нечувствительные к регистру, потому что имена заголовков должны быть именно такими. Это означает, что `headers.get("Content-Type")` и `headers.get("content-TYPE")` будут возвращать одно и то же значение.

Обратите внимание, что промис, возвращаемый `fetch`, разрешается успешно, даже если сервер выдал ответ с кодом ошибки. Промис также *может быть* отклонен в случае сетевой ошибки или если сервер, которому адресован запрос, не найден.

Первым аргументом `fetch` является запрашиваемый URL-адрес. Если он не начинается с имени протокола (например, `http:`), то он обрабатывается как *относительный*, то есть интерпретируется относительно текущего документа. Если адрес начинается с косой черты (`/`), то он заменяет путь, который стоит после имени текущего сервера. В противном случае перед относительным URL ставится часть текущего пути вплоть до последнего символа косой черты.

Чтобы получить реальное содержание ответа, можно использовать его метод `text`. Поскольку первоначальный промис разрешается сразу, как только будут получены заголовки ответа, а чтение тела ответа может занять больше времени, это снова возвращает промис.

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
// → Содержимое data.txt
```

Аналогичный метод, называемый `json`, возвращает промис, разрешаемый до значения, которое можно получить при синтаксическом анализе тела как данных в формате JSON; если же тело не является допустимой структурой данных в JSON-формате, то промис отклоняется.

По умолчанию `fetch` использует для выполнения запроса метод `GET` и не включает в себя тело запроса. Но этот интерфейс можно настроить по-другому, передав ему в качестве второго аргумента объект с дополнительными параметрами. Например, следующий запрос пытается удалить `example/data.txt`:

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

Код состояния 405 означает, что «метод не разрешен», HTTP-сервер как бы говорит: «Я не могу это сделать».

Чтобы добавить тело запроса, можно использовать параметр `body`. Для того чтобы можно было настроить заголовки, предусмотрен параметр `headers`. Например, следующий запрос включает в себя заголовок `Range`, который дает серверу команду возвращать только часть ответа.

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → содержимое
```

Браузер автоматически добавляет некоторые заголовки запроса, такие как `Host` и те, которые необходимы серверу для определения размера тела. Но добавление собственных заголовков часто бывает полезно для включения таких вещей, как данные аутентификации, или для указания серверу того, какой формат файла вы хотите получить.

HTTP-«песочница»

Выполнение HTTP-запросов в сценариях веб-страниц в очередной раз вызывает беспокойство по поводу безопасности. У того, кто управляет сценарием, могут быть другие интересы относительно того, на чьем компьютере он работает. Например, если я посещаю сайт themafia.org, то не хочу, чтобы его сценарии могли отправлять запросы на мой сайт mybank.com, используя идентификационную информацию из моего браузера с инструкциями по переводу всех моих денег на какой-либо случайный счет.

По этой причине браузеры защищают нас, не позволяя сценариям отправлять HTTP-запросы на другие домены (такие как themafia.org и mybank.com).

Это может быть досадной проблемой при создании систем, которые хотят получать доступ к разным доменам на законных основаниях. К счастью, серверы могут включить в свой ответ следующий заголовок, явно указывающий браузеру, что запрос на отправку из другого домена является допустимым:

```
Access-Control-Allow-Origin: *
```

Цените HTTP по достоинству

При построении системы, которая требует взаимодействия между JavaScript-программой, работающей в браузере (на стороне клиента), и программой на сервере (на стороне сервера), есть несколько способов моделирования этого взаимодействия.

Обычно используется модель *удаленных вызовов процедур*. В ней коммуникация происходит по принципу обычных вызовов функций, за исключением того, что на самом деле функция выполняется на другом компьютере. Вызов функции представляет собой запрос к серверу, который включает имя функции и аргументы. В ответе на этот запрос содержится возвращаемое значение.

В отношении удаленных вызовов процедур HTTP всего лишь средство коммуникации, и вы, скорее всего, напишете слой абстракции, который полностью его скроет.

Другой подход заключается в том, чтобы построить коммуникацию на основе концепции ресурсов и методов HTTP. Вместо удаленной процедуры, называемой `addUser`, можно использовать PUT-запрос к `/users/larry`. Вместо того чтобы представлять свойства этого пользователя в виде аргументов функции, можно определить формат JSON-документа (или использовать существующий формат), описывающий пользователя. Тело PUT-запроса для создания нового ресурса является таким документом. Ресурс выбирается путем выполнения GET-запроса для URL-адреса ресурса (например, `/user/larry`), который снова возвращает документ, представляющий собой ресурс.

Этот второй подход упрощает использование отдельных возможностей HTTP, таких как поддержка кэширования ресурсов (хранение копии на клиенте для быстрого доступа). Хорошо разработанные концепции HTTP предоставляют полезный набор принципов для разработки интерфейса вашего сервера.

HTTPS и безопасность

Данные имеют тенденцию передаваться по Интернету длинными и опасными путями. Чтобы добраться до пункта назначения, им приходится проходить через что угодно: от точек доступа Wi-Fi в кафе до сетей, контролируемых различными компаниями и разными государствами. В любой точке своего маршрута данные могут быть просмотрены или даже изменены.

Если вам важно сохранить данные в секрете (например, пароль к учетной записи электронной почты) или чтобы они попали в пункт назначения без изменений (например, номер счета, на который вы переводите деньги через сайт банка), то простой HTTP недостаточно хорош для этого.

Безопасный HTTP-протокол, используемый для URL-адресов, начинающихся с `https://`, обертывает HTTP-трафик таким образом, чтобы его было труднее прочитать и подделать. Перед обменом данными клиент проверяет, является ли сервер тем, за что он себя выдает, запрашивая доказательство наличия у него криптографического сертификата, выданного центром сертификации, который распознает браузер. Затем все данные, передаваемые по соединению, шифруются таким образом, чтобы предотвратить прослушивание и подделку.

Если все сделано правильно, HTTPS не позволяет посторонним выдавать себя за сайт, с которым вы пытаетесь общаться, и отслеживать ваши коммуникации. Это неидеальное решение, и бывали случаи сбоев HTTPS из-за поддельных или украденных сертификатов, а также взломанного программного обеспечения, но это намного безопаснее, чем простой HTTP.

Поля форм

Изначально формы были разработаны для сайтов, еще не использующих JavaScript, чтобы отправлять пользовательскую информацию в виде HTTP-запросов. Формы предполагают, что взаимодействие с сервером всегда происходит путем перехода на новую страницу.

Но элементы форм являются частью DOM-модели, как и остальная часть страницы, а элементы DOM, представляющие поля формы, поддерживают ряд свойств и событий, которых нет в других элементах. Это позволяет инспектировать и контролировать поля ввода с помощью JavaScript-программ и выполнять такие действия, как добавление новых функций в форму или использование форм и полей в качестве строительных блоков в JavaScript-приложениях.

Веб-форма состоит из множества полей ввода, сгруппированных внутри тега `<form>`. В HTML существует несколько видов полей, от простых переключателей типа «включено/выключено» до выпадающих меню и полей ввода текста. В этой книге мы не будем всесторонне обсуждать все типы полей, а ограничимся лишь общим обзором.

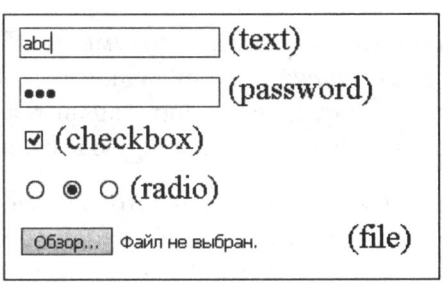
Многие типы полей используют тег `<input>`. Атрибут `type` этого тега применяется для выбора типа поля. Вот некоторые часто используемые типы `<input>`:

- `text` — текстовое поле;
- `password` — то же, что `text`, но скрывает набранный текст;
- `checkbox` — переключатель типа «включено/выключено»;
- `radio` — поля с множественным выбором;
- `file` — поле, позволяющее пользователю выбрать файл со своего компьютера.

Поля формы не обязательно должны помещаться внутри тега `<form>`. Их можно поставить в любом месте на странице. Но без формы содержимое таких полей нельзя отправить (можно отправить только всю форму), однако если реакция на ввод реализована с помощью JavaScript, мы часто все равно не хотим отправлять содержимое полей.

```
<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"> (file)</p>
```

Поля, созданные с помощью этого HTML-кода, выглядят так:



Интерфейс JavaScript для таких элементов зависит от типа элемента.

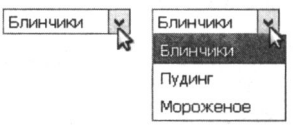
Многострочные текстовые поля имеют собственный тег `<textarea>` в основном потому, что использование атрибута для указания начального значения многострочного текста будет неудобным. Тег `<textarea>` требует соответствующего закрывающего тега `</textarea>`, а текст между ними применяется вместо атрибута `value` в качестве начального текста.

```
<textarea>
раз
два
три
</textarea>
```

Наконец, тег `<select>` используется для создания поля, которое позволяет пользователю выбирать из нескольких предопределенных параметров.

```
<select>
  <option>Блинчики</option>
  <option>Пудинг</option>
  <option>Мороженое</option>
</select>
```

Это поле выглядит так:



Всякий раз, когда значение поля формы изменяется, возникает событие `"change"`.

Фокус

В отличие от большинства элементов в документах HTML, поля формы могут получать *фокус клавиатуры*. Если щелкнуть на поле или каким-либо другим способом его активировать, оно становится текущим активным элементом и получателем информации при вводе с клавиатуры.

Таким образом, можно вводить текст в текстовое поле ввода только тогда, когда оно получило фокус. Другие поля по-разному реагируют на события

клавиатуры. Например, меню `<select>` пытается перейти к параметру, который содержит текст, введенный пользователем, и реагирует на клавиши со стрелками, перемещая выбор вверх и вниз.

Мы можем управлять фокусом из JavaScript с помощью методов `focus` и `blur`. Метод `focus` перемещает фокус на DOM-элемент, для которого вызывается этот метод, а `blur` удаляет фокус. Значение `document.activeElement` соответствует текущему элементу, на котором находится 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 ожидается, что пользователь сразу захочет взаимодействовать с полем формы. Можно использовать JavaScript для передачи фокуса на это поле при загрузке документа, но в HTML также есть атрибут `autofocus`, обеспечивающий тот же эффект, сообщая браузеру, чего мы пытаемся достичь. Это дает браузеру возможность отключить поведение, если оно не подходит, — например, если пользователь перенес фокус на что-то другое.

По традиции браузеры также позволяют пользователю перемещать фокус по документу, нажимая клавишу табуляции. Мы можем влиять на последовательность, в которой элементы получают фокус, с помощью атрибута `tabindex`. В следующем примере документа показано, как можно перемещать фокус с текстового поля ввода на кнопку `OK`, вместо того чтобы сначала перейти по ссылке справки:

```
<input type="text" tabindex=1> <a href=".">(help)</a>
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

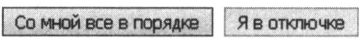
По умолчанию большинство типов HTML-элементов не могут иметь фокус. Но к любому элементу можно добавить атрибут `tabindex`, который сделает его фокусируемым. Если `tabindex` равен `-1`, то при нажатии табуляции этот элемент будет пропускаться, даже если обычно он является фокусируемым.

Отключенные поля

Любое поле формы можно *отключить* с помощью атрибута `disabled`. Это атрибут, который можно указывать без значения, — сам факт, что он присутствует, отключает элемент.

```
<button>Со мной все в порядке</button>
<button disabled>Я в отключке</button>
```

На отключенные поля нельзя перенести фокус и их нельзя изменить, а в браузерах они выглядят серыми и блеклыми.



Когда программа находится в процессе обработки действия, вызванного какой-либо кнопкой или другим элементом управления, и это может потребовать связи с сервером и, следовательно, занять некоторое время, иногда имеет смысл отключить элемент управления до завершения данного действия. Тогда, если пользователь потеряет терпение и снова щелкнет на элементе, он случайно не повторит свое действие.

Форма в целом

Если поле содержится в элементе `<form>`, у его DOM-элемента есть свойство `form`, связывающее этот элемент с DOM-элементом формы. У элемента `<form>`, в свою очередь, есть свойство `elements`, которое содержит массивоподобную коллекцию содержащихся в форме полей.

Атрибут `name` поля формы определяет способ выявления значения этого поля при отправке формы. Этот атрибут также можно использовать как имя свойства при доступе к свойству `elements` формы, которое действует как массивоподобный объект (с доступом по номеру) и как соответствие (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>
  let 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>
```

При нажатии кнопки с атрибутом `type`, равным `submit`, форма будет отправлена. То же самое произойдет, если нажать ENTER, когда фокус находится на одном из полей формы. Отправка формы обычно означает, что браузер переходит на страницу, указанную в атрибуте `action` формы, используя запрос GET или POST. Но прежде, чем это произойдет, запускается событие `submit`. Это событие можно обработать с помощью JavaScript и предотвратить поведение формы по умолчанию, вызвав для объекта события `preventDefault`.

```
<form action="example/submit.html">
  Значение: <input type="text" name="value">
  <button type="submit">Сохранить</button>
</form>
<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Сохранение значения", form.elements.value.value);
    event.preventDefault();
  });
</script>
```

Перехват событий `"submit"` в JavaScript имеет множество применений. Мы можем написать код, чтобы убедиться, что значения, введенные пользователем, имеют смысл, и вместо отправки формы сразу вывести сообщение об ошибке. Или же можно вообще отключить обычный способ отправки формы, как показано в примере, и заставить программу обрабатывать ввод, возможно используя `fetch` для отправки данных на сервер без перезагрузки страницы.

Текстовые поля

Поля, созданные с помощью тегов `<textarea>` или `<input>` с типом `text` или `password`, имеют общий интерфейс. Их DOM-элементы имеют свойство

value, которое хранит их текущее содержимое в виде строкового значения. При изменении значения этого свойства изменяется и содержимое поля.

Свойства selectionStart и selectionEnd текстовых полей дают информацию о курсоре и выделении в тексте. Если ничего не выбрано, эти два свойства содержат одинаковое число, соответствующее положению курсора. Например, 0 указывает на начало текста, а 10 — что курсор находится после десятого символа. Если часть поля выделена, то значения этих свойств будут различаться, сообщая о начале и конце выделенного текста. Подобно value, данные свойства также могут быть изменены.

Представьте себе, что вы пишете статью о фараоне Хасехемуи, но у вас возникли проблемы с написанием его имени. Следующий код связывает тег <textarea> с обработчиком событий, который, если нажать F2, вставляет строку "Хасехемуи".

```
<textarea></textarea>
<script>
  let textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", event => {
    // Код клавиши для F2, оказывается, 113
    if (event.keyCode == 113) {
      replaceSelection(textarea, "Хасехемуи");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    let from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
                  field.value.slice(to);
    // Поставить курсор после слова
    field.selectionStart = from + word.length;
    field.selectionEnd = from + word.length;
  }
</script>
```

Функция replaceSelection заменяет текущую выделенную часть содержимого текстового поля заданным словом, а затем ставит курсор после этого слова, чтобы пользователь мог продолжить набор текста.

Событие "change" для текстового поля срабатывает не каждый раз, когда что-то вводится, а тогда, когда поле теряет фокус после изменения его содержимого. Чтобы сразу реагировать на изменения в текстовом поле, нужно зарегистрировать обработчик для события "input", которое возникает

каждый раз, когда пользователь вводит символ, удаляет текст или иным образом изменяет содержимое поля.

В следующем примере показаны текстовое поле и счетчик, отображающий текущую длину текста в этом поле:

```
<input type="text"> длина: <span id="length">0</span>
<script>
  let text = document.querySelector("input");
  let output = document.querySelector("#length");
  text.addEventListener("input", () => {
    output.textContent = text.value.length;
  });
</script>
```

Флажки и переключатели

Поле флажка является двоичным переключателем. Его значение можно извлечь или изменить через свойство `selected`, которое содержит логическое значение.

```
<label>
  <input type="checkbox" id="purple">Сделать страницу розовой
</label>
<script>
  let checkbox = document.querySelector("#purple");
  checkbox.addEventListener("change", () => {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

Тег `<label>` связывает часть документа с полем ввода. Если щелкнуть в любом месте внутри тега `<label>`, то это поле активируется, на него перемещается фокус, а если это флажок или переключатель, то его значение переключается.

Переключатель похож на флажок, но он неявно связан с другими переключателями, имеющими тот же атрибут `name`, поэтому в любой момент только один из группы переключателей может быть активным.

```
Цвет:
<label>
  <input type="radio" name="color" value="orange">Оранжевый
```

```
</label>
<label>
  <input type="radio" name="color" value="lightgreen">Зеленый
</label>
<label>
  <input type="radio" name="color" value="lightblue">Синий
</label>
<script>
  let buttons = document.querySelectorAll("[name=color]");
  for (let button of Array.from(buttons)) {
    button.addEventListener("change", () => {
      document.body.style.background = button.value;
    });
  }
</script>
```

Квадратные скобки в CSS-запросе, заданном `querySelectorAll`, используются для сопоставления атрибутов. В этом случае выбираются элементы, у которых атрибут `name` имеет значение `"color"`.

Поля выбора

Поля выбора концептуально аналогичны переключателям — они также позволяют пользователю выбирать из набора вариантов. Но если переключатель оставляет расположение вариантов на наше усмотрение, то внешний вид тега `<select>` определяется браузером.

У полей выбора тоже есть вид, который больше похож на список флажков, а не на переключатели. Если у тега `<select>` есть атрибут `multiple`, то пользователь может выбирать не один вариант, а любое их количество. В большинстве браузеров это будет отображаться иначе, чем обычное поле выбора, которое чаще всего представлено в виде *раскрывающегося списка*, показывающего варианты выбора, только если его открыть.

У каждого тега `<option>` есть свое значение. Оно может быть определено с помощью атрибута `value`. Если значение не указано, то значением является текст внутри тега. Свойство `value` элемента `<select>` отражает текущий выбранный параметр. Однако в случае, если указан параметр `multiple`, это свойство не имеет большого значения, поскольку оно равно значению *только одного* из выбранных в данный момент вариантов.

Теги `<option>` для поля `<select>` могут быть доступны как массивоподобный объект через свойство поля `options`. У каждого варианта есть свойство `selected`, которое указывает, выбран ли этот вариант в данный момент. Это свойство также можно указать специально, чтобы выбрать или отменить выбор определенного варианта.

В следующем примере извлекаются выбранные значения из поля выбора с активированным параметром `multiple` и затем используются для составления двоичного числа из отдельных битов. Чтобы выбрать несколько вариантов, удерживайте нажатой клавишу Control (или Command на Mac).

```
<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>
  let select = document.querySelector("select");
  let output = document.querySelector("#output");
  select.addEventListener("change", () => {
    let number = 0;
    for (let option of Array.from(select.options)) {
      if (option.selected) {
        number += Number(option.value);
      }
    }
    output.textContent = number;
  });
</script>
```

Поля выбора файлов

Поля выбора файлов изначально создавались как способ загрузки файлов с пользовательского компьютера через форму. В современных браузерах эти поля также позволяют читать такие файлы из программ JavaScript. Поле действует как своего рода шлюз. Сценарий не может просто начать чтение личных файлов с пользовательского компьютера, но если пользователь выбирает файл в таком поле, браузер интерпретирует это действие как разрешение — теперь сценарий может прочитать файл.

Поле выбора файла обычно выглядит как кнопка с надписью вроде Выбрать файл или Просмотреть и расположенной рядом с ней информацией о выбранном файле.

```
<input type="file">
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0];
      console.log("Вы выбрали", file.name);
      if (file.type) console.log("Его тип", file.type);
    }
  });
</script>
```

Свойство `files` элемента поля для выбора файла представляет собой массивоподобный объект (снова не настоящий массив), содержащий файлы, выбранные в этом поле. Изначально поле является пустым. Причина того, что не используется свойство `file`, заключается в том, что поля выбора файлов также поддерживают атрибут `multiple`, и это позволяет выбирать несколько файлов одновременно.

Объекты, входящие в состав объекта `files`, имеют такие свойства, как `name` (имя файла), `size` (размер файла в байтах — наборах из 8 бит) и `type` (тип данных файла, такой как `text/plain` или `image/jpeg`).

А вот свойства, содержащего содержимое файла, у этих объектов нет. Получить его немного сложнее. Поскольку чтение файла с диска может занять некоторое время, интерфейс должен быть асинхронным, чтобы исключить зависание документа.

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("Файл", file.name, "начинается с",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
```

```
  });
</script>
```

Для чтения файла создается объект `FileReader`, затем для этого объекта регистрируется обработчик события `"load"` и вызывается метод `readAsText`, которому передается файл, подлежащий прочтению. После завершения загрузки свойство `result` метода чтения содержит содержимое файла.

Объект `FileReader` также создает событие `"error"`, если чтение файла по любой причине завершается неудачей. Сам объект ошибки попадает в свойство `error` объекта чтения. Этот интерфейс был разработан до того, как в языке появились промисы. Но его можно обернуть в промис, например, так:

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}
```

Хранение данных на стороне клиента

Простые HTML-страницы с небольшим количеством кода на JavaScript могут быть отличным форматом для «мини-приложений» — небольших вспомогательных программ, которые автоматизируют выполнение базовых задач. Соединив несколько полей формы с обработчиками событий, можно делать что угодно: от преобразований сантиметров в дюймы до вычисления паролей на основе мастер-пароля и имени сайта.

Если такое приложение должно что-то помнить между сеансами, нельзя использовать привязки JavaScript — они сбрасываются каждый раз, когда закрывается страница. Мы можем настроить сервер, подключить его к Интернету и сделать так, чтобы приложение что-то там хранило. В главе 20 будет показано, как это сделать. Но это означает много дополнительной работы и сложностей. Иногда достаточно просто сохранить данные в браузере.

Объект `localStorage` можно использовать для хранения данных таким образом, чтобы он выдерживал перезагрузки страницы. Этот объект позволяет хранить строковые значения, которым присвоены имена.

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

Значение сохраняется в `localStorage` до тех пор, пока оно не будет перезаписано, удалено с помощью `removeItem` или пока пользователь не очистит локальные данные на своем компьютере.

Сайты, находящиеся на разных доменах, получают разные отделения для хранения. Это означает, что данные, сохраненные в `localStorage` для определенного сайта, по определению могут быть прочитаны (и перезаписаны) только сценариями того же сайта.

Браузеры устанавливают ограничение на размер данных, которые сайт может хранить в `localStorage`. Это ограничение, а также то обстоятельство, что вообще-то не стоит засорять жесткие диски пользователей ненужными данными, не позволяют такой технологии занимать слишком много места.

В следующем коде реализовано примитивное приложение для создания заметок. Оно сохраняет множество именованных заметок, позволяет пользователю редактировать эти заметки и создавать новые.

```
Notes: <select></select> <button>Добавить</button><br>
<textarea style="width: 100%"></textarea>

<script>
  let list = document.querySelector("select");
  let note = document.querySelector("textarea");

  let state;
  function setState(newState) {
    list.textContent = "";
    for (let name of Object.keys(newState.notes)) {
      let option = document.createElement("option");
      option.textContent = name;
      if (newState.selected == name) option.selected = true;
      list.appendChild(option);
    }
```

```
    note.value = newState.notes[newState.selected];

    localStorage.setItem("Notes", JSON.stringify(newState));
    state = newState;
  }
  setState(JSON.parse(localStorage.getItem("Notes")) || {
    notes: {"shopping list": "Carrots\nRaisins"},
    selected: "shopping list"
  });

  list.addEventListener("change", () => {
    setState({notes: state.notes, selected: list.value});
  });
  note.addEventListener("change", () => {
    setState({
      notes: Object.assign({}, state.notes,
                           {[state.selected]: note.value}),
      selected: state.selected
    });
  });
  document.querySelector("button")
    .addEventListener("click", () => {
      let name = prompt("Note name");
      if (name) setState({
        notes: Object.assign({}, state.notes, {[name]: ""}),
        selected: name
      });
    });
</script>
```

В качестве начального состояния сценарий получает значение `"Notes"`, хранящееся в `localStorage`, или, если оно отсутствует, создает пример состояния, в котором содержится только список покупок. Попытка чтения из `localStorage` несуществующего поля дает `null`. Передача `null` в `JSON.parse` приводит к попытке синтаксического анализа строки `"null"` с результатом `null`. Таким образом, для предоставления значения по умолчанию в этой ситуации можно использовать оператор `||`.

Метод `setState` гарантирует, что DOM отображает заданное состояние, и сохраняет новое состояние в `localStorage`. Обработчики событий вызывают эту функцию, чтобы перейти к новому состоянию.

`Object.assign` в этом примере используется для создания нового объекта, который является клоном старого `state.notes`, но с добавленным или

перезаписанным свойством. `Object.assign` принимает первый аргумент и добавляет к нему все свойства из всех остальных аргументов. Таким образом, если передать пустой объект, то этот новый объект будет заполнен новыми значениями. Квадратные скобки в третьем аргументе используются для создания свойства, чье имя основано на определенном динамическом значении.

Существует еще один объект, похожий на `localStorage`, который называется `sessionStorage`. Разница между ними заключается в том, что содержимое `sessionStorage` забывается в конце каждого сеанса, то есть для большинства браузеров — всякий раз, когда закрывается их окно.

Резюме

В этой главе мы изучили работу протокола HTTP. *Клиент* отправляет запрос, который содержит метод (обычно GET), и путь, идентифицирующий ресурс. Затем *сервер* решает, что делать с запросом, и возвращает код состояния и тело ответа. Запросы и ответы могут содержать заголовки, содержащие дополнительную информацию.

Интерфейс, через который браузер JavaScript может делать HTTP-запросы, называется `fetch`. Создание запроса выглядит следующим образом:

```
fetch("/18_http.html").then(r => r.text()).then(text => {
  console.log(`Страница начинается с ${text.slice(0, 15)}`);
});
```

Браузеры используют GET-запросы для получения ресурсов, необходимых для отображения веб-страницы. Страница также может содержать формы, позволяющие отправлять введенную пользователем информацию в качестве запроса для новой страницы, которой будет адресована форма.

В HTML есть разные типы полей формы, такие как текстовые поля, флажки, поля с несколькими вариантами выбора и поля выбора файлов.

Такие поля могут быть проверены и обработаны с помощью JavaScript. При изменении этих полей возникает событие `"change"`, а при вводе текста и получении событий клавиатуры, когда поле получает фокус, — событие `"input"`. Для чтения или изменения содержимого поля используются такие

свойства, как `value` (для текста и полей выбора) или `checked` (для флажков и переключателей).

При отправке формы для нее возникает событие `"submit"`. Обработчик JavaScript может вызывать для этого события функцию `preventDefault`, чтобы отключить поведение браузера по умолчанию. Элементы полей формы также могут располагаться за пределами тега формы.

Если пользователь выбрал файл в локальной файловой системе и указал его в поле выбора файлов, для доступа к содержимому этого файла из JavaScript-программы можно использовать интерфейс `FileReader`.

Объекты `localStorage` и `sessionStorage` можно применять для сохранения информации таким образом, чтобы она выдерживала перезагрузку страницы. Первый объект сохраняет данные навсегда (или до тех пор, пока пользователь не решит их очистить), а второй — до закрытия браузера.

Упражнения

Согласование содержимого

Одна из вещей, которую позволяет делать HTTP, называется *согласованием содержимого*. Заголовок запроса `Accept` используется для указания серверу, какой тип документа клиент хочет получить. Многие серверы игнорируют этот заголовок, но, когда сервер допускает различные способы кодирования ресурса, он может просмотреть этот заголовок и отправить тот вариант, который предпочитает клиент.

URL-адрес `https://eloquentjavascript.net/author` настроен для ответа в формате открытого текста, HTML или JSON, в зависимости от того, что запрашивает клиент. Эти форматы идентифицируются стандартными типами данных `text/plain`, `text/html` и `application/json`.

Отправьте запросы на получение всех трех форматов данных с этого ресурса. Используйте свойство `headers` в объекте параметров, переданном для выборки, чтобы установить заголовок с именем `Accept` для нужного типа носителя.

Наконец, попробуйте запросить тип носителя `application/rainbows+unicorns` и посмотрите, какой код состояния получите.

Среда выполнения JavaScript

Создайте интерфейс, который позволял бы человеку вводить и выполнять фрагменты кода JavaScript.

Создайте поле `<textarea>` и кнопку рядом с ним, при нажатии которой использовался бы конструктор `Function`, описанный в главе 10, чтобы обернуть текст в функцию и вызвать ее.

Преобразуйте возвращаемое значение функции или возникшую ошибку в строку и отображайте ее под текстовым полем.

Игра «Жизнь» Конвея

Игра «Жизнь» Конвея — это простая симуляция, создающая искусственную «жизнь» на сетке, каждая клетка которой может быть живой или мертвой. Для каждого поколения (хода) выполняются следующие правила.

- Любая живая клетка, у которой меньше двух или больше трех живых соседей, умирает.
- Любая живая клетка, у которой два или три живых соседа, живет в следующем поколении.
- Любая мертвая клетка, у которой ровно три живых соседа, становится живой клеткой.

Соседней считается любая смежная ячейка, в том числе по диагонали.

Обратите внимание, что данные правила применяются ко всей сетке сразу, а не по одному квадрату за раз. Это означает, что подсчет соседей основан на ситуации в начале поколения и изменения, происходящие с соседними ячейками во время жизни поколения, не должны влиять на новое состояние данной ячейки.

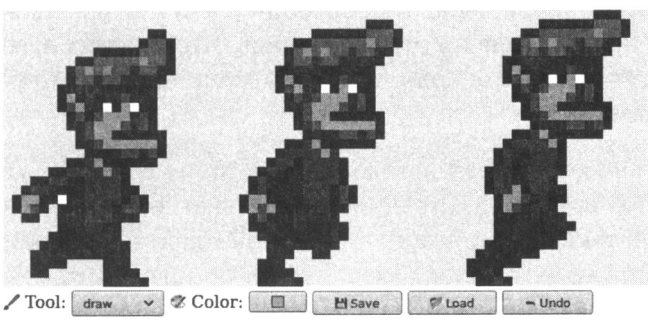
Реализуйте эту игру, используя любую структуру данных, которую считаете подходящей. Для того чтобы изначально заполнить сетку случайными данными, примените `Math.random`. Отобразите игру в виде сетки полей флажков, рядом с которой находится кнопка, позволяющая перейти к следующему поколению. Когда пользователь устанавливает или снимает флажки, их изменения должны учитываться при вычислении следующего поколения.

19 Проект: растровый графический редактор

> 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.
>
> *Хуан Миро*

Материал, представленный в предыдущих главах, дает нам все, что необходимо для создания простейшего веб-приложения. Именно это мы и сделаем в данной главе.

Нашим приложением будет растровая программа для рисования, где можно будет изменять изображение пиксел за пикселом, манипулируя увеличенным видом, отображаемым как сетка, состоящая из цветных квадратов. Такую программу можно использовать, чтобы открывать файлы с изображениями, рисовать в них с помощью мыши или другого указывающего устройства и сохранять их. Вот как это будет выглядеть.



Рисовать на компьютере — это здорово. Не приходится беспокоиться о материалах, навыках или талантах. Просто начните!

Компоненты

Интерфейс приложения представляет собой расположенный сверху большой элемент `<canvas>` и несколько полей формы под ним. Пользователь рисует на картинке, выбирая инструмент из поля `<select>`, а затем щелкая, касаясь указателя или перетаскивая его по холсту. Существуют инструменты для рисования отдельных пикселов и прямоугольников, для заливки области и для выбора цвета с картинки.

Мы будем структурировать интерфейс редактора как набор *компонентов* — объектов, которые отвечают за часть DOM и могут содержать внутри себя другие компоненты.

Состояние приложения состоит из текущего изображения, выбранного инструмента и выбранного цвета. Мы настроим все так, чтобы состояние хранилось в одном значении, а вид компонентов интерфейса всегда определялся текущим состоянием.

Чтобы понять, почему это так важно, рассмотрим альтернативу — распределение частей состояния по всему интерфейсу. До определенного момента так проще программировать. Мы можем просто вставить значение в поле цвета и прочитать его, когда нам потребуется узнать текущий цвет.

Но затем мы добавим селектор цвета — инструмент, который позволяет щелкнуть на изображении и выбрать цвет данного пиксела. Для того чтобы поле цвета показывало правильный цвет, этот инструмент должен знать, что такое поле существует, и обновлять его, когда выбирается новый цвет. Если вы когда-нибудь добавите еще один элемент программы, который делает цвет видимым (например, указатель мыши может отражать цвет), то вам придется опять обновить код изменения цвета, чтобы сохранить синхронизацию.

По сути, возникает проблема: каждая часть интерфейса должна знать обо всех остальных его частях. Это не очень модульный подход. Для небольших приложений, таких как описанное в данной главе, подобное, может, и не проблема. Но для больших проектов это может превратиться в настоящий кошмар.

Чтобы избежать такого кошмара в принципе, мы будем строго следить за *потоком данных*. Существует состояние, и экранный интерфейс отображается на основе этого состояния. Компонент интерфейса может реагировать на действия пользователя путем изменения состояния, после чего компоненты получают возможность синхронизироваться с этим новым состоянием.

На практике каждый компонент настраивается так, что, когда ему присваивается новое состояние, он также уведомляет свои дочерние компоненты о том, что они должны быть обновлены. Настройка — это немного хлопотно. Возможность сделать ее более удобной — основное преимущество многих библиотек для программирования браузеров. Но для такого маленького приложения мы можем обойтись без подобной инфраструктуры.

Изменения состояния будут представлены в виде объектов, которые мы будем называть *действиями*. Компоненты могут создавать такие действия и *отправлять* их — передавать в центральную функцию управления состоянием. Эта функция вычисляет следующее состояние, после чего компоненты интерфейса обновляются в соответствии с ним.

Мы берем на себя грязную задачу — запустить пользовательский интерфейс и применить к нему некоторую структуру. В тех его частях, что связаны с DOM, все еще полно побочных эффектов, но они хотя бы опираются на концептуально простую основу: цикл обновления состояния. Это состояние определяет, как выглядит DOM, и единственным способом, которым события DOM могут изменить состояние, является отправка действий в состояние.

Существует *огромное множество* вариантов этого подхода, каждый из которых имеет свои преимущества и проблемы, но их основная идея одна и та же: изменения состояния должны проходить через один четко определенный канал, а не происходить повсеместно.

Наши компоненты будут классами, соответствующими интерфейсу. Их конструктор получает состояние — это может быть состояние всего приложения или какое-то меньшее значение, если не требуется доступ ко всем элементам приложения, — и использует его для создания свойства `dom`. Это DOM-элемент, представляющий компонент. Большинство конструкторов также принимают другие значения, которые не изменятся со временем, такие как функция, которую они могут применить для отправки действия.

У каждого компонента есть метод `syncState`, задействуемый для синхронизации компонента с новым значением состояния. Этот метод принимает один

аргумент — состояние того же типа, что и первый аргумент конструктора соответствующего компонента.

Состояние

Состояние приложения представляет собой объект со свойствами `picture`, `tool` и `color`. Объект `picture` хранит ширину, высоту и пиксельное содержимое изображения. Пикселы хранятся в массиве, подобно классу матрицы из главы 6, построчно, сверху вниз.

```
class Picture {
  constructor(width, height, pixels) {
    this.width = width;
    this.height = height;
    this.pixels = pixels;
  }
  static empty(width, height, color) {
    let pixels = new Array(width * height).fill(color);
    return new Picture(width, height, pixels);
  }
  pixel(x, y) {
    return this.pixels[x + y * this.width];
  }
  draw(pixels) {
    let copy = this.pixels.slice();
    for (let {x, y, color} of pixels) {
      copy[x + y * this.width] = color;
    }
    return new Picture(this.width, this.height, copy);
  }
}
```

По причинам, к которым мы вернемся позже в данной главе, нам желательно иметь возможность рассматривать изображение как неизменяемое значение. Но нам также иногда нужно будет изменять множество пикселов одновременно. Для этого в классе есть метод `draw`, принимающий массив измененных пикселов — объектов со свойствами `x`, `y` и `color` — и создающий новое изображение, в котором эти пикселы перезаписаны. Для копирования всего массива пикселов в этом методе используется метод `slice` без аргументов: по умолчанию начало копируемого фрагмента равно 0, а конец — длине массива.

В методе `empty` задействованы две функциональные возможности массива, которые нам еще не встречались. Если вызвать конструктор `Array` и передать ему число, то будет создан пустой массив заданной длины. Затем можно использовать метод `fill`, чтобы заполнить этот массив заданным значением. Эти методы применяются для создания массива, в котором все пикселы имеют одинаковый цвет.

Цвета хранятся в виде строк, содержащих традиционные цветовые CSS-коды, состоящие из знака решетки (`#`), после чего идут шесть шестнадцатеричных (по основанию 16) чисел: по две для красного, зеленого и синего компонентов. Довольно непонятный и неудобный способ написания цветов, но это тот формат, который используется в поле ввода цвета HTML и который можно применять в свойстве `fillColor` контекста рисования холста, поэтому для нашего способа применения цветов в программе это достаточно практично.

Черный цвет, в котором все компоненты равны нулю, обозначается как `"#000000"`, а ярко-розовый выглядит как `"#ff00ff"`, где красный и синий компоненты имеют максимальные значения, равные 255 и записываемые в шестнадцатеричных цифрах как `ff` (где буквы от `a` до `f` используются для представления чисел от 10 до 15).

Мы позволяем интерфейсу передавать действия как объекты, свойства которых перезаписывают свойства предыдущего состояния. Когда пользователь изменяет значение в поле цвета, это поле отправляет объекты вида `{color: field.value}`, из которых функция обновления может вычислять новое состояние.

```
function updateState(state, action) {
  return Object.assign({}, state, action);
}
```

Этот довольно громоздкий шаблон, в котором `Object.assign` используется для того, чтобы сначала добавить свойства `state` к пустому объекту, а затем перезаписать отдельные из них свойствами из `action`, является широко распространенным в JavaScript способом использования неизменяемых объектов. Более удобная запись, с применением оператора троеточия для включения всех свойств другого объекта в выражение данного объекта, пока что находится на заключительной стадии стандартизации. С ее помощью можно было бы написать `{...state, ...action}`. Но на момент выхода этой книги такой код еще не работал во всех браузерах.

Построение DOM

Одна из основных функций компонентов интерфейса — создание структуры DOM. Здесь мы опять не хотим напрямую использовать для этого подробные методы DOM, поэтому вот немного расширенная версия функции `elt`:

```
function elt(type, props, ...children) {
  let dom = document.createElement(type);
  if (props) Object.assign(dom, props);
  for (let child of children) {
    if (typeof child != "string") dom.appendChild(child);
    else dom.appendChild(document.createTextNode(child));
  }
  return dom;
}
```

Основное различие между этой версией и той, которую мы использовали в главе 16, заключается в том, что она назначает узлам DOM *свойства*, а не *атрибуты*. Это значит, что мы не можем использовать данную функцию для установки произвольных атрибутов, но можем — для установки свойств, значение которых не является строкой, таких как `onclick`, которым можно задать функцию и тем самым зафиксировать обработчик события щелчка.

Это позволяет использовать следующий стиль регистрации обработчиков событий:

```
<body>
  <script>
    document.body.appendChild(elt("button", {
      onclick: () => console.log("click")
    }, "The button"));
  </script>
</body>
```

Холст

Первый компонент, который мы определим, — это часть интерфейса, отображающая рисунок в виде сетки цветных квадратов. Данный компонент отвечает за две вещи: показ изображения и передачу событий указателя, возникающих на этом изображении, остальной части приложения.

Таким образом, мы можем определить его как компонент, который знает только о текущем изображении, а не обо всем состоянии приложения.

Поскольку он не знает, как работает приложение в целом, то не может отправлять действия напрямую. Вместо этого он, отвечая на события указателя, вызывает функцию обратного вызова, предоставленную кодом, который создал данный компонент. Эта функция обратного вызова и будет обрабатывать специфичные для приложения моменты.

```
const scale = 10;

class PictureCanvas {
  constructor(picture, pointerDown) {
    this.dom = elt("canvas", {
      onmousedown: event => this.mouse(event, pointerDown),
      ontouchstart: event => this.touch(event, pointerDown)
    });
    this.syncState(picture);
  }
  syncState(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  }
}
```

Мы рисуем каждый пиксел в виде квадрата 10 × 10, согласно значению константы `scale`. Чтобы избежать ненужной работы, компонент отслеживает свое текущее изображение и перерисовывает его только тогда, когда `syncState` получает новое изображение.

Наконец, функция, непосредственно выполняющая рисование, устанавливает размер холста на основе масштаба и размера изображения и заполняет его множеством квадратиков, по одному на каждый пиксел.

```
function drawPicture(picture, canvas, scale) {
  canvas.width = picture.width * scale;
  canvas.height = picture.height * scale;
  let cx = canvas.getContext("2d");

  for (let y = 0; y < picture.height; y++) {
    for (let x = 0; x < picture.width; x++) {
      cx.fillStyle = picture.pixel(x, y);
      cx.fillRect(x * scale, y * scale, scale, scale);
    }
  }
}
```

Когда указатель мыши находится на холсте изображения и нажата левая кнопка, компонент выполняет обратный вызов функции `pointerDown`, передавая ему положение пиксела, на котором был выполнен щелчок, в виде координат изображения. Эти данные будут использованы для реализации взаимодействия мыши с изображением. Функция обратного вызова может возвращать другую функцию обратного вызова, которая будет уведомлена о том, что указатель переместился на другой пиксел при нажатой левой кнопке.

```
PictureCanvas.prototype.mouse = function(downEvent, onDown) {
  if (downEvent.button != 0) return;
  let pos = pointerPosition(downEvent, this.dom);
  let onMove = onDown(pos);
  if (!onMove) return;
  let move = moveEvent => {
    if (moveEvent.buttons == 0) {
      this.dom.removeEventListener("mousemove", move);
    } else {
      let newPos = pointerPosition(moveEvent, this.dom);
      if (newPos.x == pos.x && newPos.y == pos.y) return;
      pos = newPos;
      onMove(newPos);
    }
  };
  this.dom.addEventListener("mousemove", move);
};

function pointerPosition(pos, domNode) {
  let rect = domNode.getBoundingClientRect();
  return {x: Math.floor((pos.clientX - rect.left) / scale),
          y: Math.floor((pos.clientY - rect.top) / scale)};
}
```

Поскольку нам известен размер пикселов и мы можем использовать `getBoundingClientRect`, чтобы найти положение холста на экране, можно перейти от координат события мыши (`clientX` и `clientY`) к координатам изображения. Они всегда округляются, поэтому относятся к определенному пикселу.

Следует сделать что-то подобное и с сенсорными событиями, только зарегистрировать обработку других событий и не забыть вызвать функцию `preventDefault` для события `"touchstart"`, чтобы предотвратить панорамирование.

```
PictureCanvas.prototype.touch = function(startEvent,
                                         onDown) {
  let pos = pointerPosition(startEvent.touches[0], this.dom);
```

```
  let onMove = onDown(pos);
  startEvent.preventDefault();
  if (!onMove) return;
  let move = moveEvent => {
    let newPos = pointerPosition(moveEvent.touches[0],
                                 this.dom);
    if (newPos.x == pos.x && newPos.y == pos.y) return;
    pos = newPos;
    onMove(newPos);
  };
  let end = () => {
    this.dom.removeEventListener("touchmove", move);
    this.dom.removeEventListener("touchend", end);
  };
  this.dom.addEventListener("touchmove", move);
  this.dom.addEventListener("touchend", end);
};
```

Для сенсорных событий `clientX` и `clientY` непосредственно для объекта события недоступны, но мы можем использовать координаты первого сенсорного объекта в свойстве `touch`.

Приложение

Чтобы приложение можно было построить по частям, мы реализуем основной компонент в виде оболочки, куда заключен холст изображения, и динамического набора инструментов и элементов управления, которые мы передаем ее конструктору.

Элементы управления — это элементы интерфейса, размещаемые под картинкой. Они будут представлены в виде массива конструкторов компонентов.

Инструменты выполняют такие операции, как рисование пикселов или заполнение области цветом. В приложении набор доступных инструментов отображается в виде поля `<select>`. Инструмент, выбранный в данный момент, определяет, что произойдет, когда пользователь будет взаимодействовать с изображением с помощью указательного устройства. Множество доступных инструментов представлено в виде объекта, в котором имена, содержащиеся в раскрывающемся списке, сопоставляются с функциями, реализующими эти инструменты. Такие функции получают в качестве аргументов позицию на изображении, текущее состояние приложения и функцию `dispatch`. Они могут возвращать функцию — обработчик перемещения,

которая вызывается для новой позиции и текущего состояния при перемещении указателя на другой пиксел.

```
class PixelEditor {
  constructor(state, config) {
    let {tools, controls, dispatch} = config;
    this.state = state;

    this.canvas = new PictureCanvas(state.picture, pos => {
      let tool = tools[this.state.tool];
      let onMove = tool(pos, this.state, dispatch);
      if (onMove) return pos => onMove(pos, this.state);
    });
    this.controls = controls.map(
      Control => new Control(state, config));
    this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                   ...this.controls.reduce(
                   (a, c) => a.concat(" ", c.dom), []));
  }
  syncState(state) {
    this.state = state;
    this.canvas.syncState(state.picture);
    for (let ctrl of this.controls) ctrl.syncState(state);
  }
}
```

Обработчик событий указателя, переданный в `PictureCanvas`, вызывает текущий выбранный инструмент с соответствующими аргументами и, если он возвращает обработчик перемещения, адаптирует его для получения состояния.

Все элементы управления создаются и хранятся в `this.controls`, чтобы их можно было обновлять при изменении состояния приложения. Вызов `reduce` создает пробелы между элементами DOM, соответствующими элементам управления приложением, чтобы они не размещались слишком тесно друг к другу.

Первый элемент управления — это меню выбора инструмента. Он создает элемент `<select>`, в котором каждый вариант выбора соответствует определенному инструменту, и создает обработчик события `"change"`, обновляющий состояние приложения, когда пользователь выбирает другой инструмент.

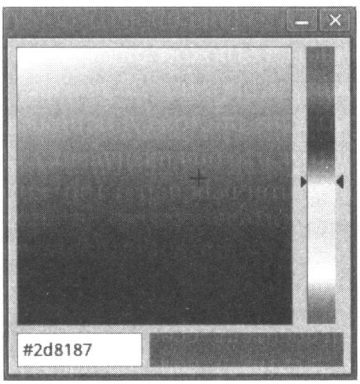
```
class ToolSelect {
  constructor(state, {tools, dispatch}) {
    this.select = elt("select", {
      onchange: () => dispatch({tool: this.select.value})
```

```
    }, ...Object.keys(tools).map(name => elt("option", {
      selected: name == state.tool
    }, name)));
    this.dom = elt("label", null, " 🖌 Инструмент: ", this.select);
  }
  syncState(state) { this.select.value = state.tool; }
}
```

Оборачивая текст метки и поле в элемент `<label>`, мы сообщаем браузеру, что эта метка относится к данному полю, и если, например, щелкнуть на метке, то поле получит фокус.

Нам также нужно иметь возможность изменять цвет, и мы создадим для этого специальный элемент управления. HTML-элемент `<input>` с атрибутом `type`, равным `color`, создает поле формы, специализированное для выбора цвета. Значением такого поля всегда является цветовой код CSS в формате `"#RRGGBB"` (красный, зеленый и синий компоненты, по две цифры на цвет). При взаимодействии пользователя с этим полем браузер покажет интерфейс селектора цвета.

В зависимости от браузера селектор цвета может выглядеть так.



Следующий элемент управления создает такое поле и синхронизирует его со свойством `color` состояния приложения.

```
class ColorSelect {
  constructor(state, {dispatch}) {
    this.input = elt("input", {
      type: "color",
      value: state.color,
      onchange: () => dispatch({color: this.input.value})
```

```
    });
    this.dom = elt("label", null, " 🎨 Цвет: ", this.input);
  }
  syncState(state) { this.input.value = state.color; }
}
```

Инструменты рисования

Прежде чем мы сможем что-нибудь нарисовать, нам нужно реализовать инструменты, которые будут управлять функциональностью событий мыши или сенсорных событий на холсте.

Простейшим инструментом является инструмент рисования. Он изменяет цвет любого пиксела, на котором вы щелкаете или которого касаетесь, на текущий выбранный цвет. Он отправляет действие, обновляющее изображение до версии, в какой выбранному пикселу присвоен текущий цвет.

```
function draw(pos, state, dispatch) {
  function drawPixel({x, y}, state) {
    let drawn = {x, y, color: state.color};
    dispatch({picture: state.picture.draw([drawn])});
  }
  drawPixel(pos, state);
  return drawPixel;
}
```

Эта функция сразу же вызывает функцию `drawPixel`, а затем вызывает ее еще раз для вновь затронутых пикселов, когда пользователь перетаскивает указатель мыши или проводит пальцем по изображению.

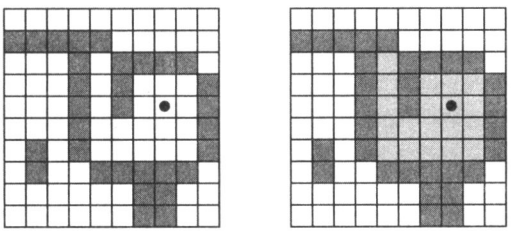
Чтобы рисовать большие фигуры, может быть полезно быстро создавать прямоугольники. Инструмент `rectangle` рисует прямоугольник от точки, с которой началось перетаскивание, до точки, в которую был перемещен указатель.

```
function rectangle(start, state, dispatch) {
  function drawRectangle(pos) {
    let xStart = Math.min(start.x, pos.x);
    let yStart = Math.min(start.y, pos.y);
    let xEnd = Math.max(start.x, pos.x);
    let yEnd = Math.max(start.y, pos.y);
    let drawn = [];
    for (let y = yStart; y <= yEnd; y++) {
```

```
      for (let x = xStart; x <= xEnd; x++) {
        drawn.push({x, y, color: state.color});
      }
    }
    dispatch({picture: state.picture.draw(drawn)});
  }
  drawRectangle(start);
  return drawRectangle;
}
```

Важной деталью в этой реализации является то, что при перетаскивании прямоугольник перерисовывается на изображении из *исходного* состояния. Благодаря этому можно сделать прямоугольник больше или меньше уже при его создании, без промежуточных прямоугольников, засоряющих итоговое изображение. Это одна из причин, почему полезно создавать неизменяемые графические объекты — позже мы увидим и другую причину.

Реализация сплошной заливки несколько сложнее. Это инструмент, который заполняет пиксел, попадающий под указатель, и все смежные пикселы, имеющие тот же цвет. «Смежными» называются пикселы, расположенные непосредственно рядом с текущим по горизонтали или вертикали, но не по диагонали. На следующем рисунке показан набор пикселов, окрашенных при применении инструмента сплошной заливки в отмеченном пикселе.



Интересно, что способ, которым это делается, похож на код поиска пути из главы 7. Если там код просматривал граф, чтобы найти маршрут, то здесь код просматривает сетку, чтобы найти все «смежные» пикселы. В обоих случаях встречаем аналогичную проблему отслеживания разветвленного множества возможных маршрутов.

```
const around = [{dx: -1, dy: 0}, {dx: 1, dy: 0},
                {dx: 0, dy: -1}, {dx: 0, dy: 1}];

function fill({x, y}, state, dispatch) {
  let targetColor = state.picture.pixel(x, y);
```

```
  let drawn = [{x, y, color: state.color}];
  for (let done = 0; done < drawn.length; done++) {
    for (let {dx, dy} of around) {
      let x = drawn[done].x + dx, y = drawn[done].y + dy;
      if (x >= 0 && x < state.picture.width &&
          y >= 0 && y < state.picture.height &&
          state.picture.pixel(x, y) == targetColor &&
          !drawn.some(p => p.x == x && p.y == y)) {
        drawn.push({x, y, color: state.color});
      }
    }
  }
  dispatch({picture: state.picture.draw(drawn)});
}
```

Массив нарисованных пикселов удваивается и образует рабочий список функции. Для каждого очередного пиксела нужно проверить, есть ли у него смежные пикселы с таким же цветом и не были ли они уже закрашены. При добавлении новых пикселов счетчик цикла отстает от длины массива `drawn`. Нужно изучить все оставшиеся пикселы. Когда счетчик станет равным длине массива, неисследованных пикселов не останется и функция завершит работу.

Последний инструмент — это селектор цвета, который позволяет выбрать цвет в изображении, чтобы использовать его в качестве текущего цвета рисунка.

```
function pick(pos, state, dispatch) {
  dispatch({color: state.picture.pixel(pos.x, pos.y)});
}
```

Сохранение и загрузка

После того как мы нарисовали наш шедевр, мы хотим сохранить его на будущее. Нам нужно добавить кнопку загрузки текущего изображения в виде файла изображения. Вот элемент управления, предоставляющий эту кнопку:

```
class SaveButton {
  constructor(state) {
    this.picture = state.picture;
    this.dom = elt("button", {
      onclick: () => this.save()
    }, "💾 Сохранить");
```

```
  }
  save() {
    let canvas = elt("canvas");
    drawPicture(this.picture, canvas, 1);
    let link = elt("a", {
      href: canvas.toDataURL(),
      download: "pixelart.png"
    });
    document.body.appendChild(link);
    link.click();
    link.remove();
  }
  syncState(state) { this.picture = state.picture; }
}
```

Компонент отслеживает текущее изображение, чтобы иметь к нему доступ при сохранении. Для того чтобы создать файл изображения, он использует элемент `<canvas>`, на котором рисует картинку (в масштабе пиксел на пиксел).

Метод `toDataURL` для элемента холста создает URL, который начинается с `data:`. В отличие от URL, начинающихся с `http:` и `https:`, URL-адреса данных содержат весь ресурс в виде URL. Они обычно очень длинные, но зато позволяют создавать рабочие ссылки на произвольные картинки прямо в браузере.

Чтобы браузер действительно загружал картинку, нужно создать элемент ссылки, который указывает на этот URL и имеет атрибут `download`. Если щелкнуть на такой ссылке, браузер отображает диалоговое окно сохранения файла. Мы добавим данную ссылку в документ, имитируем щелчок на нем и снова ее удалим.

Браузерные технологии позволяют желать многое, но иногда способы сделать это бывают довольно странными.

И это еще не все. Мы также хотим иметь возможность загружать в наше приложение существующие файлы с изображениями. Для этого мы определим еще один компонент кнопки.

```
class LoadButton {
  constructor(_, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => startLoad(dispatch)
    }, "■ Загрузить");
  }
```

```
  syncState() {}
}

function startLoad(dispatch) {
  let input = elt("input", {
    type: "file",
    onchange: () => finishLoad(input.files[0], dispatch)
  });
  document.body.appendChild(input);
  input.click();
  input.remove();
}
```

Чтобы получить доступ к файлу на компьютере пользователя, нужно, чтобы пользователь выбрал файл в поле его ввода. Но мы не хотим, чтобы кнопка загрузки выглядела как поле ввода файла, поэтому создаем поле ввода файла при щелчке на кнопке, а затем делаем вид, что пользователь щелкнул именно на нужном поле ввода файла.

После того как пользователь выбрал файл, можно использовать `FileReader`, чтобы получить доступ к его содержимому, снова в виде URL-адреса данных. Этот URL можно использовать для создания элемента `<img>`, но поскольку мы не можем получить прямой доступ к пикселам в таком изображении, то мы не можем и создать из него объект `Picture`.

```
function finishLoad(file, dispatch) {
  if (file == null) return;
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    let image = elt("img", {
      onload: () => dispatch({
        picture: pictureFromImage(image)
      }),
      src: reader.result
    });
  });
  reader.readAsDataURL(file);
}
```

Чтобы получить доступ к пикселам, нам нужно сначала нарисовать изображение в элементе `<canvas>`. У контекста холста есть метод `getImageData`, который позволяет сценарию считывать его пикселы. Поэтому, как только изображение окажется на холсте, мы сможем получить к нему доступ и создать объект `Picture`.

```
function pictureFromImage(image) {
  let width = Math.min(100, image.width);
  let height = Math.min(100, image.height);
  let canvas = elt("canvas", {width, height});
  let cx = canvas.getContext("2d");
  cx.drawImage(image, 0, 0);
  let pixels = [];
  let {data} = cx.getImageData(0, 0, width, height);

  function hex(n) {
    return n.toString(16).padStart(2, "0");
  }
  for (let i = 0; i < data.length; i += 4) {
    let [r, g, b] = data.slice(i, i + 3);
    pixels.push("#" + hex(r) + hex(g) + hex(b));
  }
  return new Picture(width, height, pixels);
}
```

Мы ограничим размер изображения до квадрата размером 100 × 100 пикселов, поскольку все, что больше этого размера, будет выглядеть на нашем дисплее действительно *огромным* и может замедлить работу интерфейса.

Свойство `data` объекта, возвращаемого `getImageData`, представляет собой массив компонентов цвета. Для каждого пиксела в прямоугольнике, заданном аргументами, этот массив содержит четыре значения, которые представляют красный, зеленый, синий и *альфа*-компоненты цвета пиксела в виде чисел от 0 до 255. Альфа-часть соответствует степени непрозрачности: когда она равна нулю, пиксел полностью прозрачен, а когда она равна 255, то полностью непрозрачен. Для наших целей мы это можем игнорировать.

Два шестнадцатеричных числа на компонент, используемые нами при обозначении цвета, точно соответствуют диапазону от 0 до 255 — два шестнадцатеричных числа позволяют выражать $16^2 = 256$ различных чисел. В метод преобразования чисел в строку `toString` может быть передано основание системы счисления, так что `n.toString(16)` формирует строковое представление числа по основанию 16. Нам нужно убедиться, что каждое число занимает две цифры, поэтому вспомогательная функция `hex` вызывает `padStart`, чтобы при необходимости добавить ведущий ноль.

Теперь мы можем загружать и сохранять картинки! Осталась только одна функция — и все будет готово.

История действий

Процесс редактирования наполовину состоит в том, чтобы делать небольшие ошибки и исправлять их. Таким образом, важной частью программы рисования является история действий.

Чтобы иметь возможность отменить изменения, нам нужно сохранять предыдущие версии изображения. Это легко сделать, поскольку они представляют собой неизменяемые значения. Но тогда потребуется создать дополнительное поле в состоянии приложения.

Чтобы сохранять предыдущие версии изображения, мы добавим массив `done`. Поддержание этого свойства требует более сложной функции изменения состояния, которая добавляет изображения в массив.

Но мы не хотим сохранять *каждое* изменение, лишь сделанные в течение определенного промежутка времени. Для этого нам понадобится второе свойство, `doneAt`, отслеживающее время, когда мы в последний раз сохраняли изображение в истории.

```
function historyUpdateState(state, action) {
  if (action.undo == true) {
    if (state.done.length == 0) return state;
    return Object.assign({}, state, {
      picture: state.done[0],
      done: state.done.slice(1),
      doneAt: 0
    });
  } else if (action.picture &&
             state.doneAt < Date.now() - 1000) {
    return Object.assign({}, state, action, {
      done: [state.picture, ...state.done],
      doneAt: Date.now()
    });
  } else {
    return Object.assign({}, state, action);
  }
}
```

Если действие является отменой операции, то функция берет самое последнее изображение из истории и делает его текущим. Она присваивает `doneAt` нулевое значение, так что при следующем изменении изображение гарантированно сохранится в истории под большим номером, что позволит при желании вернуться к нему в следующий раз.

В противном случае, если действие содержит новое изображение и в последний раз мы сохраняли что-либо более секунды (1000 миллисекунд) назад, свойства `done` и `doneAt` изменяются для сохранения предыдущего изображения.

Компонент кнопки отмены мало что делает. Он только отправляет действие отмены при нажатии и является неактивным, если отменять нечего.

```
class UndoButton {
  constructor(state, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => dispatch({undo: true}),
      disabled: state.done.length == 0
    }, " ⮪ Отменить");
  }
  syncState(state) {
    this.dom.disabled = state.done.length == 0;
  }
}
```

Давайте порисуем

Чтобы настроить приложение, нам нужно создать состояние, набор инструментов, набор элементов управления и функцию отправки. Для создания основного компонента мы можем передать все это конструктору `PixelEditor`. Поскольку в наших упражнениях нам понадобится создать несколько редакторов, мы сначала определим некоторые привязки.

```
const startState = {
  tool: "draw",
  color: "#000000",
  picture: Picture.empty(60, 30, "#f0f0f0"),
  done: [],
  doneAt: 0
};

const baseTools = {draw, fill, rectangle, pick};

const baseControls = [
  ToolSelect, ColorSelect, SaveButton, LoadButton, UndoButton
];

function startPixelEditor({state = startState,
                           tools = baseTools,
                           controls = baseControls}) {
```

```
  let app = new PixelEditor(state, {
    tools,
    controls,
    dispatch(action) {
      state = historyUpdateState(state, action);
      app.syncState(state);
    }
  });
  return app.dom;
}
```

При деструктурировании объекта или массива можно задействовать знак =, стоящий после имени привязки, чтобы присвоить привязке значение по умолчанию. Это значение будет применяться, если соответствующее свойство отсутствует или содержит значение `undefined`. В функции `startPixelEditor` это используется для того, чтобы принять объект с рядом необязательных свойств в качестве аргумента. Например, если не предоставить свойство `tools`, то значение `tools` будет привязано к `baseTools`.

Ниже показано, как получить реальный редактор на экране:

```
<div></div>
<script>
  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

Почему это так сложно?

Браузерные технологии поразительны. Они предоставляют мощный набор блоков для построения интерфейса, способы их стилизации и управления ими, а также инструменты для проверки и отладки приложений. Программное обеспечение, разработанное для браузеров, может работать практически на любом компьютере или телефоне в любой точке мира.

Тем не менее временами браузерные технологии просто смешны. Чтобы их освоить, требуется изучить множество глупых уловок и не вполне понятных фактов, а предлагаемая по умолчанию модель программирования настолько проблематична, что большинство программистов предпочитают делить ее на несколько уровней абстракции, вместо того чтобы иметь с ней дело напрямую.

И хотя со временем ситуация определенно улучшается, в основном это происходит в форме добавления дополнительных элементов для устранения недостатков, что создает еще больше сложностей. Функциональность, используемая на миллионах сайтов, не может быть заменена. И даже если бы это было возможно, было бы трудно решить, чем именно ее заменить.

Технология никогда не существует в вакууме — мы ограничены инструментарием, а также социальными, экономическими и историческими факторами, которые данную технологию породили. Временами это раздражает, но, как правило, будет более продуктивным постараться лучше понять, как работает *существующая* техническая реальность и почему она такова, вместо того чтобы злиться на нее или ждать, пока появится другая реальность.

Новые абстракции *действительно могут* быть полезны. Компонентная модель и соглашение о потоках данных, которые я использовал в этой главе, — грубая форма таких абстракций. Как уже упоминалось, существуют библиотеки, пытающиеся сделать программирование пользовательского интерфейса более приятным. На момент написания этой книги популярными вариантами были React и Angular, но существует целая индустрия таких систем. Если вы заинтересованы в программировании веб-приложений, я рекомендую изучить некоторые из них, чтобы понять, как они работают и какие преимущества предоставляют.

Упражнения

В нашей программе все еще остается простор для совершенствования. В порядке упражнений добавим в нее еще несколько функций.

Клавиатурные привязки

Добавьте в приложение сочетания клавиш. Первая буква названия инструмента выбирает инструмент, а Control+Z или Command+Z активируют отмену операции.

Для этого измените компонент `PixelEditor`. Добавьте к обертывающему элементу `<div>` свойство `tabIndex`, равное `0`, чтобы этот элемент мог получать фокус клавиатуры. Обратите внимание, что *свойство*, соответствующее *атрибуту* `tabindex`, называется `tabIndex`, с большой буквой `I`, а наша функция `elt` ожидает именно имена свойств. Зарегистрируйте обработчики

событий клавиатуры непосредственно для данного элемента. Это означает, что для взаимодействия с редактором с помощью клавиатуры сначала нужно щелкнуть на указанном элементе, коснуться его или перейти к нему с помощью табуляции.

Помните, что события клавиатуры имеют свойства `ctrlKey` и `metaKey` (для клавиши Command на Mac), и эти свойства можно использовать, чтобы увидеть, удерживаются ли данные клавиши нажатыми.

Эффективное рисование

В процессе рисования большая часть работы, которую выполняет приложение, происходит в `drawPicture`. Создание нового состояния и обновление остальной части DOM не требует много ресурсов, но перерисовка всех пикселов на холсте — довольно трудоемкая работа.

Найдите способ ускорить метод `syncState` в `PictureCanvas`, перерисовывая только те пикселы, которые действительно изменились.

Помните, что `drawPicture` также используется кнопкой сохранения, поэтому, если вы будете изменять эту функцию, убедитесь, что изменения не нарушат старый вариант применения, либо создайте новую версию с другим именем.

Обратите также внимание, что изменение размера элемента `<canvas>` путем изменения значений его свойств `width` или `height` очищает этот элемент, снова делая его полностью прозрачным.

Круги

Создайте инструмент под названием `circle`, при перетаскивании которого рисуется круг, заполненный цветом. Центр круга находится в точке, откуда начинается перетаскивание или в которой произошло касание, а радиус круга определяется расстоянием перетаскивания.

Правильные линии

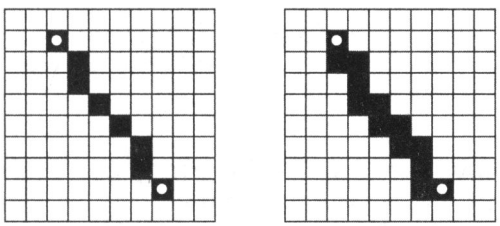
Это более сложное упражнение, чем два предыдущих, и оно потребует от вас разработки решения для нетривиальной проблемы. Убедитесь, что у вас есть достаточно времени и терпения, прежде чем приступить к работе над данным упражнением, и не спешите разочароваться от первых же неудач.

В большинстве браузеров, если выбрать инструмент `draw` и быстро перетащить указатель по изображению, вы не получаете замкнутую линию. Скорее вы получите точки с промежутками между ними, потому что события `"mousemove"` или `"touchmove"` не срабатывают достаточно быстро, чтобы отметить каждый пиксел.

Усовершенствуйте инструмент `draw` так, чтобы он рисовал сплошную линию. Это означает, что вам нужно заставить функцию обработчика движения запоминать предыдущую позицию и соединять ее с текущей.

Для этого, поскольку пикселы могут находиться на произвольном расстоянии друг от друга, вам нужно написать обобщенную функцию рисования линий.

Линия между двумя пикселами представляет собой цепочку связанных между собой пикселов, максимально прямую, идущую от начала до конца. Пикселы, смежные по диагонали, считаются связанными. Таким образом, косая линия должна выглядеть так, как показано на картинке слева, а не на картинке справа.



Наконец, поскольку у нас есть код, который рисует линию между двумя произвольными точками, мы могли бы также использовать его для определения инструмента `line`, который рисует прямую линию между начальной и конечной точками перетаскивания.

20 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*

До сих пор мы использовали язык JavaScript в одной среде: браузере. В этой и следующей главах мы кратко рассмотрим Node.js — программу, которая позволит вам применить навыки программирования на JavaScript вне браузера. С его помощью можно создавать что угодно: от небольших утилит командной строки до HTTP-серверов, обеспечивающих работу динамических сайтов.

Цель этих глав — познакомить вас с основнымм концепциями, которые используются в Node.js, и дать достаточно информации для написания полезных программ на этой платформе. Я не буду даже пытаться представить полное, всеобъемлющее описание того, как с ней работать.

Если вы хотите пойти дальше и самостоятельно запускать код, приведенный в этой главе, вам понадобится установить Node.js версии 10.1 или выше. Для этого перейдите на страницу https://nodejs.org и следуйте инструкциям по установке Node.js для вашей операционной системы. Там же вы найдете дополнительную документацию по Node.js.

Основы

Одной из наиболее сложных проблем для систем записи, которые обмениваются данными по сети, является управление вводом и выводом, то есть чтение данных из сети и с жесткого диска и запись данных в сеть и на жесткий диск. Перемещение их занимает время, а от грамотного планирования зависит, как быстро система сможет реагировать на запросы пользователя или сети.

В таких программах часто помогает асинхронное программирование. Оно позволяет программе отправлять и получать данные с нескольких устройств одновременно и без сложного управления потоками и синхронизации.

Система Node изначально была задумана с целью сделать асинхронное программирование простым и удобным. JavaScript хорошо подходит для такой системы, как Node. Это один из немногих языков программирования, у которого нет встроенного способа ввода и вывода. Таким образом, JavaScript вполне соответствует довольно эксцентричному подходу Node к вводу и выводу, так как не приводит к появлению двух несовместимых интерфейсов. В 2009 году, когда разрабатывалась система Node, уже существовало программирование на основе обратного вызова в браузерах, поэтому сообщество, сформировавшееся вокруг языка, привыкло к стилю асинхронного программирования.

Команда node

После установки Node.js в системе появляется программа под названием `node`, которая используется для запуска файлов JavaScript. Предположим, у нас есть файл `hello.js`, содержащий следующий код:

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

Тогда мы можем выполнить эту программу, запустив `node` из командной строки следующим образом:

```
$ node hello.js
Hello world
```

Метод `console.log` в Node делает примерно то же, что и в браузере: печатает фрагмент текста. Но в Node текст попадет не в JavaScript-консоль браузера,

а в стандартный поток вывода процесса. При запуске `node` из командной строки это означает, что вы увидите выведенные значения в терминале.

Если запустить `node` без указания имени файла, то появится подсказка, после которой можно ввести код JavaScript и сразу увидеть результат.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

Привязка `process`, как и привязка `console`, доступна в Node глобально. Привязка `process` предоставляет различные способы проверки и управления текущей программой. Метод `exit` завершает процесс, ему можно передать код состояния выхода, который будет сообщать программе, запустившей `node` (в данном случае самой оболочке командной строки), завершилась программа успешно (нулевой код) или с ошибкой (любой другой код).

Для того чтобы узнать, какие аргументы командной строки были переданы сценарию, можно прочитать свойство `process.argv`, которое представляет собой массив строк. Обратите внимание, что оно также включает в себя имя команды `node` и имя сценария, поэтому настоящие аргументы начинаются с индекса 2. Если сценарий `showargv.js` содержит инструкцию `console.log` (`process.argv`), то можете запустить его следующим образом:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

В среде Node присутствуют все стандартные глобальные привязки JavaScript, такие как `Array`, `Math` и `JSON`. Однако к связанным с браузером функциям, таким как `document` или `prompt`, это не относится.

Модули

Помимо упомянутых привязок, таких как `console` и `process`, Node добавляет в глобальную область еще несколько дополнительных привязок. Чтобы получить доступ к встроенным функциям, нужно запросить их в системе модулей.

Система модулей CommonJS, основанная на функции `require`, была описана в главе 10. Эта система встроена в Node и используется для загрузки чего угодно: от встроенных модулей до загруженных пакетов и файлов, которые являются частью вашей собственной программы.

При вызове `require` Node должен преобразовать заданную строку в фактический файл, который можно загрузить. Имена путей, начинающиеся с `/`, `./` или `../`, преобразуются в полные пути относительно пути текущего модуля, где `.` обозначает текущий каталог, `../` — каталог, расположенный на один уровень выше, а `/` — корневой каталог файловой системы. Например, если запросить `"./graph"` из файла `/tmp/robot/robot.js`, Node попытается загрузить файл `/tmp/robot/graph.js`.

Расширение `.js` можно опустить, Node самостоятельно добавит его, если такой файл существует. Если требуемый путь означает каталог, то Node попытается загрузить из него файл с именем `index.js`.

Если `require` передана строка, которая не выглядит как относительный или абсолютный путь, предполагается, что она ссылается на встроенный модуль либо на модуль, установленный в каталоге `node_modules`. Например, `require("fs")` загрузит модуль встроенной файловой системы Node. А `require("robot")` может попытаться загрузить библиотеку, расположенную в каталоге `node_modules/robot/`. Распространенным способом установки таких библиотек является использование менеджера пакетов NPM, к которому мы еще вернемся.

Создадим небольшой проект, состоящий из двух файлов. Первый из них, называемый `main.js`, определяет сценарий, который можно вызывать из командной строки для обращения строки.

```
const {reverse} = require("./reverse");

// Под индексом 2 содержится первый реальный аргумент командной строки
let argument = process.argv[2];

console.log(reverse(argument));
```

Файл `reverse.js` определяет библиотеку для обращения строк, которая может использоваться как в этом инструменте командной строки, так и в других сценариях, кому необходим прямой доступ к функции обращения строк.

```
exports.reverse = function(string) {
  return Array.from(string).reverse().join("");
};
```

Помните, что при добавлении свойств к `exports` они добавляются в интерфейс модуля. Поскольку Node.js обрабатывает файлы как модули CommonJS, в `main.js` можно использовать экспортированную функцию `reverse` из `reverse.js`.

Теперь мы можем вызвать наш инструмент следующим образом:

```
$ node main.js JavaScript
tpircSavaJ
```

Установка с помощью NPM

Менеджер пакетов NPM, представленный в главе 10, является онлайн-хранилищем модулей JavaScript, многие из которых специально написаны для Node. После установки Node на компьютере становится доступной команда `npm` — ее можно использовать для взаимодействия с этим хранилищем.

Основное назначение NPM — загрузка пакетов. В главе 10 нам уже встречался пакет `ini`. Мы можем применять NPM для загрузки и установки этого пакета на компьютере.

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
         open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

После запуска `npm install` NPM создает каталог с именем `node_modules`. Внутри него находится каталог `ini`, содержащий библиотеку. Вы можете открыть его и посмотреть на код. При вызове `require("ini")` эта библиотека загружается, и можно вызвать ее свойство `parse` для разбора файла конфигурации.

По умолчанию NPM устанавливает пакеты не в каком-либо едином месте, а в текущем каталоге. Если вы привыкли к другим менеджерам пакетов, это может показаться необычным, но здесь есть свои преимущества — каждое приложение получает полный контроль над пакетами, которые оно

устанавливает, упрощается управление версиями и очистка при удалении приложения.

Файлы пакетов

В примере с `npm install` выводится предупреждение о том, что файл `package.json` не существует. Рекомендуется создавать такой файл для каждого проекта либо вручную, либо с помощью `npm init`. В этом файле содержится информация о проекте, такая как имя и версия, а также перечисляются его зависимости.

Симуляция робота из главы 7, представленная в виде модулей в упражнении к главе 10, может иметь, например, такой файл `package.json`:

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Моделирование робота, доставляющего посылки",
  "version": "1.0.0",
  "main": "run.js",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

Если запустить `npm install` без указания имени устанавливаемого пакета, то NPM установит зависимости, перечисленные в `package.json`. При установке определенного пакета, который не указан в качестве зависимости, NPM добавит его в пакет `.json`.

Версии

В файле `package.json` указана не только версия самой программы, но и версии ее зависимостей. Версии — это способ справиться с тем фактом, что каждый пакет развивается сам по себе, а код, написанный для работы с пакетом в том виде, в каком он существовал в какой-то определенный момент, может не работать с более поздней модифицированной версией пакета.

NPM требует, чтобы пакеты соответствовали схеме, называемой *семантической версификацией*, при которой в номере версии зашифрована

информация о том, какие версии являются *совместимыми* (не ломают старый интерфейс). Семантическая версия состоит из трех чисел, разделенных точками, например `2.3.0`. Каждый раз, когда в пакет добавляются новые функции, среднее число увеличивается на единицу. Каждый раз, когда нарушается совместимость, так что существующий код, использующий пакет, может уже не работать с новой версией, первое число должно увеличиваться на единицу.

Знак «крышка» (`^`) перед номером версии зависимости в `package.json` указывает, что может быть установлена любая версия, совместимая с данным номером. Так, например, `"^2.3.0"` означает, что любая версия, больше или равная 2.3.0 и меньше чем 3.0.0, является допустимой.

Команда `npm` также используется для публикации новых пакетов или новых версий пакетов. Если запустить `npm publish` в каталоге, где есть файл `package.json`, то в реестре будет опубликован пакет с именем и версией, указанными в JSON-файле. Публиковать пакеты в NPM может любой желающий, но только под именем пакета, которое еще не используется. Было бы страшновато, если бы случайные люди могли обновить существующие пакеты.

Поскольку программа `npm` — это часть программного обеспечения, взаимодействующая с открытой системой — реестром пакетов, — она не делает ничего уникального. Другая программа, `yarn`, которую можно установить из реестра NPM, выполняет ту же роль, что и `npm`, используя несколько другие интерфейс и стратегию установки.

В этой книге мы не будем углубляться в детали применения NPM. Более подробную документацию и описание способов поиска пакетов вы найдете по адресу `https://npmjs.org`.

Модуль файловой системы

Одним из наиболее часто используемых встроенных модулей в Node является модуль `fs`, название которого расшифровывается как *file system* — файловая система. Этот модуль экспортирует функции для работы с файлами и каталогами.

Например, функция под названием `readFile` читает файл и затем использует функцию обратного вызова для его содержимого.

```
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
  if (error) throw error;
  console.log("Содержимое файла:", text);
});
```

Второй аргумент `readFile` определяет *кодировку символов*, используемую для декодирования файла в строку. Существует несколько способов преобразования текста в двоичные данные, но в большинстве современных систем применяется UTF-8. Поэтому, если у вас нет причин предполагать, что используется другая кодировка, передайте при чтении текстового файла значение `"utf8"`. Если не указать кодировку, то Node предположит, что вам нужны двоичные данные, и передаст объект `Buffer` вместо строки. Это массивоподобный объект, который содержит числа, представляющие байты файла (восьмибитные порции данных).

```
const {readFile} = require("fs");
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("Файл содержит", buffer.length, "байт.",
              "Первый байт:", buffer[0]);
});
```

Для записи файла на диск используется аналогичная функция `writeFile`.

```
const {writeFile} = require("fs");
writeFile("graffiti.txt", "Здесь был Node", err => {
  if (err) console.log(`Не удалось записать файл: ${err}`);
  else console.log("Файл записан.");
});
```

В данном случае не было необходимости указывать кодировку — `writeFile` предполагает, что если для записи передана строка, а не объект `Buffer`, то ее следует записать в виде текста с использованием кодировки символов по умолчанию, то есть UTF-8.

Модуль `fs` содержит много других полезных функций: `readdir` возвращает файлы каталога, представленные в виде массива строк; `stat` извлекает информацию о файле; `rename` переименовывает файл; `unlink` удаляет файл и т. д. Более подробную информацию вы найдете в документации на сайте https://nodejs.org.

Большинство из этих функций принимают в качестве последнего параметра имя функции обратного вызова, которую они вызывают в случае

либо ошибки (первый аргумент), либо успешного завершения (второй аргумент).

Как мы видели в главе 11, у этого стиля программирования есть свои недостатки. Самый большой из них заключается в том, что обработка ошибок становится многословной и подвержена ошибкам.

Несмотря на то что промисы уже некоторое время являются частью JavaScript, на момент написания этой книги их интеграция в Node.js все еще находилась на стадии разработки. Существует объект `promises`, экспортируемый из пакета `fs`, начиная с версии 10.1, который содержит большинство тех же функций, что и `fs`, но использует промисы вместо функций обратного вызова.

```
const {readFile} = require("fs").promises;
readFile("file.txt", "utf8")
  .then(text => console.log("Содержимое файла:", text));
```

Иногда асинхронность не только не нужна, но даже мешает. У многих функций `fs` есть синхронный вариант, который имеет то же имя, но с добавлением `Sync` в конце. Например, синхронная версия `readFile` называется `readFileSync`.

```
const {readFileSync} = require("fs");
console.log("Содержимое файла:",
            readFileSync("file.txt", "utf8"));
```

Обратите внимание, что во время выполнения такой синхронной операции программа полностью останавливается. Если она должна отвечать на запросы пользователя или других компьютеров сети, «застревание» на синхронной операции может вызвать раздражающие задержки.

Модуль HTTP

Еще один из основных модулей называется `http`. Он предоставляет функциональность для запуска HTTP-серверов и выполнения HTTP-запросов.

Вот все, что нужно для запуска HTTP-сервера:

```
const {createServer} = require("http");
let server = createServer((request, response) => {
```

```
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Привет!</h1>
    <p>Вы искали <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Начинаю слушать! (port 8000)");
```

Если вы запустите этот сценарий на своем компьютере, то можете указать свой браузер по адресу http://localhost:8000/hello, чтобы направить запрос на ваш сервер. В ответ вы получите небольшую HTML-страницу.

Функция, переданная в `createServer` в качестве аргумента, вызывается каждый раз, когда клиент подключается к серверу. Привязки `request` и `response` являются объектами, представляющими входящие и исходящие данные. Первая из них содержит информацию о запросе, например свойство `url`, которое сообщает, по какому URL был сделан запрос.

Поэтому, когда вы открываете данную страницу в своем браузере, она отправляет запрос на ваш собственный компьютер. В результате запускается функция сервера и отправляет обратно ответ, который затем выводится в браузере.

Чтобы отправить что-то обратно, нужно вызвать методы объекта `response`. Первый из них, `writeHead`, записывает заголовки ответа (см. главу 18). Ему передается код состояния (в данном случае 200, что означает OK) и объект, который содержит значения заголовка. В этом примере создается заголовок `Content-Type`, информирующий клиента о том, что обратно отправляется HTML-документ.

Затем с помощью `response.write` отправляется фактическое тело ответа (сам документ). Этот метод разрешается вызывать несколько раз, если нужно отправить ответ по частям, например, для потоковой передачи данных клиенту по мере их доступности. Наконец, `response.end` сигнализирует об окончании ответа.

При вызове `server.listen` сервер переходит к ожиданию подключения через порт 8000. Вот почему нужно подключиться к localhost:8000, чтобы общаться с этим сервером, а не просто к localhost, который будет использовать по умолчанию порт 80.

При запуске этого сценария процесс просто переходит в режим ожидания. Когда сценарий прослушивает события — в данном случае сетевые подключения, — `node` не будет автоматически завершать работу, когда достигнет конца сценария. Чтобы закрыть его, нажмите CONTROL+C.

Реальный веб-сервер обычно выполняет больше работы, чем в примере, — он обращает внимание на метод запроса (свойство `method`), чтобы увидеть, какое действие пытается выполнить клиент, и просматривает URL-адрес запроса, чтобы выяснить, для какого ресурса это действие должно быть выполнено. Позже в этой главе мы увидим более расширенный вариант сервера.

Чтобы действовать как HTTP-*клиент*, мы можем использовать функцию `request` из модуля `http`.

```
const {request} = require("http");
let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Сервер ответил с кодом состояния",
              response.statusCode);
});
requestStream.end();
```

Первый аргумент функции `request` содержит конфигурацию запроса, сообщая Node, с каким сервером следует общаться, какой путь запрашивать с него, какой метод использовать и т. д. Второй аргумент — это функция, которая должна вызываться при поступлении ответа. Ей передается объект, позволяющий проверять ответ, например, чтобы выяснить его код состояния.

Как и объект `response`, который мы видели на сервере, объект, возвращаемый `request`, позволяет передавать данные в запрос методом `write` и завершать запрос методом `end`. В этом примере не используется метод `write`, поскольку запросы `GET` не должны содержать данные в теле запроса.

В модуле `https` есть аналогичная функция `request`, которую можно использовать для отправки запросов на URL-адреса `https:`.

Запросы, созданные с помощью исходных функций Node, получаются довольно многословными. На NPM есть гораздо более удобные пакеты-

обертки. Например, `node-fetch` предоставляет интерфейс `fetch` на основе промисов, знакомый нам по браузерному программированию.

Потоки

До сих пор в примерах HTTP нам встретились два экземпляра доступных для записи потоков, а именно: объект ответа, в который сервер может записывать данные, и объект запроса, возвращаемый методом `request`.

Записываемые потоки являются широко используемой концепцией Node. У таких объектов есть метод `write`, которому можно передать строку или объект `Buffer` для записи чего-либо в поток. Их метод `end` закрывает поток и также может принимать значение для записи в поток перед закрытием. В оба этих метода также можно передать в качестве дополнительного аргумента функцию обратного вызова, которую они будут вызывать после завершения записи или закрытия потока.

Можно создать доступный для записи поток, который указывает на файл с помощью функции `createWriteStream` из модуля `fs`. Затем можно использовать метод `write` для результирующего объекта, чтобы записать файл по кускам, а не за один раз, как в `writeFile`.

Читаемые потоки немного сложнее. И привязка `request`, которая была передана обратному вызову HTTP-сервера, и привязка `response`, переданная обратному вызову HTTP-клиента, являются читаемыми потоками — сервер читает запросы, а затем записывает ответы, тогда как клиент сначала записывает запрос, а затем читает ответ. Чтение из потока выполняется с использованием обработчиков событий, а не с помощью методов.

У объектов, которые генерируют события в Node, есть метод `on`, аналогичный методу `addEventListener` в браузерах. Этому методу передаются имя события и функция, после чего метод зарегистрирует эту функцию, чтобы вызывать ее всякий раз, когда происходит данное событие.

Для читаемых потоков определены события `"data"` и `"end"`. Первое из них запускается каждый раз, когда поступают данные, а второе — когда завершается поток. Эта модель лучше всего подходит для *потоковой передачи* данных, чтобы немедленно обработать их, даже если весь документ еще недоступен. Файл может быть прочитан как читаемый поток с помощью функции `createReadStream` из `fs`.

Следующий код создает сервер, читающий тела запросов и возвращающий их клиенту в виде текста, все буквы которого заменены прописными:

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

Значение `chunk`, передаваемое обработчику данных, имеет двоичный тип `Buffer`. С помощью метода `toString` его можно преобразовать в строку, расшифровав байты как символы в кодировке UTF-8.

Следующий фрагмент кода, выполняемый при активном сервере преобразования букв в прописные, отправляет запрос на этот сервер и записывает полученный ответ:

```
const {request} = require("http");
request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```

В этом примере текст записывается не в `console.log`, как обычно, а в `process.stdout` (стандартный вывод процесса, который является записываемым потоком). Мы не можем использовать здесь `console.log`, так как этот метод после каждого выведенного фрагмента текста вставляет дополнительный символ новой строки, что в данном случае не подходит, так как ответ может состоять из нескольких фрагментов.

Файловый сервер

Объединим наши новые знания о HTTP-серверах и работе с файловой системой, чтобы навести мосты между ними и создать HTTP-сервер, который обеспечивает удаленный доступ к файловой системе. Такой сервер

обеспечивает все варианты использования — он позволяет веб-приложениям сохранять данные и обмениваться ими или же может предоставить группе людей совместный доступ к нескольким файлам.

Когда мы рассматриваем файлы как HTTP-ресурсы, мы можем использовать HTTP-методы `GET`, `PUT` и `DELETE` для чтения, записи и удаления файлов соответственно. Мы будем интерпретировать путь в запросе как путь к файлу, на который ссылается запрос.

Очевидно, мы не хотим предоставлять доступ ко всей файловой системе, поэтому мы будем интерпретировать эти пути как начинающиеся с рабочего каталога сервера, то есть с каталога, в котором он был запущен. Если я запускаю сервер из каталога `/tmp/public/` (или `C:\tmp\public\` в Windows), то запрос для `/file.txt` должен ссылаться на `/tmp/public/file.txt` (или `C:\tmp\public\file.txt`).

Мы будем строить программу по частям, используя объект, называемый `methods`, для хранения функций, которые обрабатывают различные HTTP-методы. Обработчики методов — это асинхронные функции, получающие объект запроса в качестве аргумента и возвращающие промис, который разрешается в объект, описывающий ответ.

```
const {createServer} = require("http");

const methods = Object.create(null);

createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
       response.writeHead(status, {"Content-Type": type});
       if (body && body.pipe) body.pipe(response);
       else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
```

```
      body: `Метод ${request.method} недопустим.`
    };
}
```

Этот код запускает сервер, который просто возвращает код ошибки 405, то есть код, указывающий на то, что сервер отказывается обрабатывать данный метод.

Когда промис обработчика запроса отклоняется, вызов `catch` переводит ошибку в объект ответа, если это еще не сделано, так что сервер может отправить ответ об ошибке, который сообщает клиенту, что запрос не был обработан.

Поле `status` в описании ответа может быть опущено, в этом случае оно по умолчанию равно 200 (OK). Тип содержимого в свойстве `type` также может быть пропущен, и тогда ответ считается простым текстом.

Если значение `body` является читаемым потоком, то у него есть метод `pipe`, который используется для пересылки всего контента из читаемого потока в поток с возможностью записи. В противном случае предполагается, что это либо `null` (тела нет), либо строка, либо буфер, и он передается непосредственно в метод ответа `end`.

Чтобы выяснить, какой путь к файлу соответствует URL-адресу запроса, функция `urlPath` использует встроенный модуль Node для анализа URL-адреса. Этот метод принимает путь, что-то вроде `"/file.txt"`, декодирует его, чтобы избавиться от управляющих кодов вроде `%20`, и преобразует его в соответствии с рабочим каталогом программы.

```
const {parse} = require("url");
const {resolve, sep} = require("path");

const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = parse(url);
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

Как только программа будет настроена для приема сетевых запросов, нужно начинать беспокоиться о безопасности. В данном случае, если мы не будем осторожны, вполне может оказаться, что вся файловая система станет доступной по сети.

Пути к файлам в Node представлены в виде строк. Чтобы перейти от такой строки к настоящему файлу, нужно выполнить набор нетривиальных операций. Пути могут, например, включать в себя `../` для ссылки на родительский каталог. Таким образом, одним из очевидных источников проблем могут стать запросы на пути типа `/../secret_file`.

Чтобы избежать подобных проблем, `urlPath` использует функцию `resolve` из модуля `path`, которая преобразует относительные пути в абсолютные. Затем она проверяет, принадлежит ли результат рабочему каталогу. Для поиска этого рабочего каталога можно воспользоваться функцией `process.cwd` (где `cwd` расшифровывается как *current working directory* — «текущий рабочий каталог»). Привязка `sep` из пакета `path` является системным разделителем пути — обратная косая черта для Windows и прямая косая черта для большинства других систем. Если путь не начинается с базового каталога, то функция генерирует объект сообщения об ошибке, используя код состояния HTTP, указывающий на то, что доступ к ресурсу запрещен.

Мы настроим метод `GET` так, чтобы при чтении каталога он возвращал список файлов, а при чтении обычного файла — содержимое этого файла.

Одна из сложностей заключается в том, какой тип заголовка `Content-Type` следует установить при возврате содержимого файла. Поскольку эти файлы могут быть любыми, сервер не может просто возвращать один и тот же тип контента для всех. Здесь нам снова поможет NPM. Пакет `mime` (индикаторы типов контента, такие как `text/plain`, также называемые MIME-типами) позволяет определить тип файлов для множества расширений.

Следующая `npm`-команда, выполненная в каталоге, где находится серверный сценарий, устанавливает определенную версию `mime`:

```
$ npm install mime@2.2.0
```

Если запрошенный файл не существует, то правильный код возвращаемого состояния HTTP — 404. Мы будем использовать функцию `stat`, которая ищет информацию о файле, чтобы выяснить, существует ли такой файл и является ли он каталогом.

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "Файл не найден"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

Поскольку метод `stat` обращается к диску и, следовательно, это может занять некоторое время, этот метод является асинхронным. Поскольку мы используем промисы, а не обратные вызовы, этот метод необходимо импортировать из `promises`, а не напрямую из `fs`.

Если файла не существует, `stat` выдаст объект ошибки со свойством `code`, равным `"ENOENT"`. Эти несколько непонятные коды в стиле Unix — то, как распознаются типы ошибок в Node.

Объект `stats`, возвращаемый `stat`, сообщает о файле такую информацию, как размер (свойство `size`) и дата изменения (свойство `mtime`). В данном случае нас интересует, является объект каталогом или обычным файлом, о чем сообщает метод `isDirectory`.

Для того чтобы прочитать массив файлов в каталоге и вернуть его клиенту, мы используем `readdir`. Для обычных файлов мы создаем читаемый поток с помощью `createReadStream` и возвращаем его в виде тела вместе с типом контента, который нам сообщает пакет `mime` по имени файла.

Код для обработки запросов `DELETE` выглядит немного проще.

```
const {rmdir, unlink} = require("fs").promises;

methods.DELETE = async function(request) {
```

```
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

Если ответ HTTP не содержит никаких данных, то, чтобы сообщить об этом, можно использовать код состояния 204 («отсутствие содержимого»). Поскольку при ответе на удаление не нужно передавать какую-либо информацию, кроме того, была ли операция успешной, возвращение такого кода имеет смысл.

Возможно, вас интересует, почему попытка удалить несуществующий файл возвращает код состояния успеха, а не ошибку. Если удаляемого файла нет, то можно сказать, что цель запроса уже выполнена. Стандарт HTTP побуждает создавать *идемпотентные* запросы, то есть такие, при которых многократное выполнение одного и того же запроса приводит к тому же результату, что и однократное выполнение. В некотором смысле, если попытаться удалить что-то, что уже и так исчезло, мы получим правильный результат: этого больше нет.

Обработчик для запросов `PUT` выглядит следующим образом:

```
const {createWriteStream} = require("fs");

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

В этот раз нам не нужно проверять, существует ли этот файл: если он есть, то мы просто перезапишем его. Мы снова используем `pipe` для перемещения данных из читаемого потока в записываемый поток, в нашем случае из запроса в файл. Но так как метод `pipe` не рассчитан на возврат промиса, мы должны написать оболочку `pipeStream`, которая создает промис вокруг результата вызова `pipe`.

Если при открытии файла что-то пойдет не так, `createWriteStream` все равно вернет поток, но он вызовет событие `"error"`. Выходной поток также может завершиться сбоем — например, если произойдет сбой в сети. Поэтому мы связываем события `"error"` обоих потоков, чтобы отклонить промис. Когда метод `pipe` завершит работу, он закроет выходной поток, что вызовет событие `"finish"`. В этой точке можно успешно выполнить промис (ничего не возвращая).

Полный сценарий для сервера доступен по адресу https://eloquentjavascript.net/code/file_server.js. Его можно скачать и после установки всех зависимостей запустить из Node, и у вас получится собственный файловый сервер. И конечно же, вы можете его изменять и расширять, чтобы выполнить упражнения этой главы или поэкспериментировать.

Для выполнения HTTP-запросов можно использовать утилиту командной строки `curl`, широко доступную в Unix-подобных операционных системах (таких как macOS и Linux). Следующая сессия кратко тестирует наш сервер. Параметр `-X` применяется для выбора 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 для `file.txt` не выполняется, так как этот файл еще не существует. Запрос `PUT` создает файл, а следующий запрос успешно извлекает его.

После удаления по запросу `DELETE` файл снова отсутствует.

Резюме

Node — маленькая приятная система, которая позволяет запускать JavaScript в небраузерном контексте. Первоначально эта платформа была разработана для сетевых задач, чтобы играть роль *узла* в сети. Но она пригодна для любых видов задач, решаемых посредством сценариев, и если вам нравится писать код на JavaScript, то автоматизация задач с помощью Node придется вам по душе.

NPM предоставляет пакеты для всего, о чем только можно пожелать (и еще о многом, о чем вы, вероятно, никогда не догадывались), и позволяет получать и устанавливать эти пакеты с помощью программы `npm`. Node поставляется в комплекте с несколькими встроенными модулями, включая модуль `fs` для работы с файловой системой и модуль `http` для запуска HTTP-серверов и выполнения HTTP-запросов.

Все операции ввода и вывода в Node выполняются асинхронно, если только явно не использовать синхронный вариант функции, такой как `readFileSync`. При вызове таких асинхронных функций им передаются функции обратного вызова, и Node вызывает их, передавая им значение ошибки и (если он доступен) результат, когда он будет готов.

Упражнения

Инструмент поиска

В Unix-системах есть инструмент командной строки, называемый `grep`, который можно использовать для быстрого поиска файлов по регулярному выражению.

Напишите сценарий для Node, который можно запустить из командной строки и который действует как `grep`. Этот сценарий рассматривает свой первый аргумент командной строки как регулярное выражение, а все остальные аргументы — как файлы для поиска. Сценарий должен выводить имена всех файлов, чье содержимое соответствует регулярному выражению.

Когда это заработает, расширьте сценарий так, чтобы, если один из аргументов является каталогом, просматривались все файлы этого каталога и его подкаталогов.

Используйте асинхронные или синхронные функции файловой системы по своему усмотрению. Настройка параметров таким образом, чтобы одновременно запрашивать несколько асинхронных действий, может немного ускорить процесс, но не очень сильно, поскольку большинство файловых систем позволяют читать файлы только по очереди.

Создание каталога

Метод `DELETE` нашего файлового сервера позволяет удалять каталоги (используя `rmdir`), но в настоящее время у сервера нет никакого способа для их создания.

Добавьте поддержку метода `MKCOL` (от *make collection* — «создать коллекцию»), который должен создавать каталог, вызывая `mkdir` из модуля `fs`. `MKCOL` не является широко распространенным методом HTTP, но он существует для этой же цели в стандарте WebDAV, определяющем набор соглашений поверх HTTP, благодаря чему HTTP становится пригодным для создания документов.

Публичное пространство в сети

Поскольку файловый сервер обслуживает все типы файлов и даже содержит правильный заголовок `Content-Type`, его можно использовать для обслуживания сайта. Так как сервер позволяет всем желающим удалять и заменять файлы, это будет интересный вид сайта: сайт, который может быть изменен, улучшен и разрушен каждым, кто не поленится создать правильный HTTP-запрос.

Напишите минимальную HTML-страницу, включающую в себя простой файл JavaScript. Поместите файлы в каталог, обслуживаемый файловым сервером, и откройте их в браузере.

Затем в качестве расширенного упражнения или даже проекта выходного дня объедините все знания, почерпнутые вами из этой книги, и создайте более удобный интерфейс для изменения сайта — изнутри сайта.

Используйте HTML-форму для редактирования содержимого файлов, составляющих сайт, которая давала бы пользователю возможность обновлять их на сервере с помощью HTTP-запросов, как описано в главе 18.

Начните с редактирования только одного файла. Затем сделайте так, чтобы пользователь мог выбрать, какой файл редактировать. Используйте тот факт, что наш файловый сервер возвращает списки файлов при чтении каталога.

Не работайте напрямую с кодом, предоставляемым файловым сервером, поскольку если вы допустите ошибку, то, скорее всего, повредите файлы. Вместо этого держите свою работу вне общедоступного каталога и копируйте ее туда во время тестирования.

21 Проект: сайт по обмену опытом

> Если у тебя есть знание, позволь
> другим зажечь от него свечу.
>
> *Маргарет Фуллер*

Встреча по *обмену опытом* — это мероприятие, на котором собираются люди с общими интересами и устраивают небольшие неформальные презентации того, что они знают. Например, на встрече по обмену опытом садоводства могут научить выращивать сельдерей. А если вы зайдете в группу по обмену опытом программирования, то можете там рассказать о Node.js.

Такие встречи — также часто называемые *группами пользователей*, если речь идет о компьютерах, — отличный способ расширить кругозор, узнать о новых разработках или просто познакомиться с людьми, имеющими схожие интересы. Группы, посвященные JavaScript, есть во многих крупных городах. Их посещение, как правило, бесплатно, а в тех, где я бывал, люди были дружелюбны и приветливы.

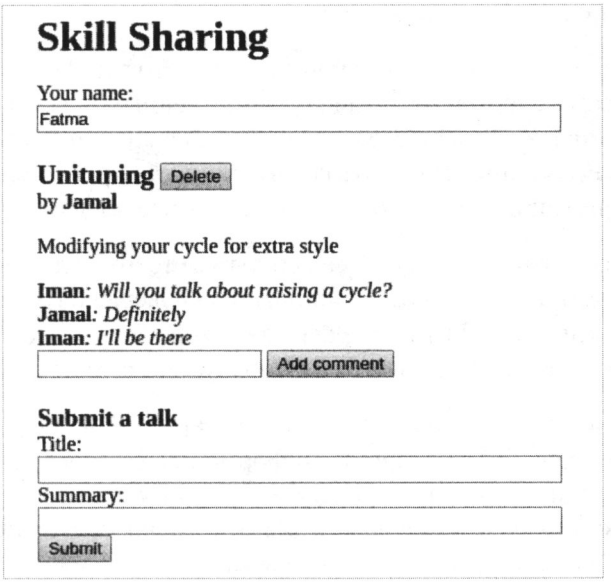
В этой последней главе проекта наша цель состоит в том, чтобы создать сайт для управления обсуждениями, которые проводятся на встрече по обмену опытом. Представьте себе небольшую группу людей, регулярно пересекающихся в офисе одного из участников, чтобы поговорить о велосипедах. Предыдущий организатор встреч переехал в другой город, и не нашлось желающих взять эту задачу на себя. Нам нужна система, которая позволила бы участникам предлагать и обсуждать разные темы без единого организатора.

Полный код проекта доступен по адресу https://eloquentjavascript.net/code/skillsharing.zip.

Структура

В этом проекте есть *серверная* часть, написанная для Node.js, и *клиентская* часть, написанная для браузера. Сервер хранит системные данные и предоставляет их клиенту. Он также обслуживает файлы, которые реализуют клиентскую часть системы.

На сервере хранится список тем для бесед, предложенных для следующей встречи, а клиентская часть показывает этот список пользователю. У каждой беседы есть имя докладчика, заголовок, резюме и массив связанных с ней комментариев. Клиентская часть позволяет пользователю предлагать новые беседы (добавляя их в список), удалять и комментировать существующие. Всякий раз, когда пользователь вносит такое изменение, клиентская часть делает HTTP-запрос, чтобы сообщить об этом серверу.



Приложение будет настроено так, чтобы показывать текущие предлагаемые темы бесед и комментарии к ним *в реальном времени*. Всякий раз, когда кто-либо откуда-либо предлагает новую беседу или добавляет комментарий, все, у кого в браузере открыта данная страница, должны немедленно увидеть изменение. Это представляет собой некоторую проблему — у веб-сервера нет возможности открыть соединение с клиентом, а также нет хорошего способа узнать, какие клиенты в данный момент просматривают указанный сайт.

Распространенное решение этой проблемы называется *длительным опросом*, что является одной из причин выбрать Node.

Длительный опрос

Чтобы иметь возможность немедленно уведомить клиента о том, что что-то изменилось, нам потребуется подключение к этому клиенту. Поскольку браузеры традиционно не принимают соединения, а перед клиентами часто стоят маршрутизаторы, которые в любом случае блокировали бы такие соединения, заставлять сервер инициировать подобное соединение было бы нецелесообразно.

Мы можем организовать открытие клиентом соединения и сохранение этого соединения таким образом, чтобы сервер мог использовать его для отправки информации по мере необходимости.

Но HTTP-запрос допускает только простой поток информации: клиент отправляет запрос, сервер возвращает единственный ответ, вот и все. Существует технология *WebSockets*, которая поддерживается современными браузерами и позволяет открывать соединения для произвольного обмена данными. Но правильно ее использовать несколько сложно.

В этой главе мы воспользуемся более простым методом — длительным опросом, когда клиенты постоянно запрашивают у сервера новую информацию, используя регулярные HTTP-запросы, а сервер задерживает ответ до тех пор, пока у него не появится что-нибудь свежее для отчета.

Поскольку клиент обеспечивает постоянно открытый запрос на опрашивание сервера, клиент будет получать информацию от сервера вскоре после того, как она станет доступной. Например, если у пользователя Фатьмы в браузере открыто наше приложение для обмена опытом, браузер отправит запрос на обновления и будет ждать ответа на этот запрос. Когда пользователь Иман отправит доклад об экстремальном велоспуске с горы, сервер заметит, что Фатьма ждет обновлений, и отправит на ее ожидающий запрос ответ, содержащий информацию о новой беседе. Браузер Фатьмы получит данные и обновит экран, чтобы показать новую беседу.

Чтобы предотвратить тайм-аут соединений (прерывание соединения вследствие отсутствия активности), методы длительного опроса обычно устанавливают максимальное время для каждого запроса, после чего сервер все

равно отвечает, даже если ему нечего сообщить, а затем клиент начинает новый запрос. Благодаря периодическому перезапуску запроса эта технология также становится более надежной, позволяя клиентам восстанавливаться после временных сбоев соединения или в случае проблем с сервером.

Если сервер, использующий длительный опрос, перегружен, то могут образоваться тысячи ожидающих запросов и, следовательно, открытых TCP-соединений. Для такой системы хорошо подходит узел, который позволяет легко управлять многими соединениями без создания отдельного потока управления для каждого из них.

HTTP-интерфейс

Прежде чем приступить к проектированию сервера или клиента, подумаем об их точке соприкосновения — HTTP-интерфейсе, через который они взаимодействуют.

В качестве формата тела запроса и ответа мы будем использовать JSON. Как и на файловом сервере из главы 20, мы постараемся эффективно задействовать методы и заголовки HTTP. Интерфейс сосредоточен вокруг пути `/talks`. Пути, которые не начинаются с `/talks`, будут использоваться для обслуживания статических файлов — HTML и JavaScript-кода для клиентской системы.

Запрос `GET` для `/talks` возвращает документ в формате JSON, например:

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]}
```

Создание новой беседы выполняется посредством отправки запроса `PUT` на URL-адрес вида `/talks/Unituning`, где после второй косой черты стоит заголовок беседы. В теле запроса `PUT` должен содержаться объект JSON, имеющий свойства `presenter` и `summary`.

Поскольку заголовки бесед могут содержать пробелы и другие символы, которые обычно не отображаются в URL, при создании такого URL необходимо закодировать строки заголовка с помощью функции `encodeURIComponent`.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

Например, запрос на создание беседы о езде на холостом ходу может выглядеть так:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
 "summary": "Standing still on a unicycle"}
```

Такие URL-адреса также поддерживают запросы `GET` для получения представления беседы в формате JSON и запросы `DELETE` для удаления беседы.

Добавление комментария к беседе выполняется с помощью запроса `POST` на URL-адрес типа `/talks/Unituning/comments` с использованием тела JSON, в котором есть свойства `author` и `message`.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
 "message": "Will you talk about raising a cycle?"}
```

Для поддержки длительных опросов запросы `GET` по адресу `/talks` могут содержать дополнительные заголовки, которые информируют сервер о задержке ответа, если новая информация недоступна. Мы будем использовать пару заголовков, обычно предназначенных для управления кэшированием: `ETag` и `If-None-Match`.

Серверы могут включать в ответ заголовок `ETag` (от entity tag — «тег сущности»). Его значение является строкой, идентифицирующей текущую версию ресурса. Когда клиент позже снова запросит этот ресурс, он сможет сделать *условный запрос*, включив в него заголовок `If-None-Match`, значение которого содержит ту же строку. Если ресурс не изменился, сервер ответит кодом состояния 304, что означает «не изменен», сообщая клиенту, что его кэшированная версия все еще актуальна. Если же тег не совпадет, сервер ответит как обычно.

Нам нужно что-то подобное, чтобы клиент мог сообщить серверу, какая версия списка бесед у него имеется, и сервер отвечал бы только в том случае, если этот список изменился. Но вместо немедленного возврата ответа 304 сервер должен заблокировать ответ и возвратить его только тогда, когда появилось что-то новое или истекло заданное количество времени. Чтобы отличить запросы длительного опроса от обычных условных запросов, мы

добавляем еще один заголовок — `Prefer: wait=90`, который сообщает серверу, что клиент готов ждать ответа в течение 90 секунд.

Сервер будет хранить номер версии, обновляемый каждый раз при изменении списка бесед, и будет использовать этот номер в качестве значения `ETag`. Клиенты могут отправлять подобные запросы, чтобы получать уведомления при изменении списка бесед:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(прошло время)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

Описанный здесь протокол не предусматривает контроля доступа. Любой пользователь может прокомментировать беседу, изменить и даже удалить ее. (Поскольку в Интернете полно хулиганов, запуск такой системы в режиме онлайн без дополнительной защиты, вероятно, не закончился бы ничем хорошим.)

Сервер

Начнем с построения серверной части программы. Код, представленный в этом разделе, работает в среде Node.js.

Маршрутизация

Для запуска HTTP-сервера серверная часть нашего проекта будет использовать функцию `createServer`. В функции, обрабатывающей новый запрос, нужно различать разные виды поддерживаемых запросов (которые определяются по методу и пути). Это можно сделать с помощью длинной цепочки операторов `if`, но есть способ получше.

Маршрутизатор — это компонент, помогающий отправить запрос функции, которая может его обработать. Например, можно указать маршрутизатору,

что запросы PUT с путем, соответствующим регулярному выражению /^\/talks\/([^\/]+)$/ (/talks/, после которого следует заголовок беседы), могут быть обработаны заданной функцией. Кроме того, маршрутизатор может извлекать значимые части пути (в данном случае заголовок беседы), которые в регулярном выражении заключены в круглые скобки, и передавать их в функцию-обработчик.

В NPM есть несколько хороших пакетов маршрутизаторов, но в этом проекте мы сами напишем такой маршрутизатор, чтобы проиллюстрировать данный принцип.

Вот содержимое модуля маршрутизатора router.js, которое позже потребуется для нашего серверного модуля:

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
    }
    return null;
  }
};
```

Этот модуль экспортирует класс Router. Объект маршрутизатора позволяет регистрировать новые обработчики с помощью метода add и разрешать запросы с помощью метода resolve.

Метод resolve возвращает ответ, когда найден обработчик, и null в противном случае. Он перебирает маршруты по одному (в том порядке, в котором они были определены), пока не будет найден подходящий.

Функции-обработчики принимают в виде аргументов значение `context` (которое в нашем случае является экземпляром сервера), строки соответствия для всех групп, которые они определили в своем регулярном выражении, и объект запроса. Строки должны быть URL-декодированы, поскольку необработанный URL может содержать коды вида `%20`.

Обслуживание файлов

Если запрос не соответствует ни одному из типов запросов, определенных в маршрутизаторе, сервер должен интерпретировать его как запрос файла, содержащегося в каталоге `public`. Для обслуживания таких файлов можно было бы использовать файловый сервер, описанный в главе 20, но нам не нужно (да мы и не хотим) поддерживать запросы `PUT` и `DELETE` для файлов, и мы хотели бы иметь расширенные функции, такие как поддержка кэширования. Поэтому вместо этого используем надежный, хорошо протестированный статический файловый сервер из NPM.

Я выбрал пакет `ecstatic`. В NPM есть и другие подобные серверы, но `ecstatic` хорошо работает и соответствует нашим целям. Пакет `ecstatic` экспортирует функцию, которая может быть вызвана с объектом конфигурации для создания функции обработчика запроса.

Чтобы сообщить серверу, где следует искать файлы, мы воспользуемся параметром `root`. Функция обработчика принимает параметры запроса и ответа и может быть передана непосредственно в `createServer` для создания сервера, который обслуживает только файлы. Мы хотим сначала проверить запросы, требующие специальной обработки, поэтому обернем их в другую функцию.

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];
```

```
    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
            return {body: String(error), status: 500};
          }).then(({body,
                    status = 200,
                    headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

Для ответов здесь используется примерно такое же соглашение, что и в файловом сервере из предыдущей главы, — обработчики возвращают промисы, которые разрешают объекты, описывающие ответ. Сервер обернут в объект, хранящий также и его состояние.

Беседы как ресурсы

Предлагаемые беседы хранятся в свойстве `talks` сервера. Это свойство представляет собой объект, имена свойств которого являются заголовками бесед. Данные имена будут отображаться в виде HTTP-ресурсов как `/talks/[заголовок]`, поэтому в наш маршрутизатор нужно добавить обработчики, реализующие различные методы, которые клиенты могут использовать для работы с этими именами.

Обработчик запросов, позволяющий методом `GET` получить отдельную беседу, должен найти эту беседу и возвратить либо данные JSON о ней, либо ответ в виде ошибки 404.

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (title in server.talks) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Для того чтобы удалить беседу, нужно удалить ее из объекта `talks`.

```
router.add("DELETE", talkPath, async (server, title) => {
  if (title in server.talks) {
    delete server.talks[title];
    server.updated();
  }
  return {status: 204};
});
```

Метод `updated`, который мы определим позже, уведомляет об ожидающих запросах длительного опроса, касающихся изменения беседы.

Чтобы извлечь содержимое тела запроса, мы определим функцию `readStream`, которая считывает все содержимое из потока чтения и возвращает промис, разрешающийся в строку.

```
function readStream(stream) {
  return new Promise((resolve, reject) => {
    let data = "";
    stream.on("error", reject);
    stream.on("data", chunk => data += chunk.toString());
    stream.on("end", () => resolve(data));
  });
}
```

Одним из обработчиков, которому требуется читать тела запросов, является обработчик `PUT`, используемый для создания новых бесед. Такой обработчик должен проверить, присутствуют ли в предоставленных данных свойства `presenter` и `summary`, значениями которых являются строки. Любые данные, поступающие извне системы, могут быть бессмысленными, и мы не хотим, чтобы наша внутренняя модель данных оказалась испорченной или чтобы сервер дал сбой в случае поступления некорректных запросов.

Если данные выглядят корректными, то обработчик сохраняет объект, представляющий новую беседу, в объекте `talks`, возможно перезаписывая уже существующую беседу с таким заголовком, и снова вызывает метод `updated`.

```
router.add("PUT", talkPath,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let talk;
  try { talk = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!talk ||
      typeof talk.presenter != "string" ||
      typeof talk.summary != "string") {
    return {status: 400, body: "Bad talk data"};
  }
  server.talks[title] = {title,                          presenter:
talk.presenter,
                         summary: talk.summary,
                         comments: []};
  server.updated();
  return {status: 204};
});
```

Добавление комментария к беседе выполняется аналогично. Мы используем `readStream`, чтобы получить содержимое запроса, проверить полученные данные и сохранить их в виде комментария, если они выглядят корректно.

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let comment;
  try { comment = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
    return {status: 400, body: "Bad comment data"};
  } else if (title in server.talks) {
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Попытка добавить комментарий к несуществующей беседе возвращает ошибку 404.

Поддержка длительных опросов

Наиболее интересным аспектом сервера является та его часть, которая обрабатывает длительные опросы. Запрос `GET`, поступивший для `/talks`, может быть либо обычным запросом, либо запросом на длительный опрос.

Есть много ситуаций, когда требуется отправить клиенту массив сообщений, поэтому сначала мы определим вспомогательный метод, который создает такой массив и включает в ответ заголовок `ETag`.

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = [];
  for (let title of Object.keys(this.talks)) {
    talks.push(this.talks[title]);
  }
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`}
  };
};
```

Сам обработчик должен просматривать заголовки запроса и проверять, присутствуют ли там заголовки `If-None-Match` и `Prefer`. Node хранит заголовки, имена которых указаны без учета регистра, под их именами, записанными строчными буквами.

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

Если теги не были указаны или если был указан тег, не соответствующий текущей версии сервера, обработчик выдает в ответ список всех бесед. Если

запрос является условным и беседы с тех пор не менялись, то обработчик просматривает заголовок `Prefer`, чтобы узнать, следует отложить ответ или ответить сразу.

Функции обратного вызова для отложенных запросов хранятся в массиве `waiting` сервера, чтобы они могли получать уведомления, когда что-то происходит. Метод `waitForChanges` также немедленно устанавливает таймер, чтобы выдать ответ с состоянием 304, если время ожидания запроса окажется слишком длинным.

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
};
```

При регистрации изменений с помощью метода `updated` свойство `version` увеличивается на единицу и активируются все ожидающие запросы.

```
SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

Теперь код сервера готов. Если мы создадим экземпляр `SkillShareServer` и запустим его на порте 8000, то получится HTTP-сервер, который будет обслуживать файлы из подкаталога `public` и обеспечивать интерфейс управления беседами по URL-адресу `/talks`.

```
new SkillShareServer(Object.create(null)).start(8000);
```

Клиент

Клиентская часть сайта для обмена опытом состоит из трех файлов: крошечной HTML-страницы, таблицы стилей и файла JavaScript.

HTML

Широко применяется соглашение для веб-серверов: в случае запроса непосредственно для пути, соответствующего каталогу, пытаться обслуживать файл с именем `index.html`. Используемый нами модуль файлового сервера `ecstatic` тоже поддерживает это соглашение. Когда делается запрос к пути `/`, сервер ищет файл `./public/index.html` (`./public` является корневым каталогом, который мы ему назначили) и возвращает файл `index.html`, если он существует.

Таким образом, если мы хотим, чтобы при переходе браузера на наш сервер открывалась веб-страница, нам нужно разместить ее в файле `public/index.html`. Вот наш файл `index.html`:

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

<script src="skillsharing_client.js"></script>
```

В этом файле определен заголовок документа и подключена таблица стилей, определяющая несколько стилей, создающих, кроме прочего, некоторое пространство между беседами.

Далее добавлен заголовок, расположенный вверху страницы, и загружается скрипт, содержащий клиентское приложение.

Действия

Состояние приложения представляет собой список бесед и имя пользователя; мы будем хранить это состояние в объекте `{talks, user}`. Мы не разрешаем пользовательскому интерфейсу напрямую манипулировать состоянием или отправлять HTTP-запросы. Вместо этого он может создавать *действия*, описывающие то, что хочет сделать пользователь.

Функция `handleAction` принимает данные действия и выполняет их. Поскольку в нашем случае изменения состояния очень просты, все они обрабатываются одной и той же функцией.

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
    return Object.assign({}, state, {user: action.user});
  } else if (action.type == "setTalks") {
    return Object.assign({}, state, {talks: action.talks});
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

Мы будем хранить имя пользователя в `localStorage`, чтобы его можно было восстановить при загрузке страницы.

Действия, которые задействуют сервер, создают с помощью `fetch` сетевые запросы к HTTP-интерфейсу, описанному ранее. Мы используем функцию-обертку `fetchOK`, которая гарантирует, что возвращенный промис будет отклонен, если сервер вернет код ошибки.

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

Следующая вспомогательная функция используется для создания URL-адреса для беседы, имеющей заданный заголовок.

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

Если запрос завершается неудачно, мы не хотим, чтобы страница осталась без изменений, ничего не делая и не объясняя причин. Поэтому мы определим функцию с именем `reportError`, которая по крайней мере покажет пользователю диалоговое окно с сообщением о том, что что-то пошло не так.

```
function reportError(error) {
  alert(String(error));
}
```

Визуализация компонентов

Мы воспользуемся подходом, аналогичным тому, который был описан в главе 19, и разделим приложение на компоненты. Но, поскольку отдельные компоненты либо не нуждаются в обновлении, либо при обновлении всегда полностью перерисовываются, мы определим их не как классы, а как функции, напрямую возвращающие узел DOM. Например, вот компонент, который отображает поле, где пользователь может ввести свое имя:

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

Для создания элементов DOM мы использовали ту же функцию `elt`, что и в главе 9.

Аналогичная функция применяется для визуализации бесед, включая список комментариев и форму для добавления нового комментария.

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
```

```
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

Обработчик события `"submit"` вызывает метод `form.reset`, чтобы очистить содержимое формы после создания действия `"newComment"`.

При создании сколько-нибудь сложных частей DOM такой стиль программирования начинает выглядеть довольно неряшливо. Существует широко используемое (нестандартное) расширение JavaScript, называемое *JSX*, которое позволяет писать HTML-код прямо в сценариях, что может сделать такой код немного красивее (в зависимости от того, что считать красивым). Прежде чем вы сможете запустить такой код, вам нужно будет запустить программу в сценарии, чтобы преобразовать псевдо-HTML в вызовы функций JavaScript, очень похожие на те, которые мы здесь используем.

Визуализировать комментарии проще.

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

Наконец, форма, с помощью которой пользователь может создать новую беседу, визуализируется следующим образом:

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Submit a Talk"),
     elt("label", null, "Title: ", title),
     elt("label", null, "Summary: ", summary),
     elt("button", {type: "submit"}, "Submit"));
}
```

Опросы

Для запуска приложения нам нужен текущий список бесед. Поскольку начальная загрузка тесно связана с процессом длительного опроса — при опросе, сопутствующем начальной загрузке, необходимо использовать `ETag`, — мы напишем функцию, которая продолжает опрашивать сервер по адресу `/talks` и запускает функцию обратного вызова, когда поступает новый набор бесед.

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
        headers: tag && {"If-None-Match": tag,
                         "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("Request failed: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("Etag");
    update(await response.json());
  }
}
```

Это асинхронная функция, упрощающая цикл и ожидание запроса. Она запускает бесконечный цикл, который на каждой итерации извлекает список бесед — либо обычный, либо, если это не первый запрос, с включенными заголовками, превращающими его в запрос длительного опроса.

Если запрос завершается неудачей, функция немного ждет, а затем повторяет попытку. Таким образом, если сетевое соединение на некоторое время разорвется, а затем возобновится, то приложение сможет восстановиться и продолжить изменения. Для того чтобы заставить асинхронную функцию ждать, используется промис, разрешенный с помощью `setTimeout`.

Если сервер возвращает ответ 304, это означает, что время запроса длительного опроса истекло, поэтому функция должна просто немедленно запустить следующий запрос. Если ответ является обычным ответом с состоянием 200, то его тело считывается как JSON и передается функции обратного вызова, а значение заголовка `ETag` сохраняется для следующей итерации.

Приложение

Следующий компонент связывает весь пользовательский интерфейс воедино:

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.syncState(state);
  }

  syncState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

При изменении бесед указанный компонент заново визуализирует весь список. Это просто, однако расточительно. Мы еще вернемся к данному вопросу в упражнениях.

Запустить приложение можно следующим образом:

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

Если запустить сервер и открыть сразу два окна браузера для адреса http://localhost:8000, то вы увидите, что действия, которые выполняются в одном окне, сразу же видны в другом.

Упражнения

Следующие упражнения подразумевают изменение системы, описанной в этой главе. Чтобы поработать над ними, не забудьте сначала загрузить код (https://eloquentjavascript.net/code/skillsharing.zip), установить Node со страницы https://nodejs.org и установить зависимости проекта с помощью команды `npm install`.

Хранение на диске

Сервер обмена опытом хранит данные исключительно в памяти. Это означает, что в случае сбоя или перезапуска по любой причине все разговоры и комментарии будут потеряны.

Дополните сервер так, чтобы он сохранял информацию о беседе на диск и автоматически перезагружал ее при перезапуске. Не беспокойтесь об эффективности — сделайте самое простое, лишь бы работало.

Сброс поля комментариев

Полная повторная визуализация всех бесед работает довольно хорошо, потому что мы, как правило, не можем определить разницу между DOM-узлом и его идентичной заменой. Но бывают исключения. Если начать что-то вводить в поле комментария для беседы в одном окне браузера, а затем в другом добавить комментарий к этой беседе, то поле в первом окне будет визуализировано заново, так что пропадет и его содержимое, и фокус.

Во время жаркой дискуссии, когда несколько человек добавляют комментарии одновременно, это будет раздражать. Можете ли вы придумать способ решить такую проблему?

Советы по выполнению упражнений

Следующие подсказки, возможно, помогут вам, если у вас возникнут трудности с одним из упражнений этой книги. Они не раскрывают полное решение, а скорее пытаются помочь вам найти его самостоятельно.

Структура программы

Построение треугольника в цикле

Для начала можно написать программу, которая печатает числа от 1 до 7. Такую программу можно получить, внеся несколько изменений в пример печати четных чисел, приведенный ранее в главе, где был введен цикл `for`.

Теперь рассмотрим эквивалентность между числами и строками, состоящими из символов «решетка». Чтобы перейти от 1 до 2, нужно прибавить 1 (`+= 1`). Чтобы перейти от `"#"` к `"##"`, нужно прибавить символ (`+= "#"`). Таким образом, ваше решение может быть почти таким же, как и для печати цифр.

FizzBuzz

Перебор чисел — это, очевидно, задача для цикла, а выбор того, что нужно напечатать, — вопрос условного выполнения. Вспомните прием с использованием оператора остатка от деления (`%`) для проверки, делится ли данное число на другое число (остаток равен нулю).

В первой версии есть три возможных результата для каждого числа, поэтому вам придется создать цепочку `if/else` `if/else`.

У второй версии программы есть два решения: одно из них простое, а другое — красивое. Простое решение состоит в том, чтобы добавить еще одну условную «ветвь», позволяющую точно проверить данное условие. Для красивого решения создайте строку, содержащую одно или несколько слов, которые будут выводиться, и печатайте либо данное слово, либо число,

если подходящего слова нет. Возможно, для этого у вас получится удачно использовать оператор `||`.

Шахматная доска

Вы можете создать строку, начав с пустой (`""`) и многократно добавляя символы. Символ новой строки обозначается как `"\n"`.

Для работы с двумя измерениями вам понадобится цикл внутри цикла. Заключите в скобки тела обоих циклов, чтобы было лучше заметно, где они начинаются и где заканчиваются. Постарайтесь правильно разместить отступы этих циклов. Последовательность прохождения циклов должна соответствовать последовательности построения строки (строка за строкой, слева направо, сверху вниз). Таким образом, внешний цикл обрабатывает строки, а внутренний — символы в строке.

Чтобы отслеживать выполнение циклов, вам понадобятся две привязки. Чтобы узнать, нужно ли ставить пробел или «решетку» в заданной позиции, можно проверить, является ли сумма двух счетчиков четной (`% 2`).

Завершение строки путем добавления символа новой строки должно происходить после того, как строка сформирована, так что делайте это после прохождения внутреннего цикла, но внутри внешнего цикла.

Функции

Минимум

Если у вас возникли проблемы с размещением фигурных и обычных скобок в нужных местах, чтобы получилось правильное определение функции, начните с копирования одного из примеров этой главы и его изменения.

Функция может содержать несколько инструкций `return`.

Рекурсия

Ваша функция, скорее всего, будет выглядеть примерно как внутренняя функция `find` в примере рекурсии `findSolution` из данной главы с цепочкой

`if/else` `if/else`, в которой выбирается один из этих трех случаев. Заключительная ветвь `else`, соответствующая третьему случаю, делает рекурсивный вызов. Каждая из ветвей должна содержать инструкцию `return` или каким-либо другим образом организовывать возвращение соответствующего значения.

Если задано отрицательное число, то функция будет повторяться без конца, передавая сама себе увеличивающиеся по модулю отрицательные числа и тем самым все сильнее удаляясь от возможности возврата результата. Это в итоге приведет к тому, что пространство стека исчерпается и браузер прервет работу.

Подсчет букв

Вашей функции нужен цикл, который просматривает каждый символ в строке. Можно перебрать все индексы от нуля до значения, которое на единицу меньше длины строки (`< string.length`). Если символ, находящийся в текущей позиции, совпадает с тем, который ищет функция, то значение переменной счетчика увеличивается на единицу. После завершения цикла счетчик можно снова обнулить.

Позаботьтесь о том, чтобы все привязки, используемые в функции, были *локальными* для нее, правильно объявив их с помощью ключевого слова `let` или `const`.

Структуры данных: объекты и массивы

Сумма диапазона

Для создания массива будет проще всего сначала инициализировать привязку к `[]` (новый пустой массив), а затем многократно вызвать для этого массива метод `push`, чтобы добавить в него значения. Не забудьте вернуть массив в конце выполнения функции.

Поскольку конечная граница является последним элементом массива, при проверке условия выхода из цикла вам нужно будет применить оператор `<=`, а не `<`.

Параметр шага необязателен, и по умолчанию (при использовании оператора =) ему присваивается значение 1.

Для того чтобы функция `range` поддерживала отрицательные значения шага, вероятно, лучше всего будет написать два отдельных цикла — один для прямого отсчета и один для обратного — потому что при обратном отсчете в условии завершения цикла должен стоять знак `>=`, а не `<=`.

Также, возможно, если конечное значение диапазона меньше начального, было бы целесообразно использовать по умолчанию другой шаг, а именно −1. Тогда `range(5, 2)` будет возвращать осмысленное значение, вместо того чтобы застревать в бесконечном цикле. В значении параметра по умолчанию можно сослаться на предыдущие параметры.

Массив в обратном порядке

Существует два очевидных способа реализации функции `reverseArray`. Первый — просто перебрать исходный массив с начала до конца и использовать метод `unshift`, чтобы вставлять каждый следующий элемент в начало нового массива. Второй — перебрать исходный массив в обратном порядке и применить метод `push`. Перебор массива в обратном порядке требует (несколько неуклюжего) варианта `for`, такого как `(let i = array.length - 1; i >= 0; i--)`.

Обратить массив на месте сложнее. Вы должны быть осторожны, чтобы не перезаписать элементы, которые позже вам понадобятся. Использование `reverseArray` или другого варианта с копированием всего массива (хорошим способом скопировать массив является `array.slice(0)`) — рабочий, но не очень честный способ.

Хитрость заключается в том, чтобы поменять местами первый и последний элементы, затем второй и предпоследний и т. д. Чтобы это сделать, нужно перебрать в цикле половину массива (используйте `Math.floor` для округления — если в массиве нечетное число элементов, то трогать средний элемент не понадобится) и поменять местами элемент, находящийся в позиции `i`, и элемент в позиции `array.length - 1 - i`. Можно использовать локальную привязку, чтобы временно сохранить один из элементов, перезаписать на его место его зеркальное отображение, а затем поместить исходное значение из локальной привязки в то место, где раньше находилось зеркальное отображение элемента.

Список

При построении списка проще двигаться с конца. Таким образом, `arrayToList` может перебирать массив в обратном направлении (см. предыдущее упражнение) и для каждого элемента добавлять в список новый объект. Вы можете использовать локальную привязку для хранения части списка, которая была построена до сих пор, и применять присвоение, например `{value: X, rest: list}`, чтобы добавить элемент.

Для прохода по списку (в функциях `listToArray` и `nth`) можно использовать следующий вариант цикла `for`:

```
for (let node = list; node; node = node.rest) {}
```

Понимаете ли вы, как это работает? На каждой итерации цикла `node` указывает на текущий подсписок, и тело может прочитать его свойство `value`, чтобы получить текущий элемент. В конце итерации `node` переходит к следующему подсписку. Когда будет `null`, это будет означать, что мы достигли конца списка и цикл завершен.

Рекурсивная версия `nth` будет аналогичным образом рассматривать все меньшую часть «хвоста» списка и в то же время уменьшать индекс до тех пор, пока он не достигнет нуля, после чего сможет вернуть свойство `value` для текущего узла. Чтобы получить нулевой элемент списка, нужно просто взять свойство `value` его головного узла. Чтобы получить $N+1$-й элемент, нужно взять N-й элемент списка, который находится в его свойстве `rest`.

Глубокое сравнение

Проверка того, имеете ли вы дело с реальным объектом, будет выглядеть примерно так: `typeof x == "object" && x != null`. Будьте внимательны: сравнивайте свойства только тогда, когда *оба* аргумента являются объектами. Во всех остальных случаях вы можете сразу же вернуть результат применения `===`.

Для просмотра свойств используйте `Object.keys`. Необходимо проверить, имеют ли оба объекта одинаковый набор имен свойств и имеют ли эти свойства одинаковые значения. Один из способов сделать это — убедиться, что у обоих объектов одинаковое количество свойств (длина списков свойств обоих объектов одинакова). Затем, когда будете перебирать свойства одного из объектов, чтобы сравнить их со свойствами другого объекта, всегда

сначала убедитесь, что у другого объекта действительно есть свойство с таким именем. Если оба объекта имеют одинаковое количество свойств и все свойства одного объекта также существуют у другого объекта, то они имеют одинаковый набор имен свойств.

Что касается возврата значения функции, то лучше всего сразу возвращать `false`, как только будет обнаружено несоответствие, и возвращать `true` в конце функции.

Функции высшего порядка

Метод every

Подобно оператору `&&`, метод `every` может прекратить сравнение дополнительных элементов, как только будет найден первый несоответствующий элемент. Таким образом, основанная на цикле версия может принудительно выйти из цикла — посредством `break` или `return` — сразу, как только натолкнется на элемент, для которого функция предиката возвращает `false`. Если цикл завершится, а такой элемент не будет обнаружен, это будет означать, что все элементы совпадают и нужно вернуть `true`.

Чтобы построить `every` на основе `some`, можно применить *законы Де Моргана*, согласно которым `a && b` эквивалентно `!(!a || !b)`. Это правило можно распространить и на массивы, где все элементы совпадают, если в массиве нет ни одного несовпадающего элемента.

Доминирующее направление письма

Ваше решение может выглядеть аналогично первой половине примера `textScripts`. Здесь тоже нужно посчитать символы по критерию, основанному на `characterScript`, а затем отфильтровать ту часть результата, которая относится к не интересующим нас (не имеющим направления) символам.

Поиск направления, к которому относится наибольшее количество символов, можно выполнить с помощью метода `reduce`. Если вам непонятно, как это сделать, обратитесь к примеру, приведенному ранее в данной главе, где был использован метод `reduce`, чтобы найти сценарий с наибольшим количеством символов.

Тайная жизнь объектов

Тип вектора

Если вы не помните, как выглядят объявления классов, вернитесь к примеру с классом `Rabbit`.

Чтобы добавить в конструктор свойство геттера, можно поместить слово `get` перед именем метода. Чтобы вычислить расстояние между точками (0, 0) и (x, y), можно использовать теорему Пифагора, согласно которой квадрат искомого расстояния равен сумме квадрата координаты x и квадрата координаты y. Таким образом, нужное нам число равно $\sqrt{x^2 + y^2}$, а в JavaScript квадратный корень вычисляется с помощью метода `Math.sqrt`.

Группы

Самый простой способ выполнить задание — сохранить массив членов группы в виде свойства экземпляра. Для проверки наличия данного значения в массиве можно использовать метод `includes` или `indexOf`.

Конструктор вашего класса может создавать коллекцию членов как пустой массив. При вызове метода `add` он должен проверить, есть ли уже такое значение в массиве, и если нет, то добавить его, например, с помощью метода `push`.

Удаление элемента из массива методом `delete` выполняется не так просто, но вы можете использовать фильтр для создания нового массива без этого значения. Не забудьте перезаписать в свойство, содержащее элементы, новую, отфильтрованную версию массива.

В методе `from` можно применить цикл `for/of`, позволяющий получить значения из итерируемого объекта, и вызвать `add`, чтобы поместить их во вновь созданную группу.

Итерируемые группы

Вероятно, стоит определить новый класс — `GroupIterator`. Экземпляры итератора должны иметь свойство, которое отслеживает текущую позицию в группе. При каждом вызове метода `next` нужно проверять, достигнут ли

конец, и если нет, то переместиться на следующее значение и возвратить его как текущее.

Класс `Group` сам получает метод с именем `Symbol.iterator`, при вызове которого возвращается новый экземпляр класса итератора для данной группы.

Заимствование метода

Помните, что методы, которые существуют для простых объектов, унаследованы из `Object.prototype`.

Помните также, что мы можем вызывать функцию со специальной привязкой `this`, используя ее метод `call`.

Проект: робот

Измерение параметров робота

Вам нужно написать вариант функции `runRobot`, которая вместо записи событий в консоль возвращает количество шагов, пройденных роботом для выполнения задачи.

Затем измерительная функция может в цикле генерировать новые состояния и подсчитывать шаги, которые выполняет каждый из роботов. После того как будет выполнено достаточное количество измерений, функция может с помощью `console.log` вывести среднее значение для каждого робота, которое представляет собой суммарное количество сделанных шагов, разделенное на количество измерений.

Эффективность робота

Основным ограничением `goalOrientedRobot` является то, что он рассматривает посылки по одной. Он часто будет ходить взад-вперед по деревне, потому что посылка, которой он занят в данный момент, находится на другой стороне карты, даже если есть другие, находящиеся гораздо ближе.

Одним из возможных решений будет сначала вычислить маршруты для всех посылок, а затем выбрать самый короткий из них. Если есть несколько кратчайших маршрутов, можно получить еще лучшие результаты, отдав

предпочтение тем из них, в которых требуется забрать посылку, а не доставить ее.

Постоянная группа

Наиболее удобный способ представления множества значений членов группы — это по-прежнему массив, поскольку массивы легко копировать.

При добавлении значения в группу можно создавать новую группу, содержащую копию исходного массива, к которому добавлено новое значение (например, с помощью `concat`). При удалении значения оно отфильтровывается из массива.

Конструктор класса может принимать такой массив в качестве аргумента и сохранять его как свойство экземпляра (только). Этот массив никогда не обновляется.

Для того чтобы добавить в конструктор свойство (`empty`), которое не является методом, нужно добавить его в конструктор как обычное свойство после определения класса.

Вам нужен только один экземпляр `empty`, так как все пустые группы одинаковы, поэтому экземпляры класса не будут меняться. Вы можете создать на основе этой единственной пустой группы много разных групп, не затрагивая ее.

Ошибки и дефекты

Повторная попытка

Функция `primitiveMultiply` обязательно должна вызываться в блоке `try`. Если это исключение является экземпляром `MultiplicatorUnitFailure`, то соответствующий блок `catch` должен гарантировать повторение вызова, в противном случае — перебрасывать исключение.

Для повторной попытки можно использовать цикл, который останавливается только в случае успешного вызова — как показано в примере ранее в этой главе, — или использовать рекурсию и надеяться, что вы не будете получать строку с ошибками достаточно долго, чтобы переполнился стек (что довольно безопасно).

Запертый ящик

Это упражнение требует блока `finally`. Ваша функция должна сначала отпереть ящик, а затем вызвать функцию-аргумент из тела `try`. После этого блок `finally` должен снова запереть ящик.

Чтобы убедиться, что мы не закроем ящик, пока он еще не был заперт, проверьте в начале функции, является ли он закрытым, затем откройте и закройте только в том случае, если изначально он был закрыт.

Регулярные выражения

Стиль цитирования

Наиболее очевидное решение состоит в том, чтобы заменить только правые или только левые кавычки на неалфавитный символ — что-то вроде `/\W'|'\W/`. Но вам также следует учитывать начало и конец строки.

Кроме того, необходимо убедиться, что замена также включает в себя символы, соответствующие шаблону `\W`, чтобы случайно их не удалить. Это можно сделать, заключив такие символы в скобки и включив их группы в строку замены (`$1, $2`). Группы, которые не соответствуют этой строке, будут заменены пустой строкой.

Снова числа

Прежде всего, не забывайте ставить обратную косую черту перед точкой.

Для того чтобы учитывать в шаблоне необязательный знак перед числом и перед показателем степени, можно использовать `[+\-]?` или `(\+|-|)` (плюс, минус или ничего).

Более сложной частью упражнения является достижение соответствия шаблона вариантам `"5."` и `".5"`, но не `"."`. Для этого хорошим решением будет использование оператора `|` для разделения двух случаев — одна или несколько цифр, после которых могут следовать точка и ноль или несколько цифр *или* точка, после которой идет одна или несколько цифр.

Наконец, чтобы шаблон был нечувствительным к регистру, добавьте к регулярному выражению параметр `i` или используйте `[eE]`.

Модули

Модульный робот

Я бы сделал следующее (но, напомню, не существует единственного *правильного* способа разработки данного модуля).

Код, используемый для построения графа дорог, находится в модуле `graph`. Поскольку я предпочитаю задействовать пакет `dijkstrajs` из NPM, а не наш собственный код поиска пути, мы построим структуру графических данных того типа, которую ожидает `dijkstajs`. Этот модуль экспортирует только одну функцию — `buildGraph`. Я бы хотел, чтобы `buildGraph` принимала массив из двухэлементных массивов, а не строк, содержащих дефисы, чтобы этот модуль был менее зависимым от формата ввода.

Модуль `roads` содержит необработанные данные о дорогах (массив `roads`) и привязку `roadGraph`. Этот модуль зависит от `./graph` и экспортирует граф дорог.

Класс `VillageState` находится в модуле `state`. Он зависит от модуля `./roads`, так как должен быть в состоянии проверить, существует ли данная дорога. Ему также нужна функция `randomPick`. Поскольку это трехстрочная функция, мы могли бы просто поместить ее в модуль состояния как внутреннюю вспомогательную функцию. Но она также нужна для `randomRobot`. Поэтому нам придется либо продублировать ее, либо поместить в отдельный модуль. Поскольку данная функция существует в NPM в пакете `random-item`, хорошим решением будет просто сделать так, чтобы оба модуля зависели от этого пакета. Мы также можем добавить в данный модуль функцию `runRobot`, поскольку она невелика и тесно связана с управлением состоянием. Указанный модуль экспортирует класс `VillageState` и функцию `runRobot`.

Наконец, роботы и значения, от которых они зависят, такие как `mailRoute`, могут войти в модуль `example-robots`, зависящий от `./roads`, и экспортировать функции робота. Для того чтобы функция `goalOrientedRobot` могла найти маршрут, этот модуль также зависит от `dijkstrajs`.

Переложив часть работы на модули NPM, мы сделали код немного компактнее. Каждый отдельный модуль делает что-то довольно простое и может быть прочитан сам по себе. Разделение кода на модули также часто обеспечивает дальнейшие улучшения структуры программы. В данном случае кажется немного странным, что `VillageState` и роботы зависят от конкретного графа дорог. Возможно, было бы лучше сделать граф аргументом

конструктора состояния, чтобы роботы считывали его из объекта состояния, — это сократило бы число зависимостей (что всегда хорошо) и позволило бы запускать симуляции на разных картах (что еще лучше).

Будет ли хорошей идеей использовать NPM-модули для того, что мы могли бы написать и сами? Как правило, да — для таких нетривиальных вещей, как функция поиска пути, вы, скорее всего, будете делать ошибки и тратить время на их написание. Написать крошечные функции, такие как `random-item`, достаточно просто. Но добавление их в нужные места, скорее всего, приведет к загромождению ваших модулей.

Однако также не следует недооценивать работу, связанную с поиском подходящего NPM-пакета. И даже если вы его найдете, этот пакет может не работать должным образом или в нем может не хватать какой-то функции, которая вам нужна. Кроме того, при установке NPM-пакетов вам нужно будет каждый раз распространять их вместе со своей программой и, возможно, периодически их обновлять.

Это компромисс, и решение за вами, в зависимости от того, насколько данные пакеты помогают решить задачу.

Модуль Roads

Поскольку это модуль `CommonJS`, вы должны использовать `require`, чтобы импортировать модуль `graph`. Это было описано как экспорт функции `buildGraph`, которую можно выбрать из объекта интерфейса с помощью объявления `const` для деструктурирования.

Чтобы экспортировать `roadGraph`, нужно добавить свойство к объекту `exports`. Поскольку `buildGraph` принимает структуру данных, которая не точно соответствует `roads`, в вашем модуле нужно выполнить разделение строк, описывающих дороги.

Циклические зависимости

Хитрость заключается в том, что инструкция `require` добавляет модули в кэш программы *до того*, как начнется загрузка модулей. Таким образом, если какой-либо вызов `require` во время работы программы попытается загрузить модуль, который уже есть в кэше, то будет возвращен текущий интерфейс, вместо того чтобы начать загружать модуль еще раз (что в итоге приведет к переполнению стека).

Если модуль перезапишет свое значение `module.exports`, то любой другой модуль, получивший свое значение интерфейса до завершения загрузки, получит интерфейсный объект по умолчанию (который, скорее всего, будет пустым), а не предполагаемое значение интерфейса.

Асинхронное программирование

Где скальпель?

Это можно сделать с помощью одного цикла, который просматривает гнезда, перемещаясь к следующему, после того как найдет значение, не соответствующее имени текущего гнезда, и возвращает имя, если найдено совпадающее значение. В асинхронной функции можно использовать обычный цикл `for` или `while`.

Чтобы сделать то же самое в простой функции, вам будет нужно построить цикл, задействуя рекурсивную функцию. Самый простой способ это сделать — заставить функцию вернуть промис, вызвав затем промис, который извлекает значение хранилища. В зависимости от того, соответствует ли возвращенное значение имени текущего гнезда, обработчик возвращает это значение или дополнительный промис, созданный повторным вызовом функции цикла.

Не забудьте запустить цикл, вызвав один раз рекурсивную функцию из основной функции.

В функции `async` отклоненные промисы преобразуются в исключения посредством `await`.

Когда асинхронная функция генерирует исключение, ее промис отклоняется. Так что этот вариант работает.

Если вы реализовали неасинхронную функцию, как описано выше, то способ работы также автоматически приводит к сбою в возвращаемом промисе. Если запрос не выполняется, то обработчик, переданный в `then`, не вызывается, и промис, который он возвращает, отклоняется по той же причине.

Построение Promise.all

Функция, переданная конструктору `Promise`, должна вызываться из `then` для каждого из промисов в данном массиве. При успешном завершении

одного из таких вызовов должны происходить две вещи. Полученное значение должно быть сохранено в правильной позиции массива результатов, и нужно проверить, было ли это последним ожидающим промисом. Если да, то нужно закончить наш собственный промис.

Последнюю операцию можно реализовать с помощью счетчика, начальное значение которого равно длине исходного массива. Каждый раз, когда выполняется промис, значение счетчика уменьшается на единицу. Когда счетчик станет равен нулю — работа закончена. Не забудьте учесть ситуацию, когда входной массив пуст (и, следовательно, никакой промис никогда не разрешится).

Для обработки неудачных завершений придется немного подумать, но решение оказывается чрезвычайно простым. Просто передайте функцию `reject` из промиса-обертки каждому из промисов в массиве в качестве обработчика `catch` или как второй аргумент для `then`, чтобы сбой в одном из этих методов приводил к отклонению всего промиса-обертки.

Проект: язык программирования

Массивы

Самый простой способ выполнить задание — представить массивы `Egg` в виде массивов JavaScript.

Значения, добавляемые в главную область видимости, должны быть функциями. Можно создать очень простое определение массива, если использовать обозначение дополнительных аргументов (в виде многоточия).

Замыкание

Здесь мы снова можем воспользоваться механизмом JavaScript, чтобы получить эквивалентную функцию в `Egg`. Специальные формы передаются в локальную область видимости, в которой они вычисляются, так что можно вычислить их подчиненные формы в той же области видимости. Функция, возвращаемая функцией `fun`, имеет доступ к аргументу `scope`, определенному в ее внешней функции, и использует данный аргумент для создания локальной области видимости функции при ее вызове.

Это означает, что прототипом локальной области видимости будет область, в которой была создана функция, что позволяет получить доступ к привязкам

в данной области из функции. Это все, что нужно для реализации замыкания (впрочем, чтобы скомпилировать его действительно эффективным способом, вам придется проделать еще некоторую дополнительную работу).

Комментарии

Убедитесь, что ваше решение обрабатывает вариант с несколькими комментариями подряд, в том числе и с пробелами между ними или после них.

Вероятно, самый простой способ решить эту проблему — использовать регулярное выражение. Напишите что-нибудь, что соответствует варианту «пробел или комментарий, ноль или более раз». Задействуйте метод `exec` или `match` и проверяйте длину первого элемента в возвращаемом массиве (полное совпадение), чтобы узнать, сколько символов нужно вырезать.

Изменение области видимости

Вам придется перебирать области видимости по одной, в цикле, используя `Object.getPrototypeOf`, чтобы перейти к следующей внешней области. Для каждой области видимости задействуйте `hasOwnProperty`, чтобы узнать, существует ли в этой области привязка, определяемая свойством `name` первого аргумента метода `set`. Если такая привязка существует, то присвойте ей результат вычисления второго аргумента `set`, а затем верните это значение.

Если будет достигнута самая внешняя область (`Object.getPrototypeOf` возвратит значение `null`), а привязка все еще не будет найдена, это означает, что данная привязка не существует и следует выдать сообщение об ошибке.

Объектная модель документа

Построение таблицы

Для создания новых узлов элементов можно использовать `document.createElement`, для создания текстовых узлов — `document.createTextNode`, а для помещения узлов в другие узлы — метод `appendChild`.

Для того чтобы заполнить верхнюю строку, вы, вероятно, захотите перебрать в цикле имена ключей, а затем повторить это для каждого объекта массива, чтобы построить строки данных. Чтобы получить массив имен ключей из первого объекта, вам пригодится метод `Object.keys`.

Чтобы поместить таблицу в правильный родительский узел, можно использовать метод `document.getElementById` или же `document.querySelector`, чтобы найти узел с правильным атрибутом `id`.

Элементы по имени тега

Эту задачу проще всего решить с помощью рекурсивной функции, аналогичной функции `talkAbout`, описанной ранее в данной главе.

Вы можете вызывать рекурсивно сам метод `byTagname`, объединяя получаемые массивы для формирования выходных данных. Или же можно создать внутреннюю функцию, которая бы рекурсивно вызывала сама себя и имела доступ к привязке массива, определенной во внешней функции, к которой эта функция могла бы добавлять найденные элементы. Не забудьте один раз вызвать внутреннюю функцию из внешней функции, чтобы запустить процесс.

Рекурсивная функция должна проверять тип узла. В данном случае нас интересует только тип узла 1 (`Node.ELEMENT_NODE`). Для таких узлов мы должны перебирать в цикле их дочерние элементы. Для каждого дочернего элемента нам нужно проверить, соответствует ли он запросу, и сделать рекурсивный вызов для проверки его дочерних элементов.

Кошка и ее шляпа

Функции `Math.cos` и `Math.sin` измеряют углы в радианах, так что полный круг равен 2π. Для того чтобы получить противоположный угол для заданного угла, нужно прибавить к данному углу половину круга, то есть `Math.PI`. Это может быть полезно для размещения шляпы на противоположной стороне ее траектории.

Обработка событий

Воздушный шарик

Вам понадобится зарегистрировать обработчик для события `"keydown"` и отслеживать значение `event.key`, чтобы выяснить, была нажата клавиша ↑ или ↓.

Текущий размер шарика можно сохранить в привязке, чтобы затем вычислять новый размер на его основе. Будет полезно определить функцию, которая изменяет размер шарика — и привязку, и стиль всплывающей подсказки в DOM, — и вызывать эту функцию из обработчика событий, а также, возможно, один раз при запуске, чтобы установить начальный размер шарика.

Для того чтобы заменить всплывающее окно на взрывающийся шарик, нужно заменить текстовый узел на другой (используя метод `replaceChild`) или назначить свойству `textContent` родительского узла новую строку.

След мыши

Создавать элементы лучше всего в цикле. Чтобы элементы появились на экране, добавьте их в документ. Чтобы позже иметь к ним доступ для изменения их положения, возможно, стоит сохранить эти элементы в массиве.

Для того чтобы организовать циклическое переключение, можно сохранить переменную `counter` и увеличивать ее на 1 каждый раз, когда происходит событие `"mousemove"`. Затем с помощью оператора остатка от деления (`% elements.length`) можно получить правильный индекс массива, позволяющий выбрать элемент, на который должен указывать фокус во время данного события.

Еще один интересный эффект может быть достигнут путем моделирования простой физической системы. Используйте событие `"mousemove"` только для обновления пары привязок, которые отслеживают положение мыши. Затем задействуйте `requestAnimationFrame` для имитации замыкающих элементов, притягивающихся к позиции указателя мыши. На каждом шаге анимации обновляйте их положение на основе их позиции относительно указателя (и, возможно, скорости, которая сохраняется для каждого элемента). Ваша задача — найти хороший способ это сделать.

Вкладки

Одна из ловушек, с которой вы можете столкнуться при решении этой задачи, — невозможность напрямую использовать свойство `childNodes` узла как коллекцию узлов вкладок. Во-первых, поскольку это динамическая структура данных, при добавлении кнопки данная кнопка также становится

дочерним узлом, принадлежащим данному объекту. Во-вторых, текстовые узлы, играющие роль пробелов между узлами, также принадлежат `childNodes`, но не должны получать собственные вкладки. Чтобы игнорировать текстовые узлы, вместо `childNodes` можно использовать `children`.

Можно начать с создания массива вкладок, чтобы иметь к ним простой доступ. Чтобы реализовать стили кнопок, можно сохранить объекты, содержащие как панель вкладки, так и ее кнопку.

Я рекомендую написать отдельную функцию для смены вкладок. Вы можете сохранить ранее выбранную вкладку и изменить только стили, позволяющие ее скрыть и отобразить новую вкладку, или просто изменить стиль всех вкладок при выборе очередной.

Возможно, имеет смысл сразу вызвать эту функцию, чтобы интерфейс начал работать с первой видимой вкладки.

Проект: игровая платформа

Приостановка игры

Приостановка анимации может происходить при возврате `false` из функции, переданной в `runAnimation`. Для продолжения анимации можно снова вызвать `runAnimation`.

Итак, нам нужно сообщить функции, переданной в `runAnimation`, о том, что игра приостановлена. Для этого можно использовать привязку, к которой имеют доступ и обработчик события, и данная функция.

При поиске способа отменить регистрацию обработчиков, зарегистрированных посредством `trackKeys`, помните, что для успешного удаления обработчика необходимо передать в `removeEventListener` точно такое же значение функции, которое было передано в `addEventListener`.

Таким образом, значение функции `handler`, созданное в `trackKeys`, должно быть доступно для кода, отменяющего регистрацию обработчиков.

Вы можете добавить к объекту, возвращаемому `trackKeys`, свойство, содержащее либо значение этой функции, либо метод, непосредственно обрабатывающий отмену регистрации.

Монстр

Для того чтобы реализовать тип движения с отслеживанием состояния — например, отскок, — не забудьте сохранить необходимое состояние в объекте субъекта. Передайте это состояние в конструктор в качестве аргумента и добавьте его в качестве свойства.

Помните, что `update` возвращает новый объект, а не заменяет старый.

При обработке столкновения найдите игрока в `state.actors` и сравните его положение с положением монстра. Чтобы получить координаты низа игрока, нужно прибавить его вертикальный размер к вертикальной координате. Создание обновленного состояния будет похоже либо на метод столкновений (удаление актора), либо на метод лавы (изменение состояния на `"lost"`), в зависимости от позиции игрока.

Рисование на холсте

Фигуры

Чтобы нарисовать трапецию (1), проще всего использовать путь. Выберите подходящие координаты центра и прибавьте к нему все четыре вершины.

Ромб (2) можно нарисовать простым способом, задействуя путь, или интересным способом, с преобразованием `rotate`. Чтобы использовать поворот, вам придется применить прием, аналогичный тому, который мы применили в функции `flipHorizontally`. Поскольку нам нужно выполнить поворот относительно центра прямоугольника, а не точки (0, 0), нам нужно сначала переместиться в эту точку, затем выполнить поворот и вернуться обратно.

После визуализации любой фигуры обязательно сбросьте преобразование, использованное при ее создании.

Для построения зигзага (3) было бы непрактично вызывать `lineTo` для каждого отрезка в отдельности. Вместо этого лучше задействовать цикл. Вы можете на каждой итерации рисовать либо по два отрезка линии (правый и снова левый), либо один, и в этом случае нужно будет использовать

свойство четности (`% 2`) индекса цикла, чтобы определить, должен зигзаг двигаться влево или вправо.

Для построения спирали вам также понадобится цикл (4). Если рисовать серию точек, каждая из которых движется дальше по кругу вокруг центра спирали, то получим круг. Если на каждом проходе цикла изменять радиус круга, на котором размещается текущая точка, и обходить круг несколько раз, то результатом будет спираль.

Изображение звезды (5) строится из линий `quadraticCurveTo`. Можно также нарисовать звезду из прямых линий. Чтобы получить звезду с восемью лучами, разделите круг на восемь частей и т. д. Соедините эти точки линиями, изогнув их к центру звезды. Если рисовать звезду с помощью `quadraticCurveTo`, можно использовать центр в качестве контрольной точки.

Круговая диаграмма

Вам нужно вызвать `fillText` и выбрать такие значения свойств `textAlign` и `textBaseline` для контекста, чтобы текст заканчивался там, где нужно.

Было бы разумно разместить метки так, чтобы текст располагался на линии, проходящей из центра сектора до его середины. Вы едва ли захотите помещать текст прямо на боковую сторону круговой диаграммы, лучше сместить его от боковой части диаграммы на несколько пикселов.

Угол этой линии вычисляется как `currentAngle + 0,5 * sliceAngle`. Следующий код позволяет найти позицию на этой линии, отстоящую на 120 пикселов от центра:

```
let middleAngle = currentAngle + 0.5 * sliceAngle;
let textX = Math.cos(middleAngle) * 120 + centerX;
let textY = Math.sin(middleAngle) * 120 + centerY;
```

При использовании этого подхода для `textBaseline`, вероятно, лучше выбрать значение `"middle"`. Что использовать для `textAlign`, зависит от выбранной стороны круга. Если слева — то нужно выбрать `"right"`, а справа — `"left"`, чтобы текст не налезал на круговую диаграмму.

Если вы не знаете, как определить, с какой стороны круга находится данный угол, почитайте описание `Math.cos` в главе 14. Косинус угла говорит о том, какой x-координате он соответствует, а эта координата, в свою очередь, позволяет точно определить, с какой стороны круга мы находимся.

Прыгающий шарик

Нарисуйте коробку. Это легко сделать с помощью `stroRect`. Определите привязку, которая содержит ее размер, или две привязки, если ширина и высота коробки различаются. Чтобы создать круглый шар, начните путь и вызовите функцию `arc(x, y, radius, 0, 7)`, которая создает дугу, идущую от нуля до значения, позволяющего замкнуть круг. Затем заполните путь.

Чтобы смоделировать положение и скорость шарика, можно использовать класс `Vec` из главы 16. Задайте ему начальную скорость, лучше такую, чтобы направление движения было не чисто вертикальным или горизонтальным, и для каждого кадра умножьте эту скорость на прошедшее время. Когда шарик окажется слишком близко к вертикальной стенке, замените значение x-компонента скорости противоположным. Аналогично, когда шарик окажется у горизонтальной стенки, инвертируйте y-компонент.

После того как будет вычислена новая позиция и скорость шарика, используйте `clearRect`, чтобы удалить сцену и перерисовать ее, применяя новую позицию.

Заранее рассчитанное зеркальное отражение

Ключом к решению задачи является тот факт, что при использовании `drawImage` мы можем задействовать элемент `canvas` в качестве исходного изображения. Можно создать дополнительный элемент `<canvas>`, не добавляя его в документ, и нарисовать перевернутые спрайты один раз. При рисовании реального кадра мы просто копируем уже перевернутые спрайты на основной холст.

Изображения загружаются не мгновенно, поэтому потребуется некоторая осторожность. Мы создаем перевернутое изображение только один раз, и если сделаем это до загрузки изображения, то ничего не будет нарисовано. Для рисования зеркально перевернутых изображений на дополнительном холсте можно применить обработчик события `"load"` для изображения. Этот холст можно сразу использовать в качестве источника данных для рисования (он просто будет пустым, пока мы не нарисуем на нем персонаж).

HTTP и формы

Согласование содержимого

При написании своего кода используйте примеры с функцией `fetch`, описанные ранее в этой главе.

Запрос неправильного типа мультимедиа вернет ответ с кодом 406 (Not acceptable). Данный код сервер возвращает, когда не может выполнить заголовок `Accept`.

Среда разработки JavaScript

Для того чтобы получить доступ к элементам, определенным в HTML, можно воспользоваться методами `document.querySelector` или `document.getElementById`. Обработчик событий `"click"` или `"mousedown"` для кнопки позволяет получить свойство `value` текстового поля и вызвать для него функцию.

Не забудьте обернуть вызов `Function` и вызов результата `Function` в блок `try`, чтобы перехватывать возможные исключения. В данном случае мы действительно не знаем, какой тип исключения может получиться, поэтому будем перехватывать все.

Свойство `textContent` выходного элемента можно использовать для заполнения его строковым сообщением. Или же, если вы хотите сохранить старое содержимое, создайте новый текстовый узел, используя `document.createTextNode`, и добавьте его к элементу.

Не забудьте добавить в конец символ новой строки, иначе все выходные данные будут отображаться в одной строке.

Игра «Жизнь» Конвея

Чтобы решить проблему одновременного концептуального изменения, постарайтесь рассматривать вычисление одного поколения как чистую функцию, которая принимает одну сетку и создает новую, соответствующую следующему ходу.

Матрица может быть представлена способом, описанным в главе 6. Чтобы подсчитать живых соседей, можно построить два вложенных цикла, перебирая смежные координаты по обоим измерениям. Постройте код так, чтобы

не считать клетки, выходящие за пределы поля, и игнорировать центральную ячейку, соседей которой мы рассматриваем.

Чтобы изменения флажков вступили в силу для следующего поколения, можно пойти двумя путями. Обработчик событий может реагировать на эти изменения и обновлять текущую сетку, отражая их, или же можно построить новую сетку из значений флажков перед вычислением следующего хода.

Если вы решите использовать обработчики событий, то, возможно, захотите прикрепить атрибуты, идентифицирующие позицию, которой соответствует каждый флажок, чтобы можно было легко определить, какую ячейку следует изменить.

Чтобы нарисовать сетку флажков, можно использовать элемент `<table>` (см. главу 14) или просто поместить все флажки в один элемент, разделив строки элементами `<br>` (разрыв строки).

Проект: растровый графический редактор

Клавиатурные привязки

Свойством `key` для событий буквенных клавиш будет сама строчная буква, если не нажата клавиша Shift. В данном случае нас не интересуют события клавиш при нажатой Shift.

Обработчик события `"keydown"` может проверять свой объект события, чтобы определить, соответствует ли он одному из сочетаний клавиш. Чтобы не выписывать весь перечень, можно автоматически получить список первых букв из объекта `tools`.

Если событие клавиши совпадает с ярлыком, вызовите для него `preventDefault` и передайте управление соответствующему действию.

Эффективное рисование

Это упражнение — хороший пример того, как можно ускорить код, используя неизменяемые структуры данных. Поскольку у нас есть и старое, и новое изображение, мы можем сравнить их и перерисовать только те пикселы, которые изменили цвет, что в большинстве случаев экономит более 99 % затрат на рисование.

Вы можете написать новую функцию `updatePicture` или изменить функцию `drawPicture` так, чтобы она принимала дополнительный аргумент, который может быть неопределенным или содержать предыдущее изображение. Для каждого пиксела функция проверяет, имела ли указанная точка такой же цвет в предыдущем изображении, и, если это так, пропускает данный пиксел.

Поскольку при изменении размера холст очищается, вам также следует избегать изменения его свойств `width` и `height`, если старое и новое изображения имеют одинаковый размер. Если они различаются — что произойдет при загрузке нового изображения, — то после изменения размера холста можно присвоить привязке, сохраняющей старое изображение, значение `null`, потому что после изменения размера холста нам не нужно будет сравнивать пикселы.

Круги

Поищите вдохновение в реализации инструмента `rectangle`. Как и в этом инструменте, при перемещении указателя вы будете продолжать рисовать на *начальном*, а не на текущем изображении.

Чтобы определить, какие именно пикселы нужно перекрасить, можно воспользоваться теоремой Пифагора. Вначале определите расстояние между текущей и начальной позициями указателя, вычислив квадратный корень (`Math.sqrt`) из суммы квадратов (`Math.pow(x, 2)`) разности x- и y-координат. Затем переберите в цикле пикселы, принадлежащие квадратной области, в центре которой находится начальная позиция. Стороны этого квадрата должны быть по крайней мере вдвое больше радиуса окружности. Раскрасьте те из пикселов, которые находятся внутри радиуса круга, снова используя формулу Пифагора, чтобы вычислить их расстояние от центра.

Убедитесь, что не пытаетесь раскрасить пикселы, которые находятся за пределами изображения.

Правильные линии

Проблема рисования растровых линий в том, что на самом деле это четыре похожие, но немного разные проблемы. Нарисовать горизонтальную линию слева направо очень легко: нужно просто перебрать в цикле x-координаты и на каждом шаге поменять цвет пиксела. Если линия имеет небольшой наклон (менее 45 градусов или $1/4\pi$ радиана), можно интерполировать y-координату вдоль наклона. Вам по-прежнему нужен один пиксел на каждую x-координату, а y-координата этих пикселов определяется наклоном.

Но как только наклон превышает 45 градусов, способ обработки координат приходится изменить. Теперь, поскольку линия меняется по вертикали сильнее, чем по горизонтали, у нас приходится по одному пикселу на каждую y-координату. А после того как вы пересечете отметку 135 градусов, вам снова придется вернуться к циклу с перебором по x-координате, но теперь уже справа налево.

На самом деле вам не нужно писать четыре цикла. Поскольку при рисовании линии из точки A в точку B получится то же самое, что и при рисовании линии из точки B в точку A, вы можете поменять местами начальную и конечную позиции для линий, идущих справа налево, и рассматривать их как идущие слева направо.

Таким образом, вам нужно только два разных цикла. Прежде всего функция рисования линий должна проверить, что больше: разность x-координат или разность y-координат. Если разность x-координат больше, то эта линия ближе к горизонтальной, а если нет — то к вертикальной.

Убедитесь, что вы сравниваете абсолютные значения разности x- и y-координат, которые можно получить с помощью функции `Math.abs`.

После того как вы определите, по какой оси будет выполняться цикл, можно проверить, имеет ли начальная точка более высокую координату вдоль данной оси, чем конечная точка, и при необходимости поменять эти точки местами. В JavaScript есть краткий способ поменять значения двух привязок с использованием деструктурирующего присваивания:

```
[start, end] = [end, start];
```

Затем можно вычислить наклон линии, который определяет величину изменения координаты по другой оси на каждом шаге вдоль основной оси. Тогда можно будет запустить цикл вдоль главной оси, одновременно отслеживая соответствующую позицию на другой оси, и рисовать пикселы на каждой итерации. Не забудьте округлять координаты по неосновной оси, так как они могут быть дробными, а метод `draw` плохо реагирует на дробные координаты.

Node.js

Инструмент поиска

Регулярное выражение, которое является первым аргументом командной строки, вы найдете в `process.argv[2]`. После него идут входные файлы. Для

преобразования строки в объект регулярного выражения можно использовать конструктор `RegExp`.

Проще всего было бы сделать это синхронно с помощью `readFileSync`, но если вы снова примените `fs.promises`, чтобы получить функции, возвращающие промис, и напишете асинхронную функцию, то код будет выглядеть аналогично.

Чтобы выяснить, является ли то, что вам передали, каталогом, можно снова использовать `stat` (или `statSync`) и метод `isDirectory` объекта `stat`.

Просмотр каталога — это процесс с ветвлением. Можно организовать ветвление либо с помощью рекурсивной функции, либо сохраняя рабочий массив (файлы, которые еще нужно просмотреть). Чтобы получить список файлов каталога, можно вызвать `readdir` или `readdirSync`. Странное использование заглавных букв вызвано тем, что именование функций файловой системы Node построено на основе стандартных функций Unix, таких как `readdir`, которые пишутся строчными буквами, но затем в них добавляется слово `Sync`, начинающееся с заглавной буквы.

Чтобы перейти от имени файла, прочитанного с помощью `readdir`, к полному имени пути, нужно объединить имя файла с именем каталога, поставив между ними символ косой черты (`/`).

Создание каталога

В качестве основы для метода `MKCOL` можно использовать функцию, которая реализует метод `DELETE`. Если файл не найден, попробуйте создать каталог с помощью `mkdir`. Если по этому пути уже существует каталог, можно вернуть ответ с кодом 204, чтобы запросы на создание каталога были идемпотентными. Если по такому пути существует файл, не являющийся каталогом, верните код ошибки. В этом случае будет уместным код 400 («неверный запрос»).

Публичное пространство в сети

Для хранения содержимого редактируемого файла можно создать элемент `<textarea>`. Чтобы получить текущее содержимое файла, можно применить запрос `GET`, использующий `fetch`. Чтобы ссылаться на файлы, размещенные на том же сервере, что и выполняемый сценарий, вместо URL-адресов вида `http://localhost:8000/index.html` можно применять относительные URL-адреса, такие как `index.html`.

Для того чтобы сохранить файл, после того как пользователь нажмет на кнопку (можно использовать элемент `<form>` и событие `submit`), сделайте `PUT`-запрос к тому же URL-адресу, указав в качестве тела запроса содержимое `<textarea>`.

Затем можно добавить элемент `<select>`, содержащий все файлы из верхнего каталога сервера, где элементы `<option>` будут содержать строки, возвращаемые GET-запросом по URL-адресу `/`. Когда пользователь выберет другой файл (событие `"change"` для поля `<select>`), сценарий должен извлечь и отобразить этот файл. При сохранении файла используйте его текущее выбранное имя.

Проект: сайт по обмену опытом

Хранение на диске

Самое простое решение, которое я смог придумать, — закодировать весь объект `talks` в формате JSON и записать его в файл с помощью `writeFile`. Для этого при каждом изменении данных сервера нужно вызывать уже существующий метод (`updated`). Его можно расширить для записи новых данных на диск.

Выберите имя файла, например `./talks.json`. При запуске сервер может попытаться прочитать этот файл с помощью `readFile`, и если это удастся, то использовать содержимое файла в качестве начальных данных.

Однако будьте осторожны. Объект `talks` сначала не имеет прототипа, так что для него можно спокойно применять оператор `in`. Но `JSON.parse` будет возвращать обычные объекты с прототипом `Object.prototype`. Если вы используете JSON в качестве формата файла, то вам придется скопировать свойства объекта, возвращаемого `JSON.parse`, в новый объект, не имеющий прототипа.

Сброс поля комментариев

Наилучший способ сделать это, вероятно, состоит в том, чтобы создать объекты компонента беседы с помощью метода `syncState`, чтобы их можно было обновлять для отображения измененной версии беседы. В обычном

режиме работы единственный способ изменить беседу — добавить новые комментарии, поэтому метод `syncState` может быть относительно простым.

Сложность состоит в том, что, когда приходит измененный список бесед, нам нужно согласовать существующий список DOM-компонентов с беседами из нового списка: удалить компоненты, если соответствующие беседы были удалены, и изменить компоненты, для которых изменилось содержание бесед.

Для этого может быть полезным сохранить структуру данных, где хранятся компоненты беседы, под заголовками этих бесед, чтобы можно было легко выяснить, существует ли компонент для данной беседы. Затем можно перебрать в цикле новый массив бесед и для каждой из них либо синхронизировать существующий компонент, либо создать новый. Чтобы удалить компоненты для удаленных бесед, вам также придется перебрать в цикле компоненты и проверить, существуют ли соответствующие беседы.

Овладейте языком Web

JavaScript лежит в основе практически всех современных веб-приложений — от социальных сетей до браузерных игр. Это гибкий, выразительный язык, позволяющий создавать полномасштабные приложения.

«Выразительный JavaScript» позволит глубоко погрузиться в тему, научиться писать красивый и эффективный код. Вы познакомитесь с синтаксисом, стрелочными и асинхронными функциями, итератором, шаблонными строками и блочной областью видимости.

Марейн Хавербеке — практик. Получайте опыт и изучайте язык на множестве примеров, выполняя упражнения и учебные проекты. Сначала вы познакомитесь со структурой языка JavaScript, управляющими структурами, функциями и структурами данных, затем изучите обработку ошибок и исправление багов, модульность и асинхронное программирование, после чего перейдете к программированию браузеров.

Прочитав эту книгу, вы:

- Познакомитесь с основами синтаксиса, управления и данных.
- Научитесь создавать чистый код, используя технологии объектно-ориентированного и функционального программирования.
- Узнаете, как эффективно использовать DOM при взаимодействии с браузером.
- Научитесь писать сценарии для браузеров и разрабатывать веб-приложения.
- Освоите Node.js для разработки серверов и утилит.

Не пора ли поближе познакомиться с языком для разработки в среде Web?

Весь исходный код доступен онлайн в интерактивной «песочнице», где его можно редактировать и запускать, чтобы сразу видеть результат.

Об авторе

Марейн Хавербеке — программист и писатель. Энтузиаст многих технологий и языков программирования, работающий с большим количеством систем, от баз данных до компиляторов. Является владельцем небольшого бизнеса, основанного на его проектах с открытым кодом.

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/piterbooks
youtube.com/ThePiterBooks
vk.com/piterbooks
facebook.com/piterbooks

ISBN: 978-5-4461-1226-5

no starch press